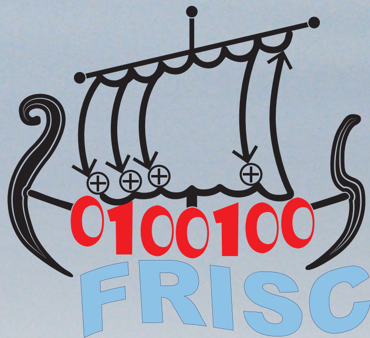
The International Conference on PASSWORDS 2014

December 8-10
Trondheim Norway



NTNU – Trondheim
Norwegian University of
Science and Technology

© Foto: Solveig Mjølsnes

Preface

This volume contains the conference pre-proceedings for Passwords'14: The 7th International Conference on Passwords held on December 7-8, 2014 at NTNU in Trondheim.

We received 30 submissions both tutorials and papers. Each submission was reviewed by at least 2, and on the average 2.7, program committee members. The committee decided to accept 21 of them for presentations. The program also includes 4 invited talks.

December 7, 2014	Stig Mjolsnes
NTNU, TRONDHEIM	Program Chair

Table of Contents

Introducing the PRINCE attack-mode 1
Jens Steube

Distributed, Stealthy Brute Force Password Guessing Attempts - Slicing and Dicing Data from Recent Incidents 43
Peter Hansteen

Crypto vs Physical Access 44
Sebastien Raveau

Universal 2nd Factor Authentication 45
Simon Josefsson

Password-manager friendly (PMF): Semantic annotations to improve the effectiveness of password managers 46
Max Spencer, Frank Stajano and Graeme Jenkinson

charPattern: Rethinking Android Lock Pattern to Adapt to Remote Authentication 60
Kemal Bicakci and Tashtanbek Satiev

FabulaRosa and the Five New Protocols 74
Christine Ziske and Ulf Ziske

Behavioral Biometrics as a Mechanism for Delaying the Obsolescence of Traditional Passwords 104
Elaine Wooton

mimikatz, or how to push Microsoft to change some little stuff 105
Benjamin Delpy

Unrevealed patterns in password databases: analyses of cleartext passwords 106
Norbert Tihanyi, Attila Kovács, Gergely Vargha and Adam Lenart

Gathering and Analyzing Identity Leaks for Security Awareness 119
David Jaeger, Hendrik Graupner, Andrey Sapegin, Feng Cheng and Christoph Meinel

Passwords - Divided they Stand, United they Fall 135
Harshal Tupsamudre, Vijayanand Banahatti and Sachin Lodha

Overview of the Candidates for the Password Hashing Competition - And their Resistance against Garbage-Collector Attacks 151
Jakob Wenzel, Christian Forler, Eik List and Stefan Lucks

Cryptographic module based approach for password hashing schemes 168
Donghoon Chang, Arpan Jati, Sweta Mishra and Somitra Sanadhya

On Password Guessing with GPUs and FPGAs 185
Markus Dürmuth and Thorsten Kranz

SlidePIN: Slide-Based PIN Entry Mechanism on a Smartphone 206
Huiping Sun, Shuaiying Guo, Ke Wang, Nan Qin and Zhong Chen

Private Password Auditing 223
Amrit Kumar and Cedric Lauradoux

SAVVIcode: Preventing Mafia Attacks on Visual Code Authentication Schemes 229
Jonathan Millican and Frank Stajano

WPS Insecurity 235
Dominique Bongard

PassCue: the Shared Cues system in Practice 236
Mats Sandvoll, Colin Boyd and Bjørn B. Larsen

Time Memory Tradeoff Analysis of Graphs in Password Hashing Constructions 258
Donghoon Chang, Arpan Jati, Sweta Mishra and Somitra Kumar Sanadhya

PRINCE

modern password guessing algorithm

Why do we need a new attack-mode?

FUTURE OF PASSWORD HASHES

Future of modern password hashes

Feature	Effect
High iteration count	Slow
Salted	Rainbow-Tables resistance
Memory-intensive	GPU resistance
Configurable parameters	Slow
Anti-Parallelization	Slow
…	

Algorithms used for password hashing, by performance*

Name	Speed
NTLM, MD5, SHA1-512, Raw-Hashes	1 BH/s - 10 BH/s
Custom (Salt): VBull, IPB, MyBB	100 MH/s - 1 BH/s
DEScrypt	10 MH/s - 100 Mh/s
MD5crypt	1 MH - 10 MH/s
TrueCrypt, WPA/WPA2 (PBKDF2)	100kH/s - 1 MH/s
SHA512crypt, Bcrypt (Linux/Unix)	10kH/s - 100 kH/s
Custom (Iteration): Office, PDF, OSX	1kH/s - 10 kH/s
Scrypt (RAM intensive): Android 4.4+ FDE	< 1 kH/s

* Performance oclHashcat v1.32
Single GPU
Default settings for configurable algorithms

Effects of modern password hashes

- Obsolete attack-modes:
  - Brute-Force-attack
  - Rainbow-Tables

So, what can the attacker do?

REMAINING ATTACK VECTORS

Remaining attack vectors

- Hardware (FPGA/ASIC)
- Extract keys from RAM
- Efficiency

Remaining attack vectors

- Hardware (FPGA/ASIC)
- Extract keys from RAM
- Efficiency

- Easier to cool
- Lower power consumption
- Easier to cluster
- Clustering only linear
- Expensive development
- Unflexible?

Remaining attack vectors

- Hardware (FPGA/ASIC)
- Extract keys from RAM
- Efficiency

- Highest chance of success
- Requires physical access to the System
- System must run

Remaining attack vectors

- Hardware (FPGA/ASIC)
- Extract keys from RAM
- Efficiency

- Exploit human weakness:
  - Psychology aspects
  - Password reuse
  - Pattern
- Limited keyspace
- Using rules:
  - Limited pattern
  - Takes time to develop

Features and advantages compared to previous attack modes

PRINCE ATTACK

Advantages over other Attack-Modes

- Simple to use, by design
- Smooth transition
- Makes use of unused optimizations:
  - Time works for attacker
  - Personal aspects

Advantages over other Attack-Modes

- Simple to use, by design
- Smooth transition
- Makes use of unused optimizations:
  - Time works for attacker
  - Personal aspects
- No monitoring required
- No extension required
- No syntax required

Advantages over other Attack-Modes

- Simple to use, by design
- Smooth transition
- Makes use of unused optimizations:
  - Time works for attacker
  - Personal aspects

- Primary goal of the algorithm
- Starts with highest efficiency
  - Wordlist
  - Hybrid
  - Keyboard walks / Passphrases
  - Brute-Force + Markov
- Not a scripted batch

Advantages over other Attack-Modes

- Simple to use, by design
- Smooth transition
- Makes use of unused optimizations:
  - Time works for attacker
  - Personal aspects
- Does not run out of (good) wordlists
  - Time-consuming monitoring
- Does not need ideas
  - Time-consuming extension

Advantages over other Attack-Modes

- Simple to use, by design
- Smooth transition
- Makes use of unused optimizations:
  - Time works for attacker
  - Personal aspects

- Personal Aspects
  - Religion
  - Political wing
  - Red car
- Not hobbies, friends, dates, …
  - Already covered with Wordlist-Attack
  - Common knowledge not to use them

Algorithm details

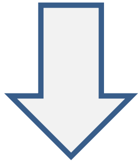
PRINCE ATTACK

PRINCE-attack

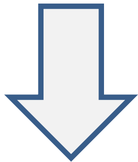
- PRobability
- INfinite
- Chained
- Elements

Attack basic components

- Element



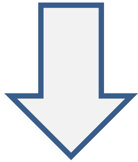
- Chain



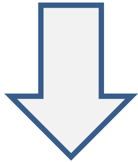
- Keyspace

Attack basic components

- Element



- Chain



- Keyspace

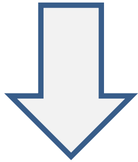
- Smallest entity
- An unmodified line (word) of your wordlist
- No splitting / modification of the line
- Sorted by their length into element database

Element example

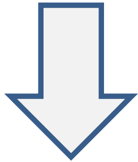
- 123456 → Table: 6
- password → Table: 8
- 1 → Table: 1
- qwerty → Table: 6
- … → …

Attack basic components

- Element



- Chain



- Keyspace

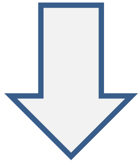
- Sum of all elements lengths in a chain = chain output length
- Fixed output length
- Best view on this is in reverse order, eg. a chain of length 8 can not hold an element of length 9

Chains example, general

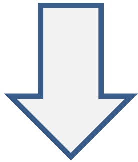
- Chains of output length 8 consists of the elements
  - 8
  - 2 + 6
  - 3 + 5
  - …
  - 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
- Number of chains per length = 2 ^ (length - 1)

Attack basic components

- Element



- Chain



- Keyspace

- Number of candidates that is getting produced, per chain
- Different for each chain
- The product of the count of the elements which build the chain

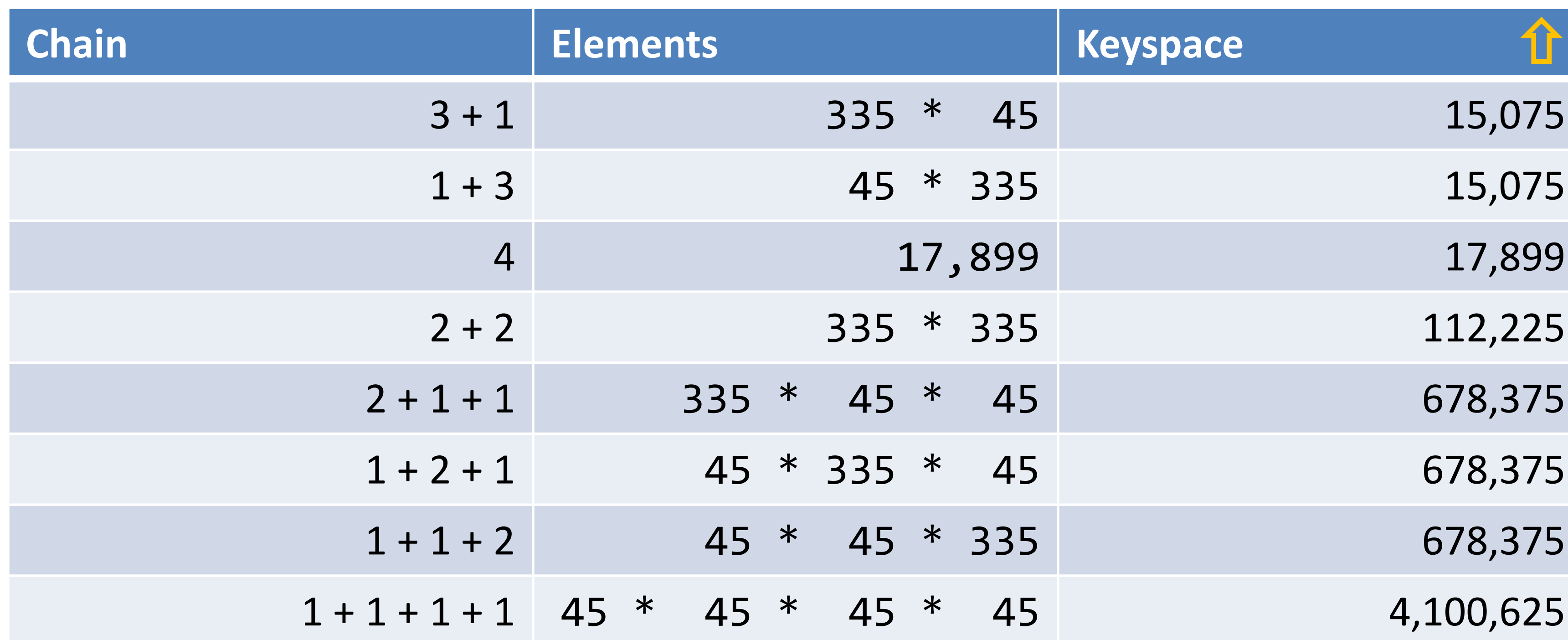
Element example (rockyou)

- length 1: 45
- length 2: 335
- length 3: 2461
- length 4: 17899
- …

Keyspaces of chains of length 4 (rockyou)

Chain	Elements	Keyspace
4	17,899	17,899
1 + 1 + 1 + 1	45 * 45 * 45 * 45	4,100,625
1 + 1 + 2	45 * 45 * 335	678,375
1 + 2 + 1	45 * 335 * 45	678,375
1 + 3	45 * 335	15,075
2 + 1 + 1	335 * 45 * 45	678,375
2 + 2	335 * 335	112,225
3 + 1	335 * 45	15,075

Keyspaces of chains of length 4 (rockyou)



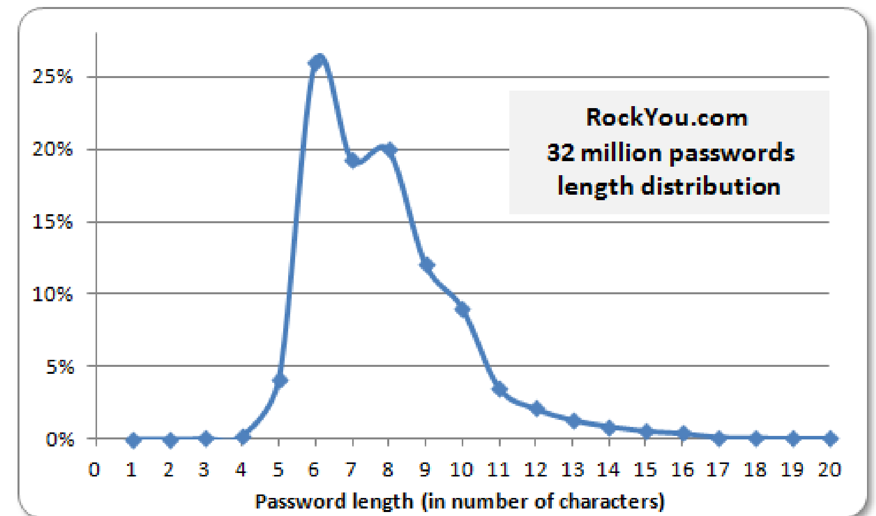
Chain	Elements	Keyspace ⇧
3 + 1	335 * 45	15,075
1 + 3	45 * 335	15,075
4	17,899	17,899
2 + 2	335 * 335	112,225
2 + 1 + 1	335 * 45 * 45	678,375
1 + 2 + 1	45 * 335 * 45	678,375
1 + 1 + 2	45 * 45 * 335	678,375
1 + 1 + 1 + 1	45 * 45 * 45 * 45	4,100,625

Keyspace selection, general

- Sorting by lowest keyspace creates the floating effect inside the prince attack-mode:
  - Wordlist
  - Hybrid
  - Keyboard walks / Passphrases
  - Brute-Force + Markov

Candidate output length selection

- The Algorithm has to chose the order of the output length for candidates
- Word-length distribution in a wordlist is a known structure
- The algorithm recreates its own stats from the input wordlist



http://blog.erratasec.com/

Personal aspects

- To make use of this feature, you need a specific wordlist
  - Use a tool like wordhound to compile such a wordlist (grabs data from URL, twitter, reddit, etc)
- Cookbook phase:
  - Decide yourself if you want to use the raw list or
    - Preprocess the wordlist with some rules applied
    - Mix in like top 10k from rockyou
    - Mix in some single chars for late BF

Problems of the attack

- Elements in the wordlist requires all lengths
- Chain-count for long outputs
- Generated dupes

Problems of the attack

- Elements in the wordlist requires all lengths
- Chain-count for long outputs
- Generated dupes

- For calculation length distribution

Problems of the attack

- Elements in the wordlist requires all lengths
- Chain-count for long outputs
- Generated dupes

- Can be suppressed with divisor parameter

Problems of the attack

- Elements in the wordlist requires all lengths
- Chain-count for long outputs
- Generated dupes

Princeprocessor internal

- Load words from wordlist
- Store words in memory
- Generate element chains for each password length
  - Reject chains that does include an element which points to a non-existing password length
- Sort chained-elements by keyspace of the chain
- Iterate through keyspace (mainloop)
  - Select the next chain of that password length
  - Generate password with chain
  - Print

Usage

PRINCE ATTACK

How to use it from users view

- Download princeprocessor
- Choose an input wordlist which could be:
  - One of your favourite wordlist (rockyou, etc…)
  - Target-specific optimized wordlist
- Pipe princeprocessor to your cracker
  - ./pp64 < wordlist.txt | ./oclHashcat hash.txt

How to use it from users view

- Optionally
  - Choose password min / max length
  - Choose character classes to pass / filter
  - Choose start / stop range -> Distributed
  - Choose minimum element length
  - Choose output file, otherwise written to STDOUT

LIVE DEMO 1

- Wordlist
  - Top 100k of rockyou.txt
- Hashlist
  - Public leak „stratfor“, 822k raw MD5 hashes
- Preparation
  - Removing raw dictionary hits first

LIVE DEMO 2

- Wordlist
  - Generated by scraping stratfor site
- Hashlist
  - Public leak „stratfor“, 822k raw MD5 hashes
- Preparation
  - Removing raw dictionary hits first

Download from: https://hashcat.net/tools/princeprocessor/

- Linux
- Windows
- OSX

PRINCEPROCESSOR V0.10 RELEASE

Email: jens.steube@gmail.com

IRC: freenode #hashcat

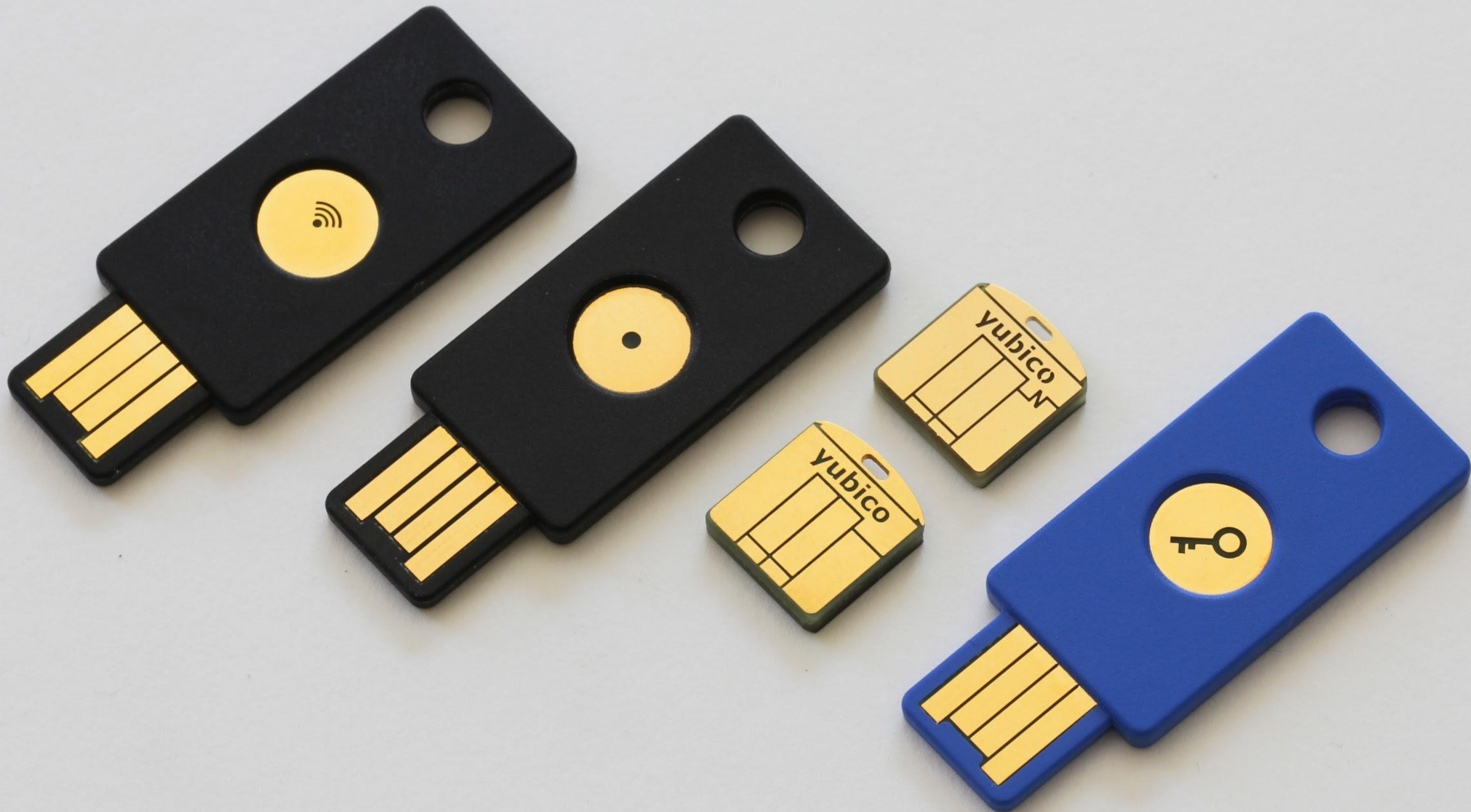
THANKS! QUESTIONS?

yubico

Universal 2nd Factor

2014-12-08 Trondheim Norway
Passwords 2014
Simon Josefsson

yubico


yubico
yubico

FOSS and open protocol friendly - see developers.yubico.com

yubico

HAPPY MOTHER'S DAY, MOMMY!
Look what I made for you - more secure passwords!
www.ITworld.com
PJ

Pre-History of U2F: Gnubby

Yubico designed a precursor to U2F with Google and NXP. Deployed to Google staff around the world.

To reach mass market, standardization and multiple vendors are needed. During 2012 the FIDO Alliance started working on U2F.

Bank of America.

IdentityX

Google

DISCOVER

PayPal™

QUALCOMM

BlackBerry.

ARM

oberthur TECHNOLOGIES THE M COMPANY

lenovo. FOR THOSE WHO DO.™

yubico Trust the Net.

Synaptics

Over 150 Members

NXP

Microsoft

CrucialTec

MasterCard

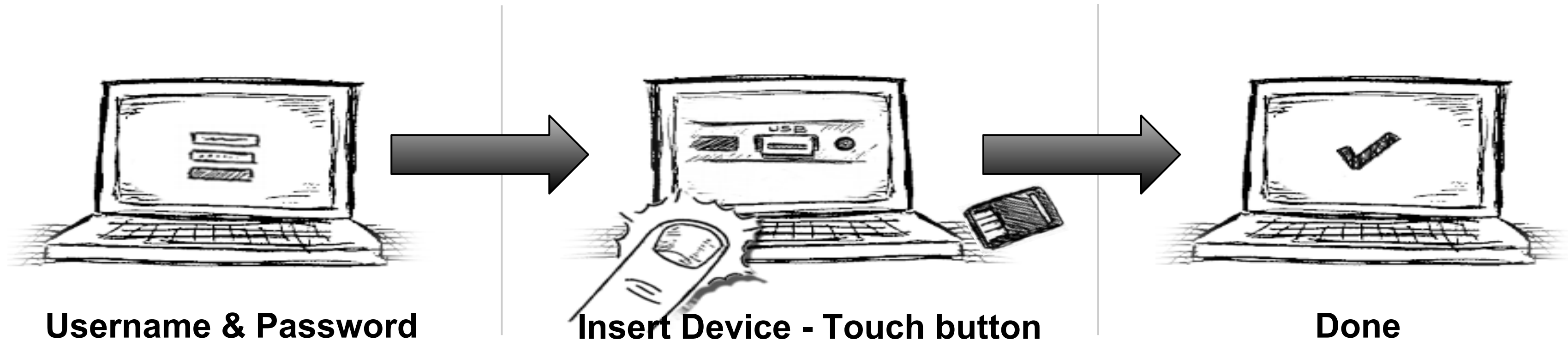
Nok Nok LABS

VISA

Alibaba Group

RSA

U2F: Doing 2FA the Easy Way



What is this U2F Protocol?

Core Idea: Standard Public Key Cryptography:

- User's device mints new key pair, gives public key and "keyhandle" to server
- Server asks user's device to sign data to verify the user
- One device, many services - "**Bring Your Own Authenticator**"

Design Considerations:

- **Privacy**: Site Specific Keys, No unique ID per device
- **Security**: No phishing, Man-In-The-Middle
- **Trust**: User decides what authenticator to use
- **Pragmatics**: Affordable today
- **Usability**: No delays, Fast crypto on device

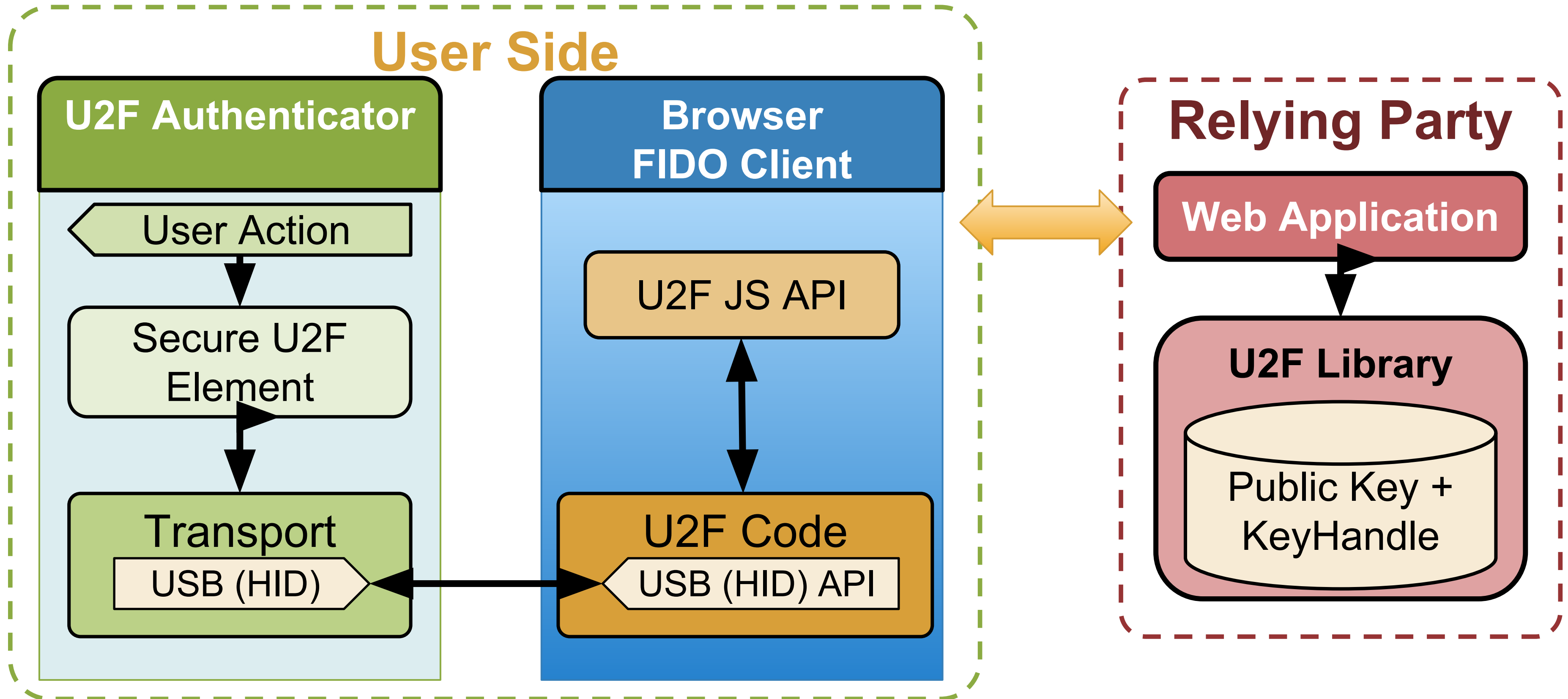
yubico

THINK:

Driverless Smartcard re-designed for the Modern Consumer Web

yubico

U2F Entities



USB today, the world tomorrow…

NFC

yubico

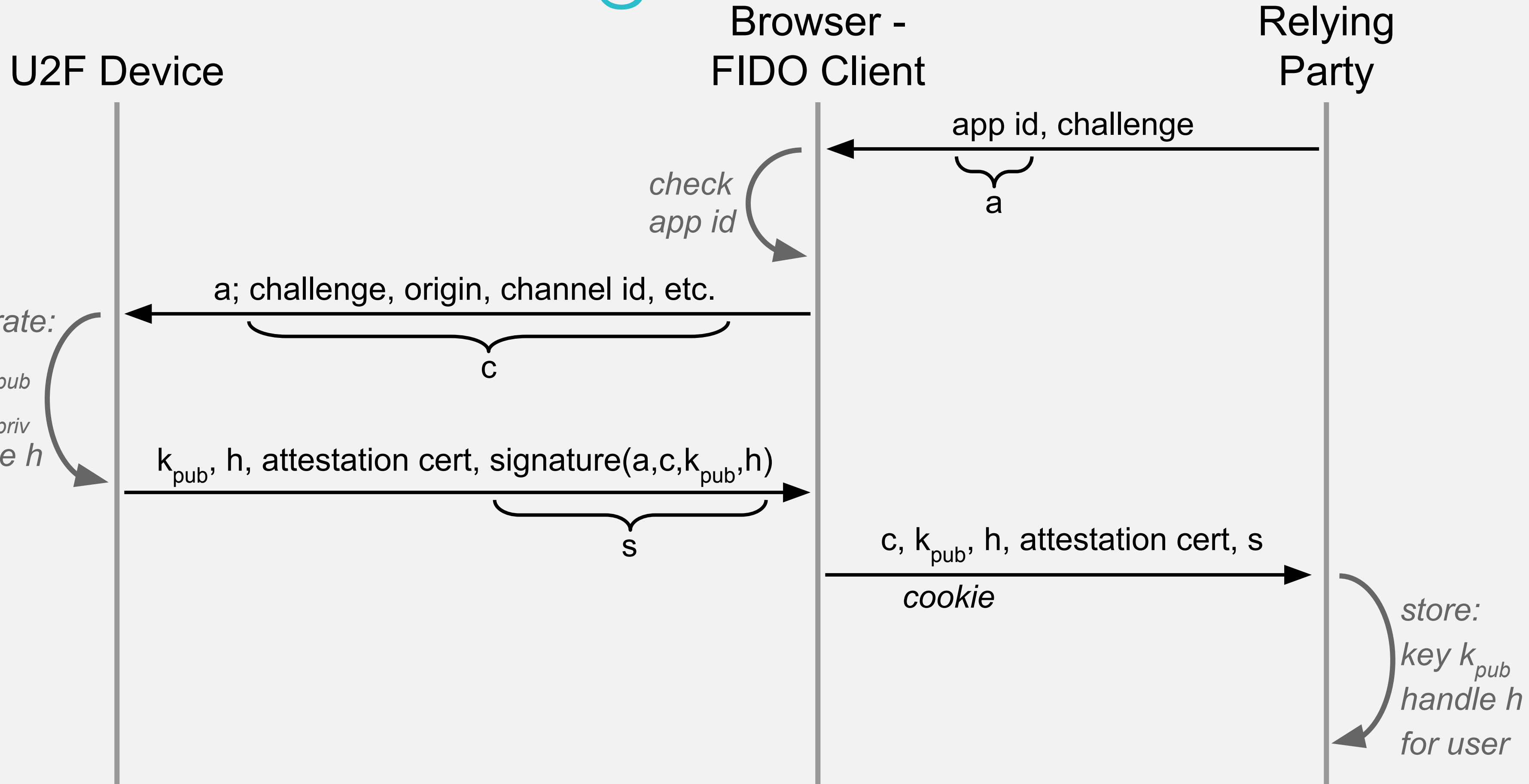
U2F Concepts

- Registration
  - User binds U2F device to her account
  - Server gets a device certificate
  - Server stores public key and keyhandle for the user

- Authentication
  - Normal username+password process - retrieve keyhandle
  - User uses already registered U2F device to login
  - Server gets signed blob and compares with public key

yubico

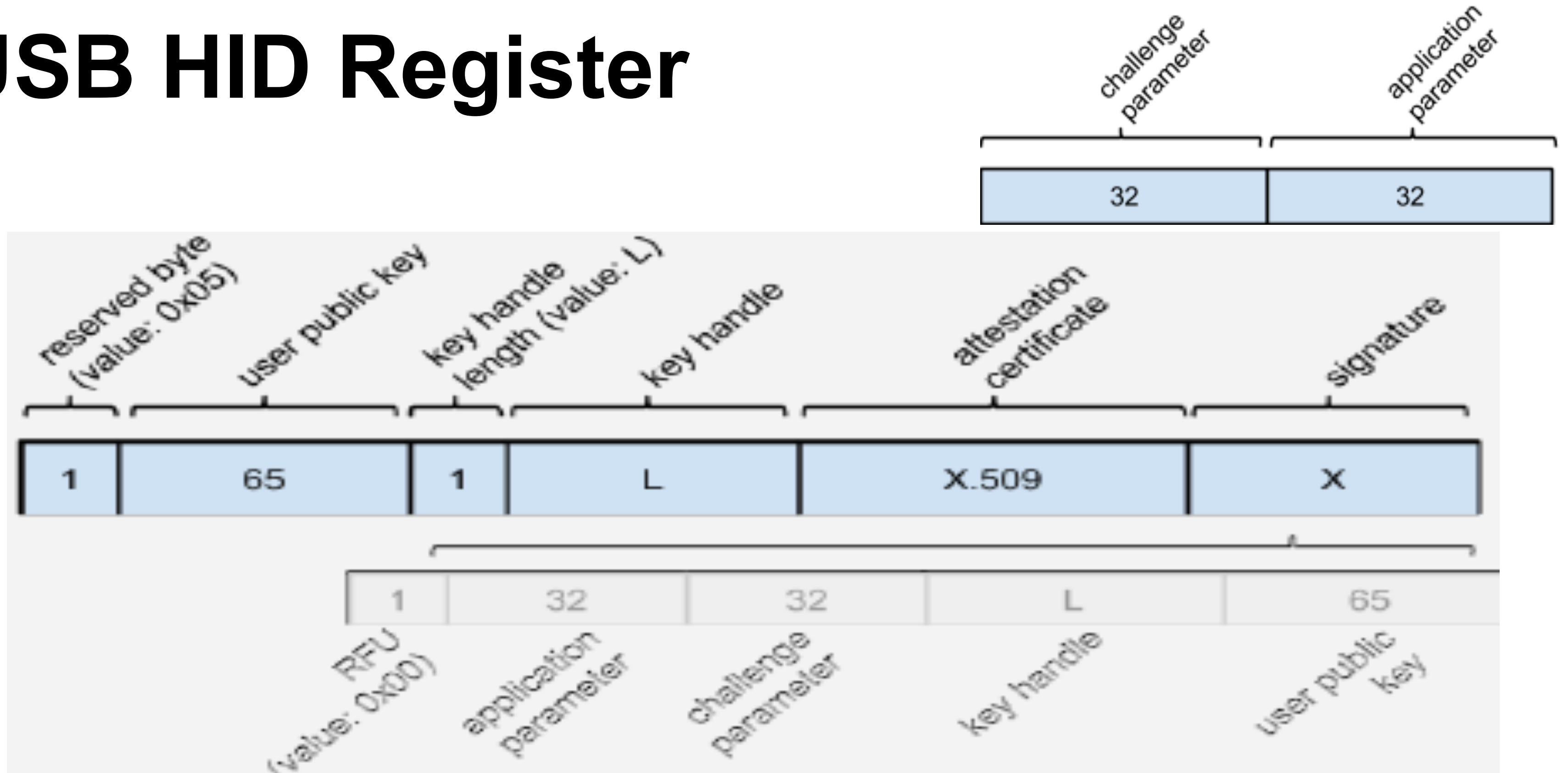
Registration



U2F Register JavaScript API

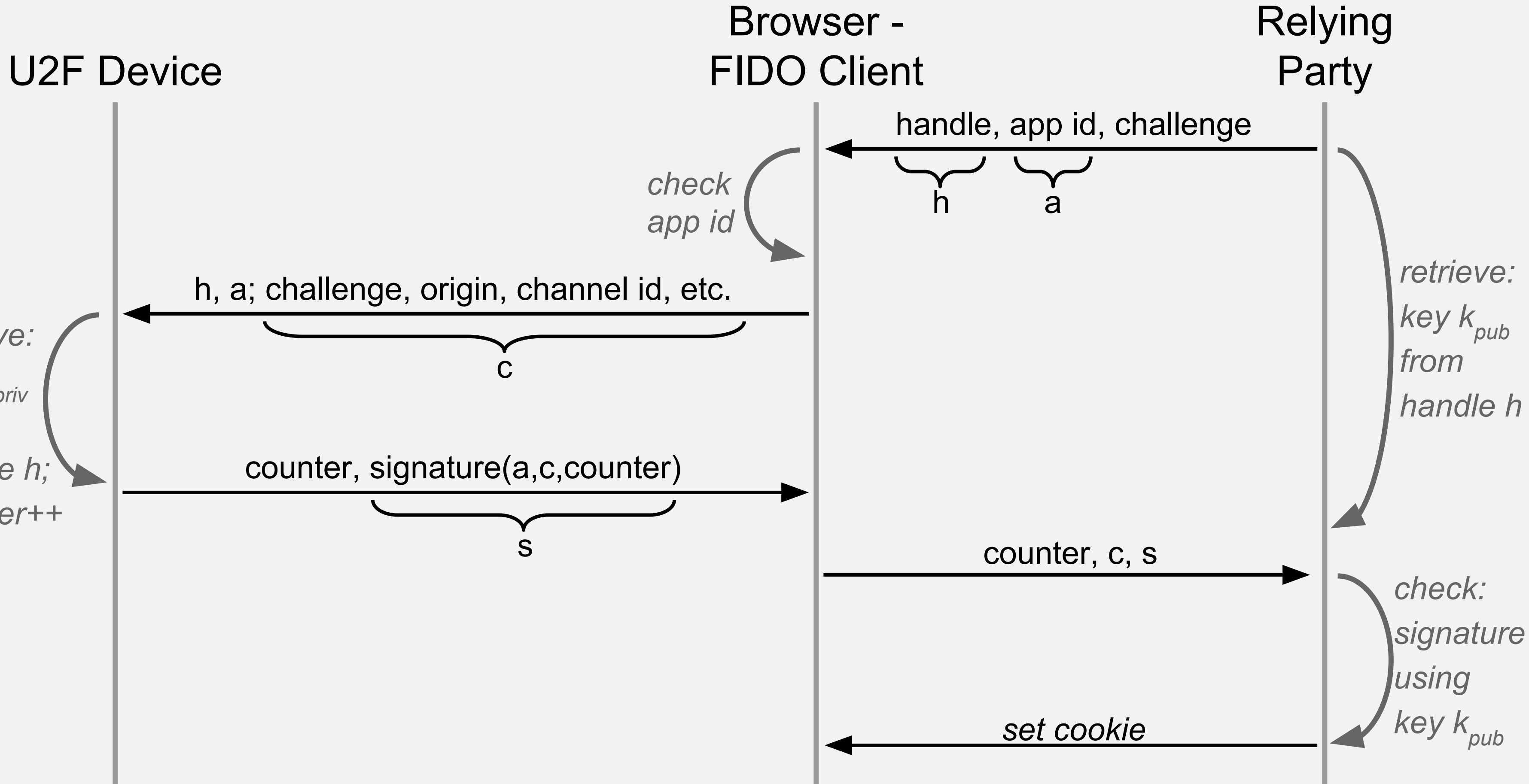
```
navigator.handleRegistrationRequest({
    ‘challenge’: ‘KSDJsdASAS-AIS_AsS’,
    ‘app_id’: ‘https://www.acme.com/facets.json’
        }, callback);

callback = function(response) {
       sendToServer(
    response[‘clientData’],
    response[‘registrationData’]);
};
```

USB HID Register



yubico

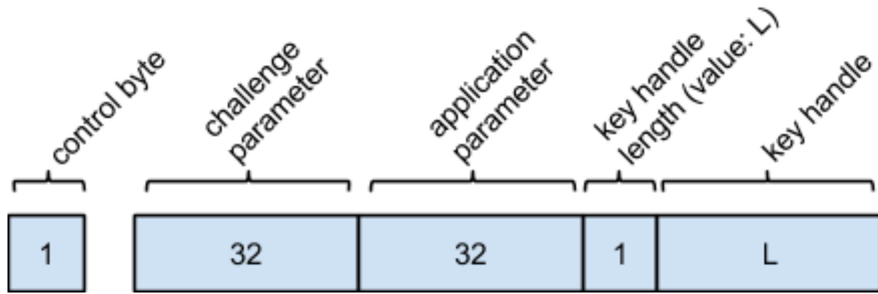
Authentication

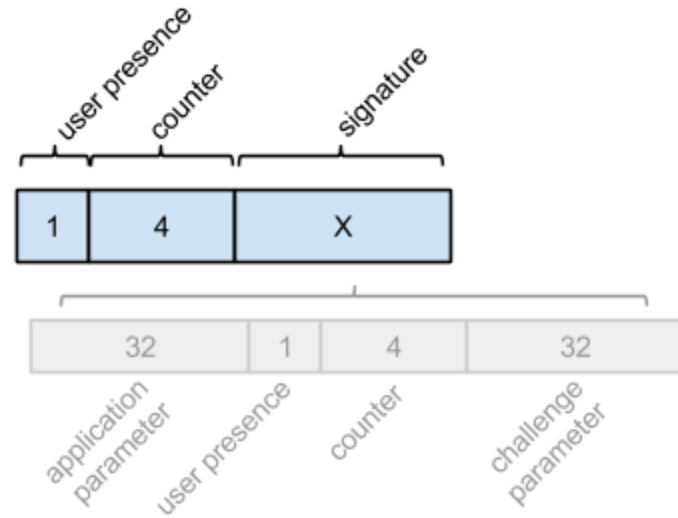


U2F Sign (Authenticate) JavaScript API

```
navigator.handleAuthenticationRequest({
    ‘challenge’: ‘KSDJsdASAS-AIS_AsS’,
    ‘app_id’: ‘https://www.acme.com/facets.json’,
        ‘key_handle’: ‘JkjhdsfkjSDFKJ_ld-sadsAJDKLSAD’
         }, callback);

callback = function(response) {
        sendToServer(
    response[‘clientData’],
    response[‘signatureData’]);
};
```

USB HID Authenticate





yubico

DEMO

So many keys...

- Authentication public/private key
  - Unique for every RP
  - Generated during U2F Registration
  - Public key sent to RP during Registration
  - Keyhandle can be used to derive private key
    - Unlimited number of RPs on small device
  - Hard coded to ECDSA using NIST P.256 curve

yubico

So many keys...

- Device-unique symmetric secret
  - Unwrap/derive per-RP ECDSA key from keyhandle
  - Unique random key for every device
  - Yubico derives private key using HMAC-SHA256

yubico

Yubico’s U2F KeyHandle

- Keyhandle is nonce+MAC instead of encrypted
- Device can derive ECDSA private key from nonce and symmetric device secret
- MAC detects invalid keyhandle or malicious RP

So many keys...

- ECDSA attestation key (unique per batch)
  - Linked with device attestation certificate
  - Signs U2F Registration blobs

yubico

U2F Attestation

- Proves what U2F device the user used
- X.509 Certificate with batch-unique key
- Why batch-unique and not device-unique?
  - Privacy: device-unique key permits conspiring RPs to link a physical key to particular user
  - Common batch size could be 10k-100k (could be 1 - breaking the privacy aspects)

Registration completed!

You have now completed registration and U2F device enrollment!

Use the login form below to test authentication using the enrolled U2F device.

Verified device

Security Key by Yubico

What if… I want to support U2F?

- Server/Browser: Call Javascript APIs
  - Send KeyHandle in HTML/JavaScript to browser
- Server: Implement registration flow
  - decide how to handle attestation certificates
  - verify registration response
  - store public key, key handle with user account
- Server: Implement login flow
  - check username/password, look up key handle
  - verify authentication response (origin, signature, counter, …)
- Relying Party: Check your account recovery flow

Yubico U2F Software

Our idea is to publish host and server libraries in common languages as FOSS code

- C: libu2f-host & libu2f-server
- Java: java-u2flib-server
- PHP: php-u2flib-server
- Python: python-u2flib-host & python-u2flib-server

yubico

Resources

Libraries, Plugins, Sample Code, Documentation — developers.yubico.com/U2F

U2F Protocol Specification — fidoalliance.org/specifications

Yubico U2F Demo Server - Test your U2F device here! — demo.yubico.com/u2f

THANK YOU!

yubico

Password-manager friendly (PMF): Semantic annotations to improve the effectiveness of password managers

Frank Stajano, Max Spencer, Graeme Jenkinson
{max.spencer, frank.stajano, graeme.jenkinson}@cl.cam.ac.uk

University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge, CB3 0FD, United Kingdom

Abstract. Subtle and sometimes baffling variations in the implementation of password-based authentication are widespread on the web. Despite being imperceptible to end users, such variations often require that password managers implement complex heuristics in order to act on the user's behalf. These heuristics are inherently brittle. As a result, password managers are unnecessarily complex and yet they still occasionally fail to work properly on some websites. In this paper we propose PMF, a simple set of semantic labels for password-related web forms. These semantic labels allow a software agent such as a password manager to extract meaning, such as which site the login form is for and what field in the form corresponds to the username. They also allow it to generate a strong password on the user's behalf. PMF reduces a password manager's dependency on complex heuristics, making its operation more effective and dependable and bringing usability and security advantages to users and website operators.

1 Introduction

We don't have to explain to this audience that, on the web, we are asked to remember way too many passwords. One reasonable way of coping with this burden is with a password manager—a piece of software that remembers passwords on the user's behalf and submits them automatically when required. All modern browsers such as Chrome, Firefox and Internet Explorer provide an integrated password manager. Because websites frequently have slight differences in the way they handle asking the user to type a password (or to define a new one), every password manager must implement complex heuristics in order to parse, auto-fill and submit the password-requesting web pages. Such code is inherently fragile and requires continuous maintenance as login web pages evolve and become fancier. As a result, some websites don't work seamlessly with password managers.

Password managers would be simpler and more dependable if websites adopted a set of semantic labels for HTML forms that allowed unambiguous registration and submission of passwords by programs acting on the user's behalf. In this

Pre-proceedings version. Do not circulate.
Final version to appear in *Proceedings of Passwords 2014*, Springer LNCS.

paper we offer two main contributions. First, we document the many ways in which websites ask for passwords and the many subtle ways in which the heuristics commonly employed by password managers can break, demonstrating how such code requires extensive maintenance to be reliable. Second, and most important, we propose PMF, a practical set of semantic labels that websites may immediately adopt. We also very briefly discuss incentives and benefits for the various parties involved.

2 Inconsistencies in password-based login on the web

Ignoring issues of style and presentation, password-based authentication on the web presents a fairly consistent interface to the user. To log in, users first find the login form, enter their username and password for that site into the appropriate boxes, and then press return or click the submit button. And, to a first approximation, the behaviour of the browser and the website is consistent across sites as well: the username and password entered into the form are sent to the server in an HTTPS `POST` request and a session cookie is returned. However, when we look in more detail, we notice a huge range of variations, some subtle and some baffling. Whilst these variations are imperceptible to the user, they present difficulties for a software agent parsing or automatically submitting the login form. This is because sub-tasks like entering the right username and password "into the appropriate boxes" are non-trivial and must rely on heuristics.

What are, then, the variations commonly exhibited by popular websites? A significant one is the modification of the static HTML login form by JavaScript. Some sites, such as Pinterest [1], use JavaScript to dynamically insert the login form into the page. Besides being annoying for users who browse with JavaScript disabled for security reasons, this practice also complicates the task of a password manager. Instead of parsing the HTML document just once at load-time to find any login forms, it must also monitor all changes made to the Document Object Model (DOM) by JavaScript thereafter.

The sins of JavaScript don't end there, though. Many sites use JavaScript to actually submit the form, thereby confusing utilities such as password managers that commonly intercept submission of the form to save the credentials. Forgoing a simple "submit" input adds little benefit and obfuscates the login process. Furthermore, to mitigate Cross-Site Request Forgery (CSRF) attacks [2], JavaScript is sometimes used to automatically insert values (nonces/challenges associated with the session) into hidden fields within the login form. For example, the following is added to the form on the Vimeo login page (`https://vimeo.com/log_in`) using JavaScript:

```
<input type="hidden" name="token" value="8113..." />
```

Attempts to log in to Vimeo with JavaScript disabled fail. Programs that parse or submit a login form must be compatible with such approaches, without being explicitly aware of what they achieve or even that they are being used.

Another problem facing a software agent such as a password manager is extracting the meaning (semantics) from the HTML login form. In the first instance, we'd like to determine what site the login form is for. The continued prevalence of phishing attacks demonstrates that reliably determining the website of a login page is too difficult for many humans. Software agents should have an advantage here. Indexing the username and password by the site's URL ensures that, provided HTTPS is used, the username and password are only submitted to correct site. Unfortunately, things are rarely this simple. Some websites have login forms on multiple pages—for example Facebook has one on its main landing page (`www.facebook.com`) and one on a dedicated login page (`www.facebook.com/login.php`). Should these login forms be considered as being for the same service? In the case of Facebook, both URLs are in the same second level domain `facebook.com`, so the answer is probably yes. But what about in a corporate intranet, where diverse services such as for submitting expenses and time sheets are all likely to be under the same second level domain?

It might be argued that services should only be considered the same if they have exactly the same URL. But what about the query string? Does that have to match as well? What about the order of the query parameters? What about dynamic URLs that provide alternative but equivalent encodings of the URL's query component? Any heuristic trying to shed light on this morass is likely to get things wrong (at least some of the time). Should users really have to accept that the computer doesn't even know what service is being logged in to?

Whilst it's obvious to a human whether a login to Facebook or Gmail succeeded, it's actually pretty hard for a software agent to know what happened. Whether or not the submitted username and password were correct, a HTTP 200 OK response is returned, indicating that a page was successfully served in response to the request (possibly after a sequence of redirects—in the case of Gmail rather a lot of redirects). Password managers can not reliably differentiate between these outcomes and, as a result, they often ask users to save passwords that are mistyped or just plain wrong. This seems needlessly annoying.

3 Incentives

Our proposal offers obvious advantages to users in terms of usability (you don't have to remember or type the passwords any more) and security[1] (you can use strong, distinct passwords). The advantages for password manager writers are even clearer (without guesswork, code becomes simpler, more reliable and much easier to maintain). Let's thus spend a few words on the incentives for website operators.

We believe it is in the best interests of website operators to support password managers: the website users will gain in usability and security. If users, thanks to

[1] Users of password managers are still exposed to malware; we are not claiming that the security offered by password managers is absolute (see section 5). Besides, our proposal implicitly also supports higher-security password managers running on dedicated hardware.

password managers, adopted strong unique random passwords, website operators would have much less to worry about confidentiality compromises of their hashed password file.

We understand that website operators don't want to allow bots to register thousands of accounts and we support this goal. Any techniques the websites may wish to use to ensure the presence of a human registrant (from CAPTCHAs to telephone callbacks and so forth) will continue to be available. We are only concerned with helping the human registrant store the password in a password manager instead of having to remember it in their brain. Only websites with delusions of grandeur may still believe that, regardless of all other demands on the user's memory and patience, *their* password is so important that it must be uncrackably strong *and* different from any others *and* never written down. They should study the Compliance Budget model [3], manage risks more maturely and cure their superiority complex.

4 The PMF semantic markup

4.1 Overview

We propose adding "password-manager friendly" (PMF) semantic markup to forms related to creating, accessing and managing user accounts, to simplify the following tasks:

- Finding forms and determining their purpose (login, registration, etc.).
- Finding the important inputs within the forms.
- Parsing password policies and generating valid new passwords.
- Detecting errors.

We adopt a simple and pragmatic approach used in other HTML microformats, of using semantic class names. A `class` attribute value can be specified for any HTML element [4] and the use of semantic class names is supported by the W3C [5]. We use the `pmf` prefix as a poor man's namespace to avoid clashes with programmer-defined class names. For example[2], a login form is marked with the `pmf-login` class:

```
<form action="/login" method="POST" class="pmf-login">
```

Although form inputs have other attributes such as `name` and `type` which may *often* give sufficient semantic information, standardised class values can be used to remove *any* ambiguity. For example, not all inputs with `type="password"` are for long-term passwords: some are for one-time codes generated by hardware tokens. Furthermore, as `name` attribute values are sometimes automatically generated by web frameworks or are specified by other standards such as OAuth [6], use of these attributes could cause conflicts. In contrast, any HTML element may have multiple classes [4], so our use of semantic class names ensures interoperability.

[2] In these examples, underlined text denotes PMF-related additions.

4.2 Forms

Being able to reliably determine the type or purpose of a given form enables a software agent like a password manager to offer a richer and/or more consistent user experience. `form` elements should be marked with the semantic classes specified in Table 1.

Table 1. Semantic classes for forms.

Form type	Semantic class name
Login	`pmf-login`
Registration	`pmf-registration`
Change password	`pmf-change-password`
Password reset	`pmf-reset-password`

4.3 Inputs

Username Login and registration forms typically contain an `input` element of type `text` or `email` for entering a username (which is often the user's email address). These inputs should be marked with the `pmf-username` class:

```
Username or email address:
<input type="text" name="user" class="pmf-username"/>
```

Password resets and changes are tricky for a password manager because the software cannot tell—in the case where a user may have multiple accounts with the same website—which password is being changed. For example, a simple experiment using Firefox's built-in password manager and two Google accounts reveals that, in some cases, the password manager must prompt the user to ask which account they are updating the password for, even though they are already logged in.

We propose that site authors should include a `hidden`-type field in these forms, marked with the `pmf-username` semantic class and with its value set to the username of the relevant account:

```
<form action="/reset" method="POST" class="pmf-reset-password">
  <input type="hidden" class="pmf-username" value="jimbojones"/>
  ...
</form>
```

Passwords Inputs for passwords typically appear in all four of the above form types. Some password inputs, such as those in registration forms, are for new passwords, while others are for existing passwords. These sub-types are unambiguously distinguished by the `pmf-new-password` and `pmf-password` semantic classes respectively. It is useful to distinguish them because they appear together

in “change password” forms. These typically contains three `password`-type inputs, one for the user’s current password and two for their desired new password (one to confirm the other). All three will have a different `name` attribute values but, using semantic class names, the purpose of each input is made clear:

```
<form action="/change" method="POST"
    class="pmf-change-password">
  <input type="password" name="current" class="pmf-password"/>
  <input type="password" name="new" class="pmf-new-password"/>
  <input type="password" name="confirm"
     class="pmf-new-password"/>
</form>
```

Stay signed in Many login forms include a “stay signed in” check box which allows the user to control whether their session with a website should persist across multiple browser sessions. If present, this input should be marked with the `pmf-stay-signed-in` class:

```
Stay signed in?
<input type="checkbox" name="persist"
    class="pmf-stay-signed-in"/>
```

Annotating the “stay signed in” check box allows a software agent to apply a global policy on staying signed in for the user, across all websites. Many websites tick the “stay signed in” box by default and users accept this. But, if their password manager could apply a “never stay signed in” policy for them, they may be happy for it to do so and thereby gain a valuable security (and privacy) boost by not being permanently signed-in to their online accounts.

Another scenario in which this feature might be useful is the cybercafé: for the benefit of the patrons, the web browsers installed on the public cybercafé machines might be configured to disable the “stay signed in” feature by default.

Hidden inputs Forms often contain `hidden`-type input elements which are not visible when the HTML is rendered[3]. As human users are normally unaware of and cannot interact with these inputs, it is not useful for a software agent acting on the user’s behalf to be able to interact with them either and we don’t propose any additional markup.

4.4 Password policies

Large-scale password leaks have shown that many users optimise for memorability and convenience rather than security, choosing trivially-guessable passwords

[3] The values of these hidden inputs are usually populated by the web server when it generates the HTML of the page and then not changed on the client side. For example web frameworks, such as Django [7], use them to implement Cross Site Request Forgery protection.

like `123456`, `qwerty` or `password`. Password composition policies ("between 8 and 16 characters, of which at least one uppercase, one digit and one symbol") are an attempt to enforce selection of passwords that will be harder to guess. Although we may not agree with the password policies that websites impose,[4] we believe that their rules should be made available in machine-readable form to allow password managers to generate strong compliant passwords.

In this section we therefore define a simple specification for a machine-readable (JSON) description of a password composition policy. Our goals have been to make it easy for the website developer to write their intended policy and for the spec to be sufficiently expressive that most commonly observed policies can be represented with it.[5]

A machine-readable password composition policy is included in an HTML document as the value of a `hidden`-type input element[6]. This hidden input should be marked with the `pmf-policy` semantic class and appear within a form with the `pmf-register` semantic class. A library routine, written once and for all as part of the standard, can then generate the corresponding human-readable version and localize it to any language[7]:

```
<form action="/register" method="POST" class="pmf-register">
  ...
  New password:
  <input type="password" name="new" class="pmf-new-password"/>
  <input type="hidden" class="pmf-policy" value='
      Machine-readable policy as a multiline string
      according to the syntax described below.
  '/>
  Human-readable policy
  ...
```

[4] The debate on whether they are effective would sidetrack us into a different discussion; what we note here is that such policies may reject some otherwise very strong passwords such as those that a software agent might generate. For example because they exceed the maximum length, or because they fail to include a character from one of the classes, or because they include a disallowed character, maybe outside the ASCII range. Bonneau and Xu's study [8] of non-ASCII passwords, or more accurately of passwords from people whose native language doesn't fit into ASCII, is instructive.

[5] We have strongly resisted the temptation to cover absolutely all cases and become Turing-complete. Useful rules that our notation cannot express include blacklists of known weak passwords and stateful checks such as "you can't use your username or a variation of it". But such violations are very unlikely to occur if the password is generated randomly.

[6] Note that, to allow double quotes in the policy as required by the JSON syntax, the whole policy must be enclosed in single quotes. The policy itself won't normally contain single quotes but, should it need to, those must be escaped.

[7] The routine could even be embedded as JavaScript in the page itself. Alternatively, the translation to human-readable form could be performed offline and the result statically embedded in the page, as in this example.

```
</form>
```

The policy optionally specifies a minimum and a (bleah)[8] maximum length as non-negative integer fields. It then specifies what classes of characters are allowed and which classes the password must contain.

In the spirit of the 2014 Stanford password policy [9], we allow the rules to be more strict for short passwords but more relaxed for long ones: this is achieved in practice by defining multiple sub-policies that apply depending on the length of the password.

Character class notation Several common character classes are predefined in Table 2. Although many current password systems only work with ASCII characters, for future-proofing we allow the definition of password policies that allow arbitrary subsets of Unicode characters. Arbitrary character classes can thus be defined by enumerating the Unicode characters they contain, in any order, in a JSON list. For example, the `symbol` class could also be written as

```
["!", "@", "#", "$", "%", "&", "*", "-", "+", "/", "="]
```

or, using Unicode escape sequences,[9] as

```
[
  "\u0021", "\u0040", "\u0023", "\u0024", "\u0025",
  "\u0026", "\u002a", "\u002d", "\u002b", "\u002f",
  "\u003d"
]
```

As a shorthand, contiguous ranges of Unicode characters in the list can be specified by listing the first and last character with the string `"..."` between them. For example, the `digit` class could be written as

```
["\u0030", "...", "\u0039"]
```

Table 2. Predefined character classes

String constant	Class contents
`"lower"`	the 26 lowercase ASCII letters
`"upper"`	the 26 uppercase ASCII letters
`"digit"`	the 10 ASCII digits
`"symbol"`	the following 11 symbols[10]: `! @ # $ % & * - + / =`
`"base64"`	the 64 ASCII characters defined by the “base 64” encoding
`"ascii"`	the 95 printable ASCII characters from 32 (space) to 126 (tilde)

[8] We believe setting a maximum password length is a dumb idea but here we are trying to allow websites to express their policy in machine-readable format rather than compelling them to switch to a sensible policy.

[9] `"\u"` followed by 4 hexadecimal digits (`http://json.org/`)

Simple policies Most of the policies observed in the wild are "simple" policies that apply the same character class requirements to passwords of all lengths. Simple policies are represented by a list of length one, containing a single sub-policy object:

```
[
  {
    minLen: 8,
    mustHave: ["upper", "lower", "digit"],
    mayHave: ["base64", "␣"],
  }
]
```

In the above example, the password must have a length of at least 8 characters inclusive; it must contain at least one character from each of the three classes listed on the `mustHave` line[11]; and it may also contain any character in any of the two classes listed on the `mayHave` line.

For example the `amazon.com` policy, at the time of writing, simply requires the password to be between 6 and 128 characters and can therefore be rendered as follows.[12]

```
[
  {
    minLen: 6,
    maxLen: 128,
    mustHave: [],
    mayHave: ["ascii"]
  }
]
```

Complex policies A complex policy is one where, as in the Stanford policy [9], the password composition rules depend on the length of the password. It consists of a list of two or more sub-policies with non-overlapping length ranges. If a certain password length is not included in any of the ranges, then passwords of that length are not allowed. For example, Fig. 1 is a rendering of the Stanford policy as written up in their poster.

4.5 Errors

As mentioned previously, determining whether a login attempt (or other action) was successful or not is a difficult problem for software agents, because at the HTTP layer a "200 OK" status code is returned in both cases. The user is informed of any problems using prominent human-readable error messages within

[11] Our notation cannot express more elaborate rules such as "must include characters from at least 3 of these 5 classes".

[12] The `mayHave` line is just guesswork: we have not tested whether Amazon allows even more characters—perhaps even non-ascii ones.

```
[
  {
    minLen: 8,
    maxLen: 11,
    mustHave: ["upper", "lower", "digit", "\u0020",
        "...", "\u002f"],
    mayHave: "ascii";
  },
  {
    minLen: 12,
    maxLen: 15,
    mustHave: ["upper", "lower", "digit"],
    mayHave: "ascii";
  },
  {
    minLen: 16,
    maxLen: 19,
    mustHave: ["upper", "lower"],
    mayHave: "ascii";
  },
  {
    minLen: 20,
    mustHave: [],
    mayHave: "ascii";
  },
]
```

Fig. 1. Stanford password policy expressed in the PMF policy language.

the returned HTML page, but we would like these messages to be just as easy to find for machines.

We propose marking these error messages with the `pmf-error` semantic class name to make them trivial for software agents to find:

```
<p class="pmf-error">Incorrect username and/or password</p>
```

5 Related work

Bonneau and Preibusch's [10] comprehensive review of the authentication landscape on the web argues that some sites deploying passwords do so primarily for psychological rather than security reasons. For example, they speculate that password-protecting accounts serves as a justification for collecting marketing data and as a way to build trusted relationships with customers. Whatever the underlying reasons, it is apparent that the number of password-protected accounts an average user manages has increased markedly since the advent of the web. Florencio and Herley [11] report that the average user has 6.5 passwords, each of which is shared across 3.9 different sites. Furthermore, each user has about 25 accounts that require passwords. Without the reported level of password reuse, managing 25 separate accounts with unique random passwords is barely imaginable for most users.

A password manager, either as a separate program such as PasswordSafe [12] or integrated with or in the browser, is now a well established solution for managing the increasing burden password-based authentication on the web. Given the increasing reliance on password managers[13], a recent thread of research has investigated their security properties.

Gasti and Rasmussen [13] investigate the security properties of the password database formats used in range of popular password managers. They define two new games to analyse the security of password manager databases: *indistinguishability of databases* (IND-CDBA) game and *chosen database* (MAL-CDBA) game; the *indistinguishability of databases* game models the capabilities of a realistic passive adversary, and the *chosen database* game models the capabilities of an active adversary able to both read and write the password database file. Google Chrome stores plaintext username/passwords in the user's profile directory. As a result, an attacker can trivially win both the IND-CDBA and MAL-CDBA games with Chrome as the Challenger. Firefox also fails both games; however, Firefox optionally allows users to encrypt the passwords stored in the password managers database under a user-supplied master key. This option provides at least some security benefits over Google Chrome's password manager, even if the full benefits of indistinguishability under the IND-CDBA and MAL-CDBA games aren't afforded. Gasti and Rasmussen's analysis concludes that, among the systems they studied, only PasswordSafe v3 [12] is invulnerable to attackers under the IND-CDBA and MAL-CDBA security models.

[13] As an example, 1Password alone is estimated to have a install base of 2 to 3 million users.

Silver *et al* [14] identify a class of vulnerabilities exploitable when using several popular passwords managers. The treat model they consider is a user connecting to a network controlled by the attacker, such as a rogue WiFi hot-spot. Under this model the attacker is able to inject, block and modify packets on the network. The attacker's goal is to extract passwords stored by the password manager without further action from the user. The attacks presented by Silver *et al* rely on exploiting the password manager auto-filling policies: for example, the password manager can be coerced into auto-filling forms in invisible iframes embedded within the WiFi hot-spot's landing page.[14]

Li *et al* [15] analysed the security of five popular integrated password managers (that is, password managers integrated with or in the web browser). Four key concerns with browser-based password managers were identified in this study: bookmarklet vulnerabilities, web vulnerabilities, authorisation vulnerabilities and user interface vulnerabilities. Bookmarklet[15] vulnerabilities, introduced by Adida *et al* [16], result from the bookmarklet's code running in a JavaScript environment potentially under the control of an attacker. Li *et al* show that such vulnerabilities are still widespread in popular password managers. The web vulnerabilities identified by Li *et al* consist of well know cross-site request forgery (CSRF) and cross-site scripting (XSS) attacks. The authorisation flaws identified by Li *et al* result from sloppy implementations. User interface vulnerabilities can be considered as *phishing* attacks against the password manager itself. In cases where the user is not authenticated to their password manager, a number of in-browser password managers automatically open the login form for the password manager in an iframe. Users have no means to differentiate between this behaviour and a phishing attack.

Password managers should be considered as tactical solutions, alleviating some of the gross security and usability failings of passwords. Pico [17] is a strategic solution seeking a more usable and secure replacement for passwords everywhere they are used (not just on the web). Recent work on Pico has attempted to provide a mechanism that can work alongside passwords [18]. The Pico bootstrapping technologies, whilst not being a password manager in the classic sense, are required to parse and automatically submit login forms on the user's behalf and would thus also benefit from our semantic annotations.

[14] Auto-filling of forms by the password manager improves usability and therefore, before mitigating this vulnerability by disabling the auto-filling, careful consideration is needed of the inherent trade off between security and usability. We shouldn't lose sight of the fact that normal users don't have threat models; therefore, simply asking them whether they want to enable or disable auto-filling is a bit of a cop out.

[15] A bookmarklet is a bookmark containing JavaScript that can be used to extend a web browser's capabilities. Bookmarklets have advantages over alternatives such as addons or extensions as they are cross browser and are managed by the user like bookmarks.

6 Conclusions

All password managers rely on fallible heuristics. Such code is complex, never fully accurate and it requires constant updates, besides wasteful replication of efforts by every password manager developer. We argue that all parties would benefit if websites offered a standard interface to password managers, enabling consistent and accurate agent-supported password creation, registration and login, without brittle programmatic guesswork.

Our PMF proposal, of augmenting a website's password pages with simple and unambiguous machine-readable semantics, makes the operation of password managers much simpler and more reliable. Users benefit from reduced cognitive load and reduced typing burden. Reliable generation of strong random passwords increases security for both users and websites. A well-defined interface eliminates guesswork and makes the password manager code leaner and much easier to maintain. We feel PMF is beneficial for all parties involved: users, website operators, password manager developers. We will be pleased to work with developers of websites, browsers and password managers, as well as with standards bodies, to promote its widespread adoption.

7 Acknowledgements

References

1. Pinterest. `https://pinterest.com` Accessed: 2014-11-07.
2. OWASP: Cross-site request forgery (csrf) prevention cheat sheet. `https:///www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet` (August 2014) Accessed: 2014-11-06.
3. Beautement, A., Sasse, M.A., Wonham, M.: The compliance budget: Managing security behaviour in organisations. In: Proceedings of the 2008 Workshop on New Security Paradigms. NSPW '08, New York, NY, USA, ACM (2008) 47–58
4. Berjon, R., Faulkner, S., Leithead, T., Pfeiffer, S., O'Connor, E., Doyle Navara, E.: HTML5. Candidate recommendation, W3C (October 2014)
5. Stuven, Sybrel (W3C): Use class with semantics in mind. `http://www.w3.org/QA/Tips/goodclassnames` Accessed: 2014-11-07.
6. Hardt, D.: The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard) (October 2012)
7. Django documentation: Cross Site Request Forgery protection. `https://docs.djangoproject.com/en/1.7/ref/contrib/csrf/` Accessed: 2014-11-07.
8. Bonneau, J., Xu, R.: Of contraseñas, sysmawt, and mìmǎ: Character encoding issues for web passwords. In: Web 2.0 Security & Privacy. (May 2012)
9. Stanford University. `https://itservices.stanford.edu/service/accounts/passwords/quickguide` Accessed: 2014-11-07.
10. Bonneau, J., Preibusch, S.: The password thicket: technical and market failures in human authentication on the web. In: WEIS 2010. (2010)
11. Florencio, D., Herley, C.: A large-scale study of web password habits. In: Proceedings of the 16th International Conference on World Wide Web. WWW '07, New York, NY, USA, ACM (2007) 657–666

12. Schneier, B.: Password safe. `https://www.schneier.com/passsafe.html` Accessed: 2014-11-06.
13. Gasti, P., Rasmussen, K.B.: On the security of password manager database formats. In: ESORICS. (2012) 770–787
14. Silver, D., Jana, S., Boneh, D., Chen, E., Jackson, C.: Password managers: Attacks and defenses. In: 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USENIX Association (August 2014) 449–464
15. Li, Z., He, W., Akhawe, D., Song, D.: The emperor's new password manager: Security analysis of web-based password managers. In: 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USENIX Association (August 2014) 465–479
16. Adida, B., Barth, A., Jackson, C.: Rootkits for javascript environments. In: Proceedings of the 3rd USENIX Conference on Offensive Technologies. WOOT'09, Berkeley, CA, USA, USENIX Association (2009) 4–4
17. Stajano, F.: Pico: no more passwords! In: Proceedings of the 19th international conference on Security Protocols. SP'11, Berlin, Heidelberg, Springer-Verlag (2011) 49–81
18. Stajano, F., Jenkinson, G., Payne, J., Spencer, M., Stafford-Fraser, Q., Warrington, C.: Bootstrapping adoption of the pico password replacement system. In Christianson, B., Malcolm, J.A., Matyás, V., Svenda, P., Stajano, F., Anderson, J., eds.: Security Protocols XXII - 22nd International Workshop Cambridge, UK, March 19-21, 2014 Revised Selected Papers. Volume 8809 of Lecture Notes in Computer Science., Springer (2014) 172–186

charPattern: Rethinking Android Lock Pattern to Adapt to Remote Authentication

Kemal Bicakci, Tashtanbek Satiev

Computer Engineering Department, TOBB University of Economics and Technology, Ankara, TURKEY

Abstract. Android Lock Pattern is popular as a screen lock method on mobile devices but it cannot be used directly over the Internet for user authentication. In our work, we carefully adapt Android Lock Pattern to satisfy the requirements of remote authentication and introduce a new pattern based method called *charPattern*. Our new method allows dual mode of input (typing a password and drawing a pattern) hence accommodate users who login alternately with a physical keyboard and a touchscreen device. It uses persuasive technology to create strong passwords which withstand attacks involving up to 10^6 guesses; an amount many experts believe sufficient against online attacks. We conduct a hybrid lab and web study to evaluate the usability of the new method and observe that logins with *charPassword* are significantly faster than the ones with text passwords on mobile devices.

1 Introduction

As being a viable alternative to traditional text based passwords, graphical passwords have gained significant attention in academic research in the last 15 years [1]. From practical point of view, maybe the most successful graphical password example is Android Lock Pattern (ALP) which comes preinstalled in most Android smartphones and is presumably the most widely deployed one. As its name implies, Android Lock Pattern (ALP) is mainly used to lock smartphones. Security and usability requirements for remote access (over the Internet) are very different than the ones present in local operation while locking/unlocking a phone. We identify two main differences as follows:

1. ALP provides a theoretical password space of 18 or 19 bits [1, 2]. Recent research estimates a partial guessing entropy of only 9.1 bits [2]. This may provide adequate level of security for its intended purposes especially with a policy enforcing maximum number of false trials. On the other hand, although there is not a consensus among security researchers for the minimum security requirements for web authentication, there is no doubt that ALP in its present form offers much less than required.
2. Even though touch screen devices are being widely deployed, use of a desktop or a laptop computer with an old-fashioned monitor is still common. Previous research suggested that an authentication scheme designed for touch screen

devices such as ALP is likely not accommodate users alternating between desktops and touch screen devices, well [3].

In our work, we propose a new knowledge-based authentication method called charPattern targeting web applications by a careful adaptation of ALP method addressing the aforementioned differences and thus challenges. We also conduct a hybrid lab and web study to compare the usability of charPattern with text passwords and gridWordX [4]; a recent multiword password proposal answering the research challenge arising from the evolution of Internet access devices [3]. The results of user study show that while there is no significant difference between login times of charPattern and text passwords on desktop/laptop machines, login times on mobile devices are significantly lower with our new method, charPattern.

The rest of the paper is organized as follows: Section 2 overviews the related work. In section 3, the proposed system is presented. The methodology of user study is discussed in section 4 followed by presenting its results in section 5. We discuss the results of user study in Section 6. Section 7 concludes the paper.

2 Related Work

Graphical password schemes could be grouped based on how they are memorized: recall-based, cued-recall and recognition-based schemes.

Pass-Go, inspired by an old Chinese game, is a recall-based scheme where passwords are drawn by using grid intersection points [5]. Another grid-based system is Gridsure which specifically uses a 5 x 5 grid [6] as an alternative one-time PIN system. The grid is populated with different random digits, thus a user who memorizes her pattern could enter a different PIN occupied by the pattern in each login. PassPattern system [7] is a similar one-time password scheme.

Graphical passwords on mobile devices based on the recognition of photographs in the context of mobile devices were investigated by Dunphy et al. [8]. Schaub et al. explore the design space of graphical passwords on smart phones by implementing five different graphical password schemes on one smartphone platform [9]. They perform usability experiments and analyze shoulder surfing success rates. They consider two levels of theoretical password strength (14-bits and 42-bits).

Android Lock Pattern(ALP) could be considered as a variation of the Pass-Go scheme by using nine points arranged in a 3x3 grid [1, 2]. By setting the minimum number of points that should be chosen as four, the number of possible patterns is 389.112 giving an approximate security of 19 bits. However, this is just a theoretical maximum value. Uelenbeck et al. shows that in practice only a partial guessing entropy of 9.1 bits is achieved which is around the same security level of 3-digits random PINs [2].

Given the popularity of ALP, it is of no surprise to see that the idea is ported to other platforms as well. For instance Eusing Maze Lock 3.1 is such a free product for Windows platforms [10].

Building **passwords from multiple words** is a long-standing idea promoted to increase memorability and security. Cheswick [11] (See also summary by Rik Farrow [12]), was the first who proposed user-chosen multi-word passwords for convenient entry on smartphones.

gridWordX, improved version of gridWord [3], is a hybrid multi-word password scheme which supports elements of text and graphical passwords [4]. With gridWordX, the user could choose from a grid of words to form a password without requiring character-by-character text entry. In Fig. 1(b)), the words are arranged in a 8 x 13 (8 rows, 13 columns) 2D grid. Besides the grid, the interface also includes three combo boxes with autocomplete property for each words of the password to allow dual mode of input (either by typing or touching on the grid). Here, three-word-length password provides around 20 bits of password security.

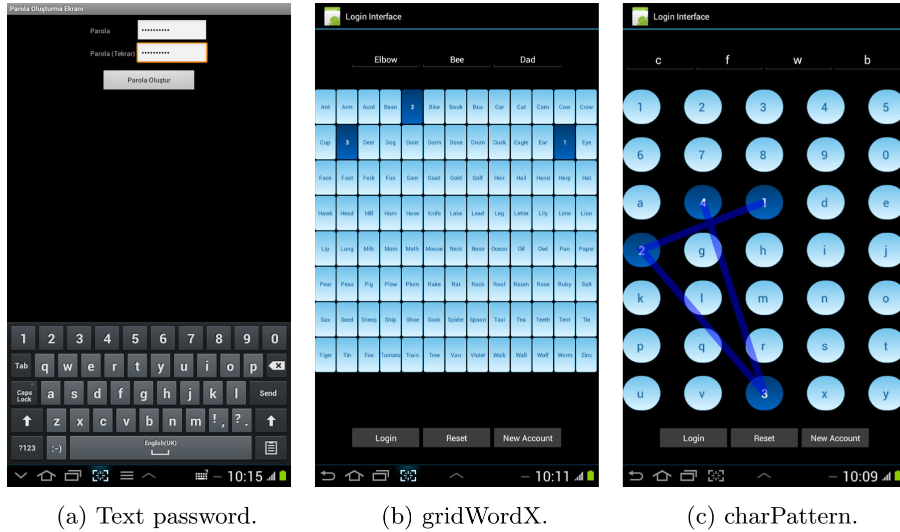
3 The Proposed System

The proposed system in this research, charPattern (see Fig. 1(c)), allows drawing a pattern over so called dot-characters to support entering a password by touching on the mobile device (dot-character is a dot corresponding to a unique character). Since patterns stimulate visual memory, charPattern is expected to leverage password memorability. Alternatively, the system also facilitates password entry by typing the dot-characters forming the pattern with a physical or a virtual keyboard.

3.1 Design Features

We identify the main differences between charPattern and Android Lock Pattern (ALP) as follows (see Table 1):

1. ALP has 9 dots organized in a 3 x 3 grid. On the other hand, charPattern has 35 dots organized in seven rows and five columns. With at least four dots forming a password, this gives a password space over one million. We note that if the passwords are chosen uniformly, a password space of one million could withstand against online attacks if lockout rules are in use [13, 14]
2. Theoretical password space could not be reached in practice with user-chosen passwords since users are more likely to select a password among hotspots, a more popular subset. However, with persuasive technology proposed first with Persuasive Cued Click Points (PCCP) method [1], hotspots could be avoided. The basic idea is to suggest users a randomly generated password while they are creating their account. While users are allowed to ask for a new suggestion as much as they wanted, this significantly slow the password creation process. Hence a secure password selection becomes a path of "least resistance". In a sense, use of persuasive technology could be regarded as balancing the tradeoff between system generated passwords and user chosen passwords regarding usability and security properties. In charPattern,



(a) Text password. (b) gridWordX. (c) charPattern.

Fig. 1: Login interfaces of authentication methods investigated in our work on a mobile device.

we borrow this technique to suggest users a randomly generated pattern password composed of four dot-characters.

3. In charPattern, each of the dot is mapped to a unique alphanumeric character. We choose 10 numeric digits and 25 lowercase letters (all letters in English alphabet except the letter "z") to have 35 characters in total. This gives the opportunity to map each pattern to a text password composed of 4 characters. Users are free to enter their passwords either by drawing the pattern or by typing the password. For instance the pattern seen in Fig. 1(c) could also be entered by typing the password "cfwb".
4. To be able to draw a pattern with any of four dots (not only the consecutive dots), we require pausing for minimum of 150 ms on a dot to select it as part of the pattern. In other words, unlike ALP method with charPattern it is possible to skip dots if we draw a pattern without pausing over them.

3.2 Implementation

The proposed system is implemented for both mobile devices and as a web application for desktop/laptop computers. On the mobile device, charPattern is implemented as a standalone full-screen Android application (see Fig. 1).

We also develop charPattern as a web application for desktop computers using PHP, HTML and Javascript version 5 (not shown as a figure). Both the mobile and the web application are developed by the same programmer to achieve a comparable look and feel.

Table 1: ALP vs. charPattern

Comparison Criteria	ALP	charPattern
Number of dots	9	35
Dot-matrix size	3x3	5x7
Dot interface	only dots	each dot mapped to a unique character
Password-length	[4,9] dots	4 dots
# of possible passwords	389112	1256640
Max. password entropy (bits)	17	20
Compatibility with entry using physical keyboards	NO	YES
Creating a password	user-selected	use persuasive technology
Dot selection method	every dot in a path	150 ms pausing on a dot to select

4 User Study

We conduct a user study to compare the usability of traditional text passwords, gridWordX and charPattern on mobile devices and in a traditional desktop/laptop environment. Before the study, we formed our hypotheses as follows:

1. Login with charPattern takes shorter time than with text-based authentication on mobile devices.
2. Login with charPattern takes shorter time than with gridWordX on mobile devices.
3. Login with charPattern takes comparable time with login using text passwords on computers having physical keyboard.
4. Login with charPattern takes comparable time with login using gridWordX on computers having physical keyboard.

In the user study, 25 undergraduate and graduate students of TOBB University of Economics and Technology (17 males and 8 females) participated. The ages of participants are ranged between 19 and 28. We note that every participant is already familiar with using desktop computers and mobile devices for Internet access.

4.1 Sessions of the Study

The user study has a within-subjects design and consists of four sessions. The interval between each session is minimum of four days and maximum seven days. In the first session, each participant is invited to the lab and asked to create an account by entering a username and creating a password on a mobile device. A password is created for all three systems; text password authentication, gridWordX and charPattern hence each participant has three passwords in total. The participant also performs a login on the mobile device after solving a mental

rotation test (MRT) test. MRT is used to remove users' short term memory. We employ counterbalancing between password methods to handle order effects.

In the second and third sessions, the participants perform logins on their own laptop/desktop computers remotely by their username-password pairs created in the first session (with all three systems).

In the fourth (last session) session, the participants are re-invited to the lab and asked to perform a second login on the mobile device with their username-passwords (again with all three systems).

4.2 Pre-experimental Instructions

Before the first session, a brief presentation about the user study was provided which includes general oral instruction and a short demo on three password methods. The oral instruction covers the following points:

- We emphasize that our aim is to evaluate the authentication methods, not the participants themselves.
- We ask participants to create a text password which consists of at least eight characters.
- We ask them not to use a password they use in real life as the text password they create for the study.
- The participants should not take a note of their passwords in any form (writing down, taking a photo, etc.).
- The participants are asked to treat their passwords as a real passwords rather than just experimental as they have to use them in future sessions, again.

We do not mention which authentication method is designed by us in order to avoid any bias among the participants with respect to usability of the methods.

4.3 Lab Study

In the lab study, all participants used the same mobile device (Sumsung Tab2 7 inch tablet with Android SDK API 17 which has 600x1024 resolution and 170 ppi pixel density) so that they are tested under same conditions. The participants filled out a post-task questionnaire after the second login performed in their second visit to the lab.

4.4 Web Study

Second and third sessions were conducted over the Internet hence we call it as a web study. The web study was held to compare usability of charPattern with traditional text password and gridWordX on desktop/laptop computers. We asked participants not to use their touch-screen devices in the web study. But we did not ask anything particular regarding mouse use. The users were free to use a keyboard or a mouse (applicable only with gridWordX and charPattern) to enter their passwords. In the web study, users were allowed to ask for their passwords through email if they decided they could not recall their passwords.

5 Results

The following data is collected in the user study:

- **Timing**. Creation & confirmation and login times.
- **Number of Attempts**. The number of attempts until the correct login.
- **Number of Shuffles.** How many times a user asks for a new password suggestion (applicable to gridWordX and charPattern).
- **Modes of Input.** In gridWordX and charPattern, users enter passwords either by typing or by drawing (touching). Mixing these two modes is also possible. As a result, there are three different modes of input.
- **Questionnaire.** User responses to survey questions.

5.1 Collected Data Analysis

Here, we provide the results of the collected data analysis. While applying statistical tests, a difference is considered statistically significant if the p value is less than 0.05.

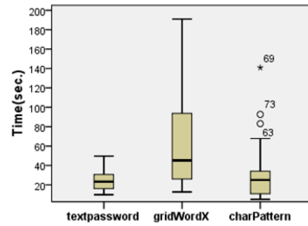


Fig. 2: Creation & Confirmation times.

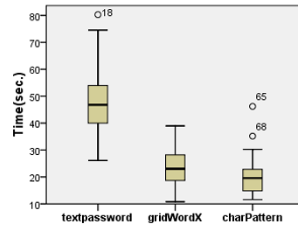


Fig. 3: Login times in lab study.

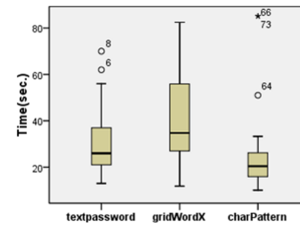


Fig. 4: Login times in web study.

The times to create and confirm passwords for each method are presented in Figure 2.

Login times in the lab study and in the web study are presented in Figure 3 and Figure 4, respectively.

Table 2: Friedman Test Results for Lab Study.

Method Name	Mean Ranks			Test Results	
	First Login	Last Login		First Login	Second Login
text password	2.93	2.84	Chi-Square	31.76	26.64
gridWordX	1.56	1.64	df	2	2
charPattern	1.52	1.52	Asymp.Sig.	0.00000	0.00000

Regarding login times of three methods on the mobile device (lab study), we obtain highly significant difference between three datasets by applying non-parametric k-related sample-test Friedman to three datasets in each of two sessions separately as shown in Table 2.

Table 3 presents results of non-parametric k-related sample-test Friedman applied to login times of text passwords, gridWordX and charPattern in the Web study. Here, we find no significant difference although charPattern has shorter login times than text passwords and gridWordX.

Table 3: Friedman Test Results for Web Study.

Method Name	Mean Ranks			Test Results	
	First Login	Last Login		First Login	Second Login
text password	1.92	2.16	Chi-Square	4.16	2.96
gridWordX	2.32	2.12	df	2	2
charPattern	1.76	1.72	Asymp.Sig.	0.125	0.228

Table 4 presents success rates of text passwords, gridWordX and charPattern with regard to creation & confirmation and login (a user is considered successful if he/she could complete it with no more than three attempts and if the password is not asked by email). We apply non-parametric k-related sample-test Friedman and obtain no significant difference between results.

Table 4: Login Success Rates

	Create & Confirm	Login Sessions			
		First	Second	Third	Fourth
text password	25/25	25/25	24/25	25/25	25/25
Success Rates	100.00 %	100%	96%	100%	100%
gridWordX	23/25	25/25	17/25	23/25	25/25
Success Rates	92%	100%	68%	92%	100%
charPattern	25/25	24/25	16/25	24/25	25/25
Success Rates	100%	96%	64%	96%	100%

As presented in Table 5, shuffle count of charPattern is less than of gridWordX, but by applying the paired-sample Wilcoxon test, we obtain no significant difference between them.

The number of participants using more than 5 shuffles with gridWordX is 5, whereas with charPattern it equals to 1. In Table 6, we show how number of shuffles in gridWordX and charPattern influences success rates.

Input modes are typing, drawing (clicking) and hybrid mode in charPattern and gridWordX. The distribution of participants regarding these three input modes is shown in table 7.

Table 5: Shuffle Results of gridWordX and charPattern

	N	Mean	Std. Dev	Min	Max
gridWordX	25	4.60	7.984	0	36
charPattern	25	1.56	1.981	0	7

Table 6: Effects of Shuffles on Success Rates for gridWordX and charPattern

	# of Shuffles	# of Trials	Confirm and Login Success Rates				
			Conf.	1st	2nd	3rd	4th
gridWordX	Low:<6	20 (80%)	95%	100%	70%	90%	100%
	High:>5	5 (20%)	80%	100%	60%	100%	100%
charPattern	Low:<6	24 (96%)	100%	95.8%	62.5%	96.8%	100%
	High:>5	1 (4%)	100%	100%	100%	100%	100%

In the questionnaire, we ask seven 10-point Likert-scale (1 is disagreement, 10 is strong agreement) questions. The results are given in Table 8.

6 Discussion

Before the user study, we conjectured that users would spend less time to login with charPattern on a mobile device because drawing a pattern is much natural than typing on a virtual keyboard (as in text passwords) or touching on cells in a grid (as in gridWordX). As seen in Table 2, charPattern is faster than text passwords and gridWordX with respect to login times on the mobile device which supports our first two hypothesis.

Regarding login times of text passwords, gridWordX and charPattern in the Web study, we find no significant difference (see Table 3). As a result, hypothesis 3 and 4 are also supported. Before the user study, we conjectured that on a machine without a touchscreen the advantage of charPattern regarding login times is lost because drawing the pattern on the screen is no longer possible. But we thought charPattern still yields comparable login times with the other methods since users have the chance to try other modes of input *i.e.*, by typing. After the user study, we see that the expected result is observed due to a reason not we have foreseen. In the user study, users still prefer drawing the pattern over typing the password but this time with a mouse or a touchpad. Since drawing a pattern with a mouse or a touchpad is not as comfortable as drawing it on the screen, the login times turned out to be as expected; comparable to other two methods.

Figure 5 demonstrates the change in login times in subsequent logins. The login time in the second login on a mobile device takes longer that the one in the first login for all three methods. This results suggest that although we applied a MRT test, users were more comfortable in entering their passwords just after they created it. On the other hand, in the web study second logins took much less time than the first login. This result is as expected because the first login was the

Table 7: Frequency of Input Modes in charPattern and GridWordX

		Create & Confirm	Logins			
			wk 1	wk 2	wk 3	wk 4
gridWordX	clicking	25	24	23	23	25
	typing	0	0	1	0	0
	hybrid	0	1	1	2	0
charPattern	drawing	25	25	22	24	25
	typing	0	0	2	1	0
	hybrid	0	0	1	0	0

Table 8: The Questionnaire Results

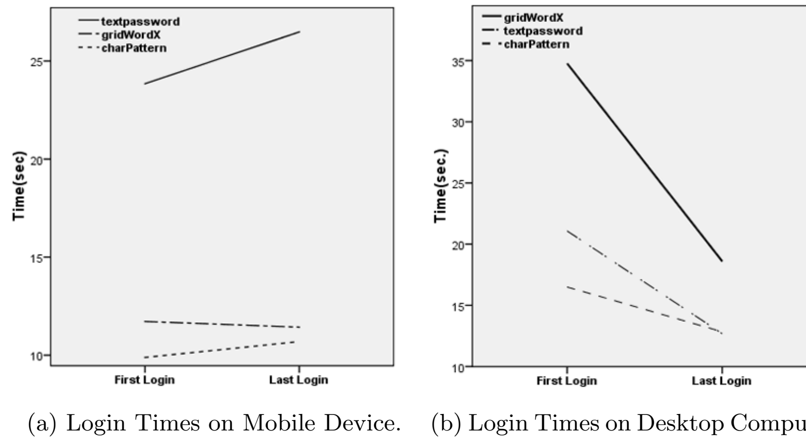
Question	Mean
1. Using pattern makes charPattern easily memorable	8.56
2. The increase in number of dots does not make drawing a pattern more difficult	6.76
3. I easily created a password in charPattern	8.68
4. Login using charPattern was easy on a desktop computer	9.48
5. Login using charPattern was easy on a mobile device	9.08
6. I liked charPattern as much as a text password	8.04
7. charPattern is at least as secure as a text password	7.72

first time the web interface was presented to the users. The important point here is that in the lab study the difference between login times of charPattern and text passwords holds for both logins (on the other hand, the difference between gridWordX and charPattern drops significantly).

The survey results show that users find charPattern easy-to-use both on desktops and mobile devices. It is surprising to see that users find charPattern easier to use than text passwords more on desktop machines than mobile devices (although the difference is not significant).

Limitations. One obvious limitation is with regard to demographics and number of the participants. The participants were all university students which might not reflect the behavior of general public. Secondly, the number of participants was limited and not sufficient to make sharp conclusions. Finally, we conducted the study within a short time period. Studies in longer time frames would be better for analyzing memorability of charPattern.

Security Analysis. We mentioned that the password entropy of charPattern is 20 bits which can safeguard against online attacks with lockout rules. On the other hand, it is still of an issue whether some passwords are more likely to be chosen in this space. An attacker could exploit the nonuniform password distribution by giving priority to more likely passwords while guessing. In charPattern, we mitigate guessing attacks by disallowing user-chosen passwords and suggesting users randomly generated passwords. Hotspots could still be present in charPattern passwords if users ask for suggestions (hit the “Shuffle” button) until an easy-to-guess password is suggested.



(a) Login Times on Mobile Device. (b) Login Times on Desktop Computer.

Fig. 5: The change in login times in the first and second logins on Web and Lab Studies.

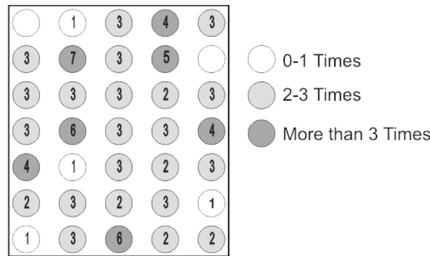


Fig. 6: Frequency of dots selected as part of a charPattern password.

Figure 6 presents the frequency of dots selected by the participants where 17.14% were selected 0-1 times, 62.86% were 2-3 times and 20% of dots were selected more than 3 times. To understand whether this particular distribution is different than a random distribution, we generate simulated data consisting of 100 datasets each of which has 25 pairs of (x, y) elements where x ranges from 1 to 5 and y ranges from 1 to 7 corresponding to the size of data in our user study charPattern. Then, we calculate rough estimate values of password entropy for the collected dataset together with random datasets using the formula H(X) defined in [15]. Our rough estimate password entropy of collected dataset is between maximum and minimum entropy values of simulated datasets. Since each random dataset represents a chance to include the observed data, with 99% probability, the user study dataset is a dataset occurred by chance. This analysis gives an evidence that hotspots does not skew the password distribution for charPattern.

7 Conclusion

As Android Lock Pattern has successfully demonstrated, drawing a password pattern is preferred over typing a password or a PIN by many users for locking/unlocking their touchscreen devices. However, lock patterns could not be used over the Internet directly for remote user authentication due to different security and usability requirements. In this paper, we introduce charPattern, a new pattern-based authentication method which increases password space to adequate levels by (i) increasing number of possible patterns by careful addition of more dots (ii) by using persuasive technology to avoid hotspot passwords (more popular patterns). To accommodating users who alternately login from devices with, and without,full physical keyboards, the new scheme improves on the idea of Android Lock Pattern by introducing a second mode of input by enabling users to type the characters corresponding the dots forming their pattern password.

Our user study, which involves lab and web sessions, shows that charPattern has significantly shorter login times than text passwords on a mobile device. In addition, most users choose to enter charPattern passwords by drawing the pattern rather than by typing via keyboard even on desktop machines, which leads to login times comparable to those of text passwords on desktops. Based on user study findings, we conclude that charPattern is a promising alternative to text passwords for those who access same sites from mobile devices and desktops. In the future, we plan to compare recall of charPattern passwords with recall of text passwords in a long term user study.

Bibliography

[1] Robert Biddle, Sonia Chiasson, and P.C. Van Oorschot. Graphical passwords: Learning from the first twelve years. *ACM Comput. Surv.*, 44(4):19:1–19:41, September 2012.

[2] Sebastian Uellenbeck, Markus Dürmuth, Christopher Wolf, and Thorsten Holz. Quantifying the security of graphical passwords: The case of android unlock patterns. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, CCS '13, pages 161–172, New York, NY, USA, 2013. ACM.

[3] Kemal Bicakci and Paul C. van Oorschot. A multi-word password proposal (gridword) and exploring questions about science in security research and usable security evaluation. In *Proceedings of the 2011 Workshop on New Security Paradigms Workshop*, NSPW '11, pages 25–36, New York, NY, USA, 2011. ACM.

[4] Ugur Cil and Kemal Bicakci. gridwordx: Design, implementation, and usability evaluation of an authentication scheme supporting both desktops and mobile devices. *Workshop on Mobile Security Technologies (MoST13)*, 2013.

[5] Hai Tao and Carlisle Adams. Pass-go: A proposal to improve the usability of graphical passwords, 2006.

[6] Sacha Brostoff, Philip Inglesant, and M. Angela Sasse. Evaluating the usability and security of a graphical one-time pin system. In *Proceedings of the 24th BCS Interaction Specialist Group Conference*, BCS '10, pages 88–97, Swinton, UK, UK, 2010. British Computer Society.

[7] T. Rakesh Kumar and S. V. Raghavan. Passpattern system (pps): A pattern-based user authentication scheme. In *Proceedings of the 7th International IFIP-TC6 Networking Conference on AdHoc and Sensor Networks, Wireless Networks, Next Generation Internet*, NETWORKING'08, pages 162–169, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Paul Dunphy, Andreas P. Heiner, and N. Asokan. A closer look at recognition-based graphical passwords on mobile devices. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, SOUPS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM.

[9] Florian Schaub, Marcel Walch, Bastian Könings, and Michael Weber. Exploring the design space of graphical passwords on smartphones. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, SOUPS '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[10] Eusing Maze Lock 3.1. www.bit.ly/maze203, 2014.

[11] Cheswick. Rethinking passwords. invited talk, usenix lisa 2010, 2010.

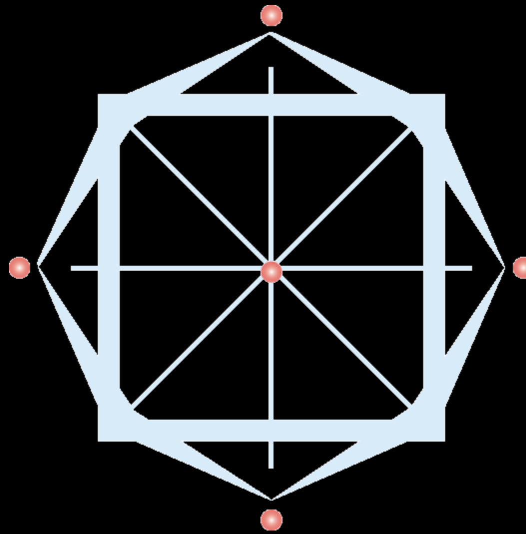
[12] Rik Farrow. Login: Usenix magazine, 2011.

[13] Dinei Florêncio, Cormac Herley, and Baris Coskun. Do strong web passwords accomplish anything? In *Proceedings of the 2Nd USENIX Workshop*

on Hot Topics in Security, HOTSEC'07, pages 10:1–10:6, Berkeley, CA, USA, 2007. USENIX Association.

[14] Dinei Florêncio, Cormac Herley, and Paul C. van Oorschot. An administrator's guide to internet password research. In *28th Large Installation System Administration Conference (LISA14)*, Seattle, WA, November 2014. USENIX Association.

[15] Kemal Bicakci, Nart Bedin Atalay, Mustafa Yuceel, and Paul C. van Oorschot. Exploration and field study of a browser-based password manager using icon-based passwords, 2011.

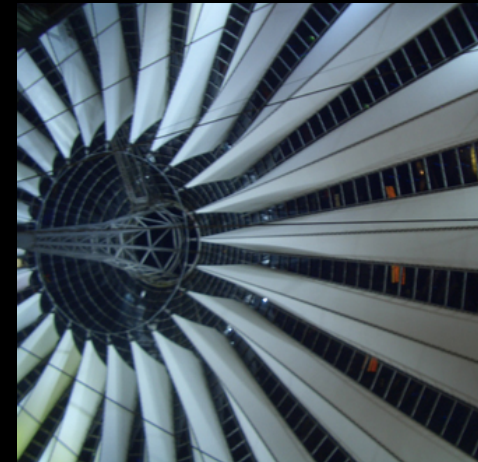


FABULAROSA

AND THE FIVE NEW PROTOCOLS

PASSWORD2014| 8-10 NOVEMBER 2014 | TRONDHEIM

Who are we?

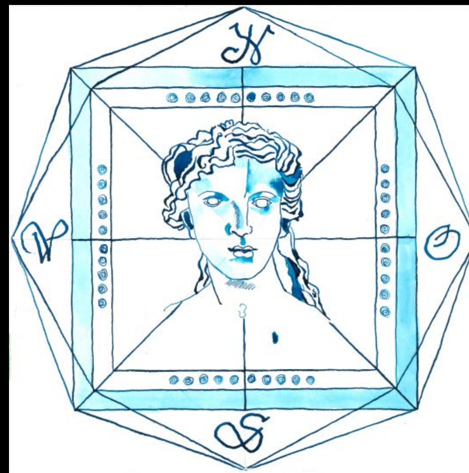


- The KikuSema GmbH
  - Headquarters Berlin/ Germany (1998)
  - Founder & CEO Ulf Ziske
    - Scientific Analyst (MSc) , Software developer.
- The Kikusema AB
  - Headquarters in Mariestad, Sweden (2001)
  - Founder & CEO Christine Ziske
    - MBA/Scientific Analyst (MSc) , IT-Consultant and project manager

SETTING THE SCENE

FABULAROSA

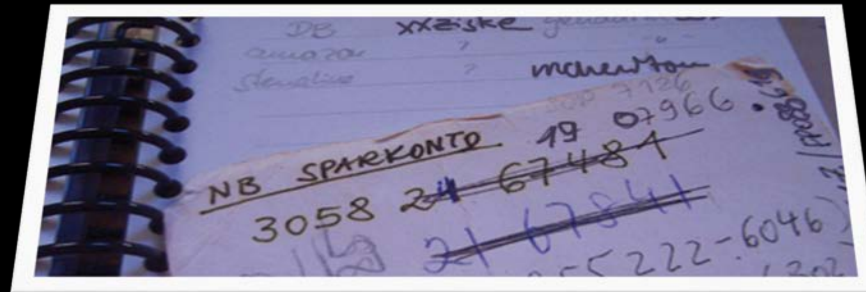
Thesis



The involvement of the human factor,
the cognitive performance and the capabilities of humans,
the Non-Technical Skills, needs to be improved within
authentication procedures.

FABULAROSA

Security Pains?



"I needed a password eight characters long so I picked Snow White and the Seven Dwarves." nick helm

FABULAROSA

What if?

TRUECRYPT EjF`SSPu`26jVfuGdj3frfjaHTaXif_5IpoAiS7MIxdRnX1zYiwS1XixrZTWwX_r

COMODO Creating Trust Online DYb}nMRkRFojgFgTaBOTZNo}Iu{rQeM14m

symantec. n0&00}_DTcAXI]1ACH00);:I^h{3+

Google Mail 0_5-5PNAqfiR04A%ek@,_^

.A]-;0-6r0344ggDr0)ijBAJ}+cw]

PayPal /{|h7LcJPb}000CU^c_p

MasterCard. SecureCode. SGBmPAntRW

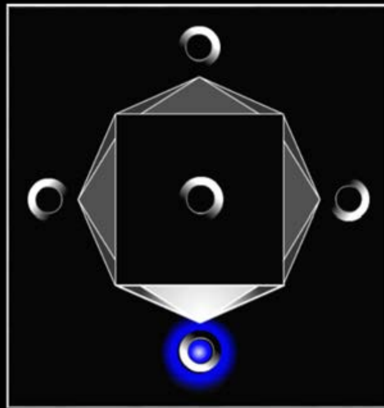
LinkedIn 0!<A0X0!byznrQi$PV7A%%0

SAS [0a{`AgJ00T{]

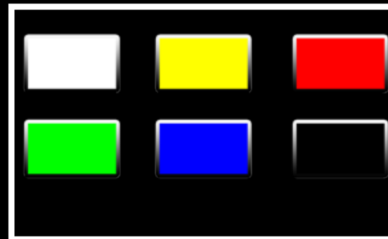
Adobe 5LVL;_:rngD:;

FABULAROSA

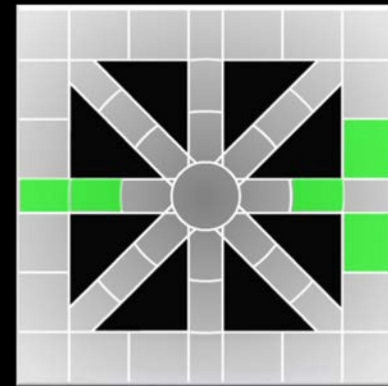
Combination amount?



4+1 directions



6 colours



1 pattern/ max 45 button

6 x 5 = 30 levels x 45 = 1350 possibilities to activate a button

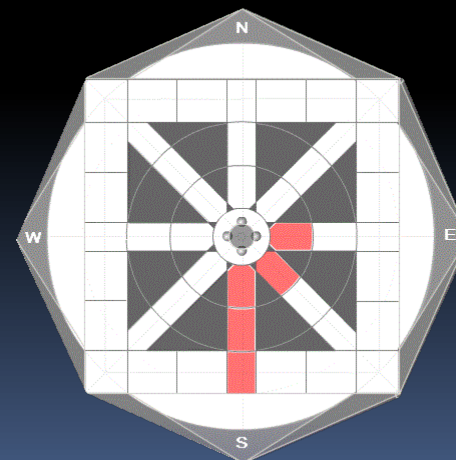
UTF16 = 65,535 CHARACTERS

FABULAROSA Combination amount?

- 3x3 Matrix Google

- 4of9 Pin iPad/iPhone

- FabulaRosa:
  - $65{,}535^{1350} = 1{,}7$ e+6502
    - $65535^{64} = 1{,}8$ e+308



FABULAROSA

What is it?

Android Certificate KikuSema
Comode codeSignature
Lufthansa Ulf Ziske
PassShipOne FabulaRosa
SAS 巴另工人德
TrueCrypt RSA2010

Please select an addressee

red

Please select a color

center

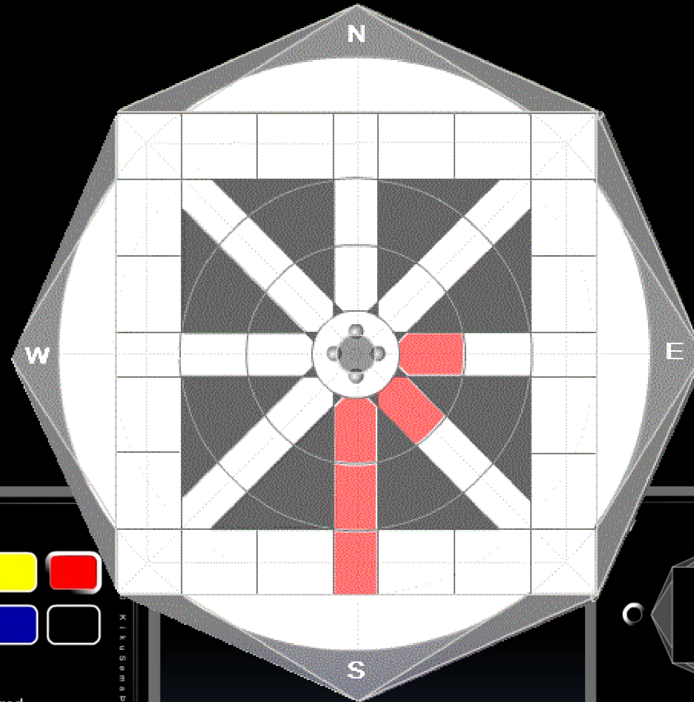
Please select a direction

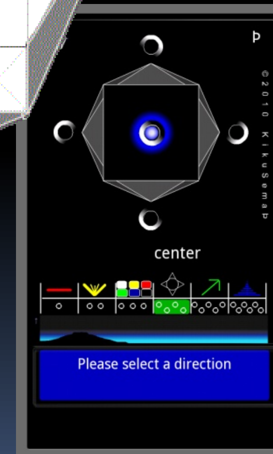
red center 5

Please draw your passsign

FABULAROSA

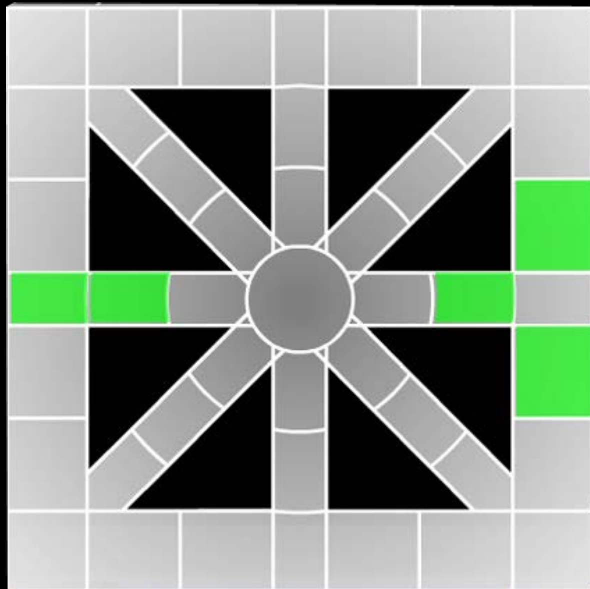
How does it work?





FABULAROSA

What do you get?



Ӣ◌Sð°ūѠ϶ƠU£5ä²mꝯh4Ó·¿ÆǺÚSmd
4Ä7ÛÎǺùPĪυ1§Í–³ÑбPǺßeıǺЧ◌Ǻ-
ÁǺ⎯wÙráǺǺǺx{I{÷Ǻɰ{gu◌ǺS죗
ωzjZ¥ì¬v¢寊Ǻ×
疰ĦȞúú°«Ìþ¢¶¿Ǻ}dǺǺº◌oôĤ榇
ë9Bɤ◌Ǻâa½ω×ǺǺÄÞ◌ÊǺǺĭaǺ1ǺÕǺ
4d_TzǺg◌ǺæǺ7ϱǺǺ◌ȷõǺE猂ÀǺR

EXTREMELY LONG & COMPLEX HIGH DEFINITION (HD) PASSWORDS

UTF16 = 65,535 CHARACTERS
COMBINED WITH 1350 POSSIBLE SELECTIONS

FABULAROSA What do you get?

_%::j#D>SbHHfLA}#rMZWh][AdPS'002TIXV0x0ht0e
<|0060v!3p0424@#0{|&

^){5]M7RAAG00q]@lrw,c

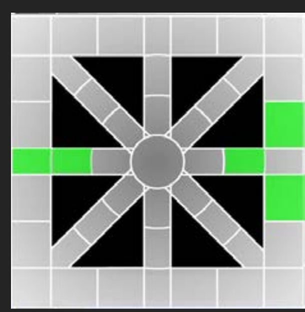
斎刲昪讟淂鯤奢乖三京淨刡丨蛳亩�injected

fhn1mkYXncfl9QAAZApgs9akxW9vAp5AJ2LoMqA
h1wAjMOAjVbS7yAuvW4RwEqsc

FABULAROSA Innovation?

- Change of the user interface
  - No use of the keyboards any longer
- Many different HD-passwords are created
  - by drawing only one image on a virtual 'Compass Rose' and applying such universal concepts as colours, directions and patterns,
  - By applying the unique algorithms
- Passwords of **850** characters from a **65,535** character set
  - Instead of passwords with **8** characters from a **94** character set

FABULAROSA

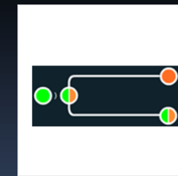


State?

Desktop

Smartphone/ Tablet

Server



THE BACKGROUND

FABULAROSA Algorithms behind?

- **The Pattern you draw –** *The Pattern*
  You can choose *45 buttons, 6 colours and 5 directions*. 6 colours by 5 directions that equals 30 levels. 30 levels by 45 buttons equals *1350 possibilities* to activate a button. All buttons are linked with each other and each button correlates with a so-called *Mass Number.*

- **The addressees you allocate –** *The Mass Number algorithm*
  A so-called *Alias Addressee* is created for each addressee. The *Mass Number* will be calculated by certain algorithm based on this Alias Addressee.

- **The user you are**
  Each user has its own so-called *Individual Security Constant* which ensures that two users using the same *<pattern>* will NOT use the same *<password>.*
  The user gets an individual key set. These keys are transformed into the Individual Security Constant.

FABULAROSA Algorithms behind?

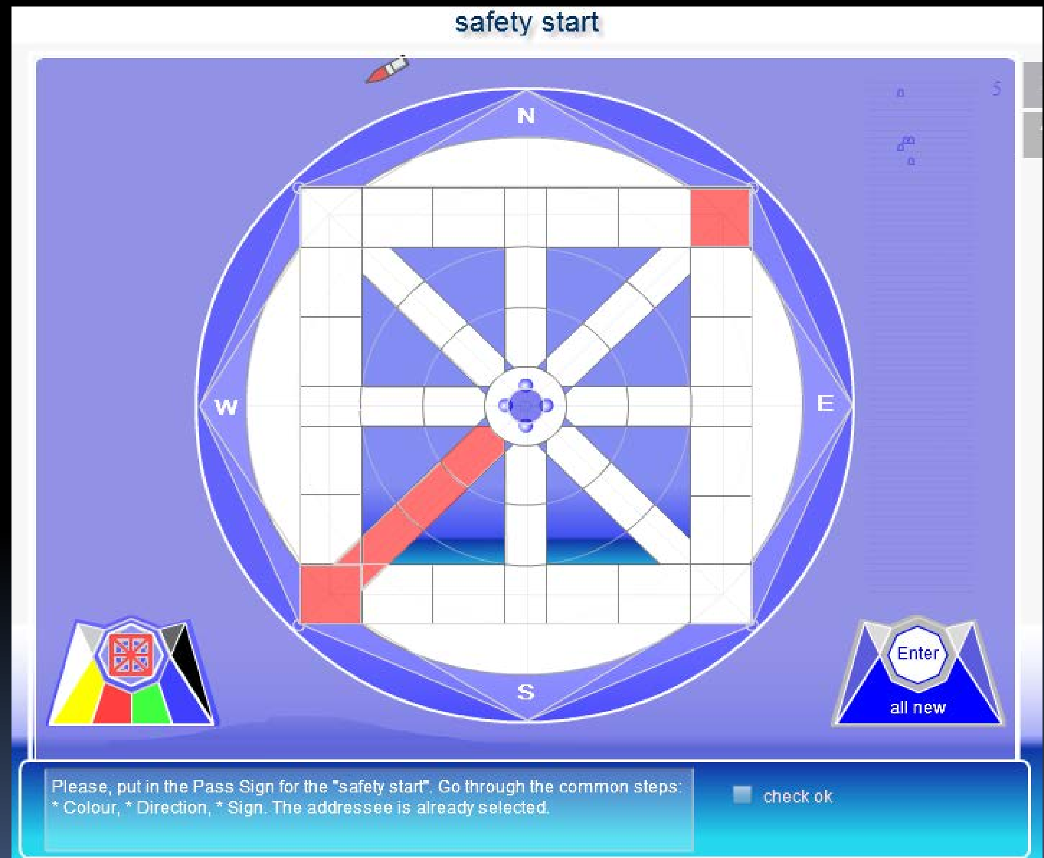
The Password is calculated by the *FabulaRosa Algorithm* based on the *Pattern*, the *Mass Number Algorithm* and the *Individual Security constant*. There are two steps:

1. The password is calculated with a length of *1350* characters using the *UTF16* character set.
2. The password is customized to the requirements of the addressee regarding the feasible length and the character set

FABULAROSA Safety Start Algorithm

The *Safety Start Mode* protects the App against unauthorized access.

The entire user data, including all the information about the addressees are encrypted by an *own-developed algorithm.* You can't proceed within the app without drawing an additional pattern.



THE AUGMENTATION
THE FIVE NEW PROTOCOLS

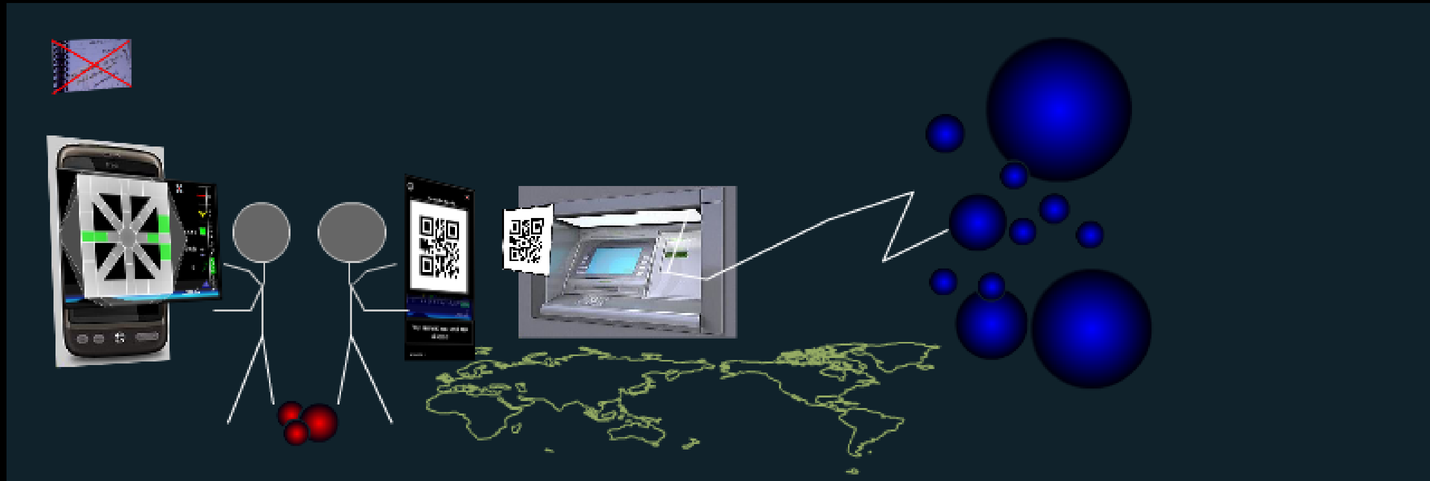
FABULAROSA

Innovation?

Applying Five New Protocols

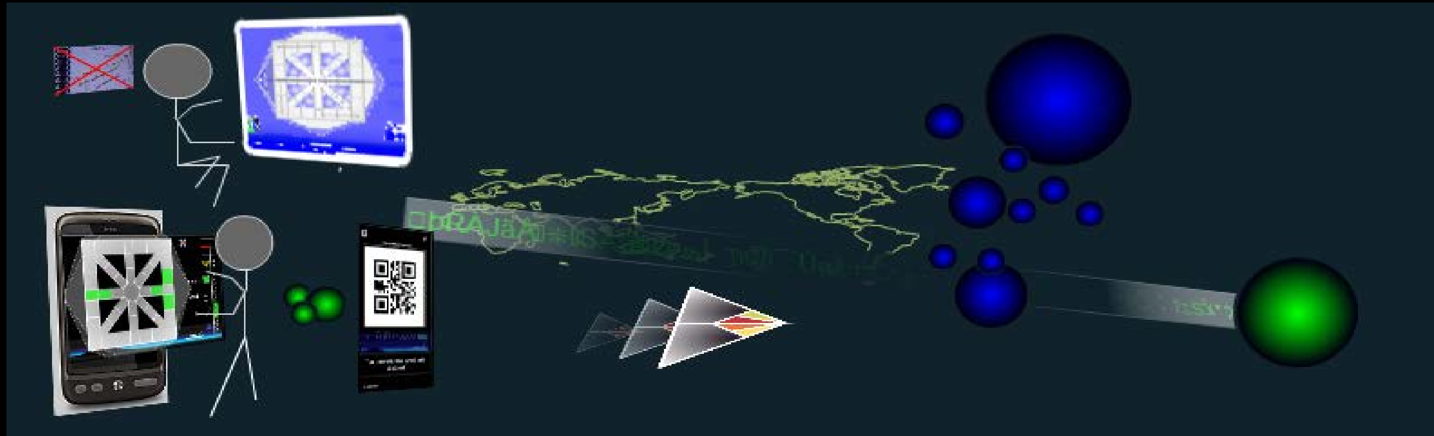
1. QR Code
2. Stealth Mode
3. Secret Based Login
4. Encrypted User Data
5. Multi Instance Authentication/ Scrambled Secrets

Protocol 1 – QR Code



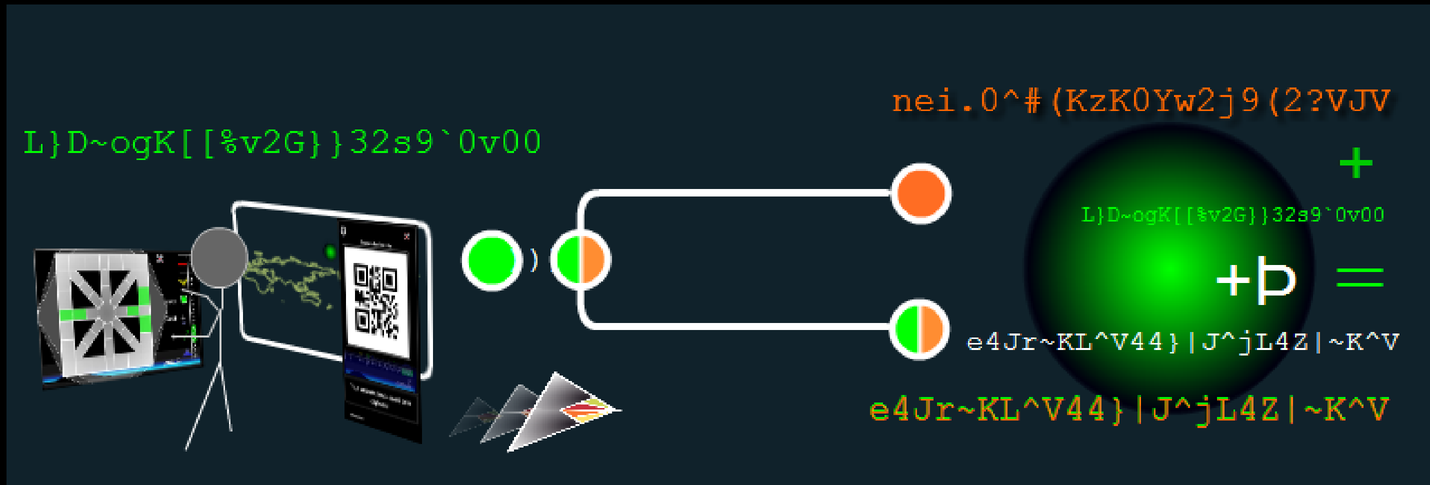
Between the communication partners: personal smart phones and public devices, a password is transferred by QR Code.

Protocol 2 – Stealth Mode



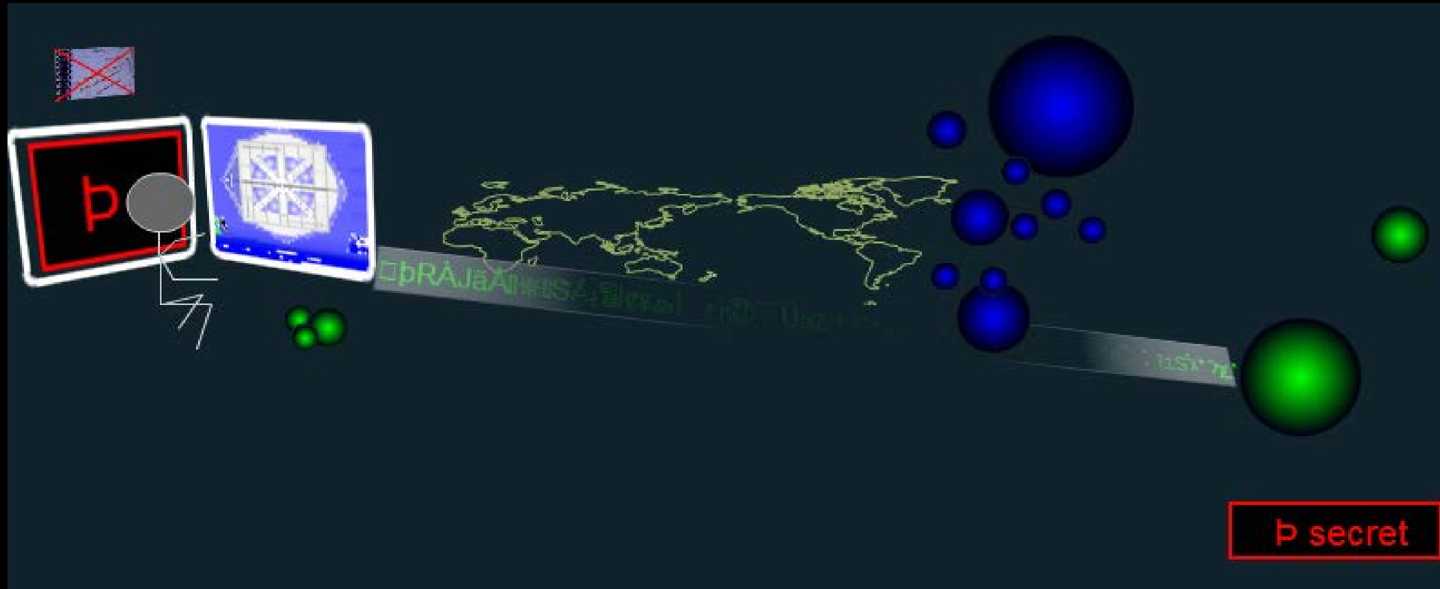
The password is modified that it cannot be identified. The type of the modification is changed within a certain time frame.

Protocol 3 – Login with a Secret



Once the password will be exchange with the addressee.
Later no password will be exchanged.
Only a content-free string will be send to the user.
During the handling of the login by FabulaRosa the string will be scrambled.

Protocol 4 – Encrypted User Data



The user data were encrypted and later locally decrypted within FabulaRosa by using the complex password matrix; suitable for

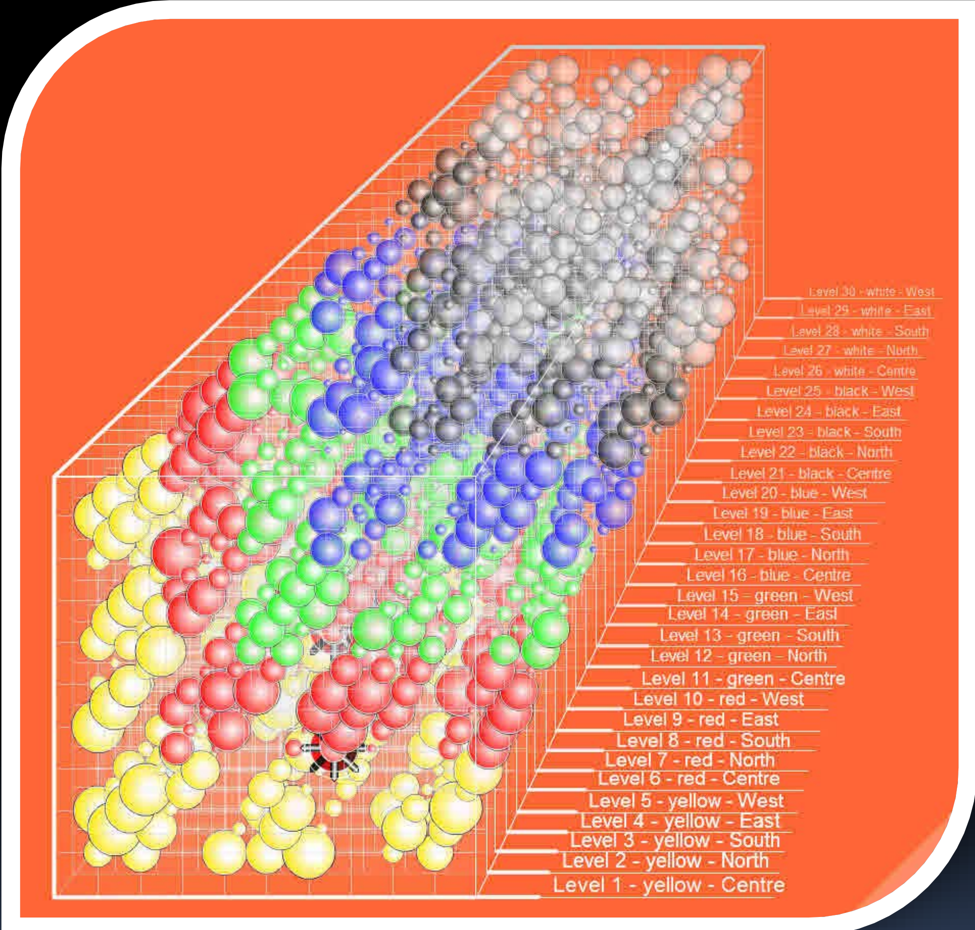
- user's credentials,
- files,
- applications. (i.e. Apple’s app development data - signatures and "mobile provisions„)

Protocol 5 – Multi Instance Authentication / Scrambled Secrets

Up to eight instances which could be a mixture of different technical factors with different simultaneous authorities can be involved in the process.

Feasible scenarios:

- image authentication,
- real electronic money
- joined control of machines.



LIVE DEMO SCRAMBLED SECRETS

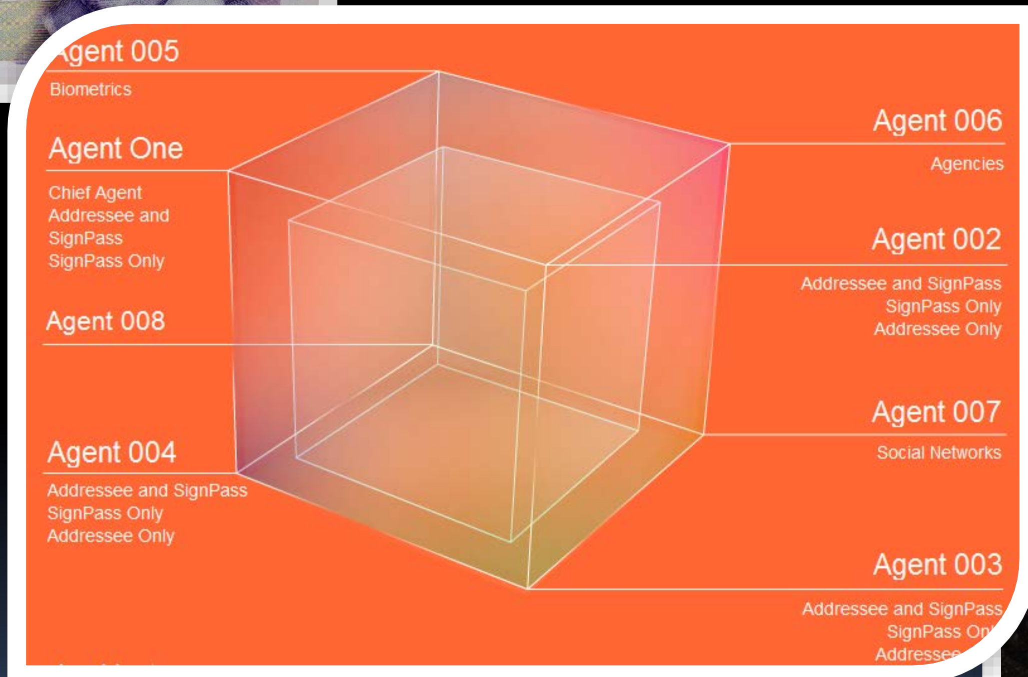
FABULAROSA

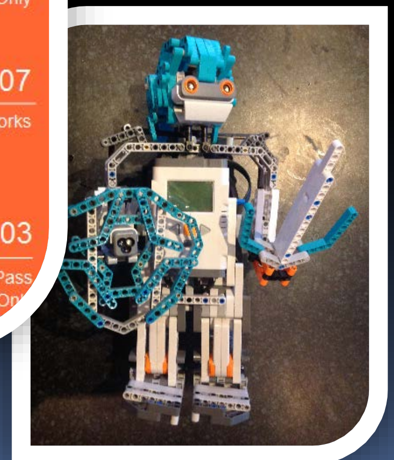
Thesis

The Keyhole Security has to overcome

- by participating of several difference instances in the authentication process
- by protecting the „values“ themselves by encrypting

FABULAROSA SCRAMBLED SECRETS APPS





FABULAROSA

THE JOURNEY IS A LONG ONE!

THE NEW NEEDS FRIENDS

www.fabularosa.com
www.kikusema.com

@FabularosaOne
@kikusema
@cyopblog

www.facebook.com/fabularosa

Unrevealed patterns in password databases Part one: analyses of cleartext passwords

Norbert Tihanyi[1], Attila Kovács[2],
Gergely Vargha[1], Ádám Lénárt[1]

[1] National Security Authority of Hungary,
{norbert.tihanyi, gergely.vargha, adam.lenart}@nbf.hu
[2] Eötvös Loránd University, Budapest, Hungary, Dept. of Computer Algebra,
attila.kovacs@compalg.inf.elte.hu

Abstract. In this paper we present a regression based analyses of cleartext passwords moving towards an efficient password cracking methodology. Hundreds of available databases were examined and it was observed that they have similar behavior regardless of their size: password length distribution, entropy, letter frequencies form similar characteriscics in each database. Exploiting these characteristics a huge amount of cleartext passwords were analyzed in order to be able to design more sophisticated brute-force attack methods. New patterns are exposed by analyzing millions of cleartext passwords.

1 Introduction

IT security companies publish complete IT security solutions from week to week which are able to protect complete IT systems. Large IT companies spend more and more money each year on IT security. In spite of all this the amount of compromised data are continuously increasing. The high-profile incidents are constantly published but there are a lot of incidents which remain in haze. The purpose of this study is not to resolve this contradiction, but rather to point out what kind of information can be gathered from cleartext databases.

1.1 Recent data breaches relating to passwords

In the last few years many security incidents have been published [1]. Hundreds of large databases containing passwords have been compromised and are accessible on the Internet. It has been reported that millions of usernames and passwords were stolen from huge companies. Without loss of generality we enumerate a few major incidents relating to password leakage:

- **September 2014:** 4.93 million Gmail usernames and passwords leaked [2] to a Russian Bitcoin forum. Close inspection revealed that the user details are old (3+ years). The retrieving method is suspected to be multiple individual targeted hacks rather than one big data leak.

- **June 2014:** The Rex Mundi hacking group stolen details from users in France and Belgium from Dominos Pizza database [3]. User's details including 650 000 passwords, email, home addresses and phone numbers.
- **April 2014:** A bunch of AOL accounts have been hacked [4]. AOL says that a "significant number of user accounts" are affected and that the breach involves accessing information associated with these accounts. It seems that those responsible for the security breach have been able to gain access to email addresses, postal addresses, and address book contact information, as well as encrypted versions of passwords and answers to security questions.
- **March 2014:** 145 million eBay accounts were compromised in massive hack including email addresses and encrypted passwords [5]. It is unclear if or when hackers will be able to break the encryption protections of passwords.
- **November 2013:** MacRumors user forums have been breached by hackers who may have acquired cryptographically protected passwords belonging to 860 000 users [6].
- **September 2013:** Over 150 million breached records from Adobe have surfaced online [7]. Hackers obtained access to a large swathe of Adobe customer IDs and encrypted passwords and removed sensitive information (i.e. names, encrypted credit or debit card numbers, expiration dates, etc.). It concerned approximately 38 million Adobe customers.
- **July 2013:** Hack exposed e-mail addresses, password data for 2 million Ubuntu Forum users [8]. The discussion forum for the operating system was compromised leaking personal details and passwords. The passwords were cryptographically scrambled using the MD5 hashing algorithm – considered an inadequate means of protecting stored passwords by security experts.
- **May 2013:** Drupal reset login credentials after hack exposed password data [9]. Malicious files placed on association.drupal.org servers via a 3rd-party application. Usernames, email addresses, country information and cryptographically hashed passwords were exposed.
- **April 2013:** Scribd, the "world's largest online library", admits to network intrusion, passwords have been breached [10]. The hack resulted in a few hundred thousand stolen passwords.
- **April 2013:** 50 million usernames and passwords lost as LivingSocial "special offers" site hacked [11]. Online criminals gained access to user names, email addresses, dates of birth and encrypted passwords for 50 million people.
- **March 2013:** Evernote asked its 50 million users to reset their passwords following an attempt to hack the note-taking network [12].
- **February 2013:** 250 000 accounts believed compromised from Twitter [13]. Hackers had access to limited user information – usernames, email addresses, session tokens and encrypted/salted versions of passwords.

These databases are very good starting points for analysis of passwords. Hundreds of databases had been analyzed.

1.2 Earlier results on passwords

In the first subsection we showed that a huge number of compromised password data are present in order to analyze them thoroughly. Several databases can be downloaded from various sources containing millions of passwords. Numerous articles can be found on the Internet concerning password analysis (see e.g: [14–20]). They analyzed password length and character type distributions, dictionary words penetrations, dictionary attack success ratios, etc. One of the largest cleartext password database available on the Internet is the RockYou database. The analysis of 32 million cleartext password leaked from RockYou shows the following character type distributions:

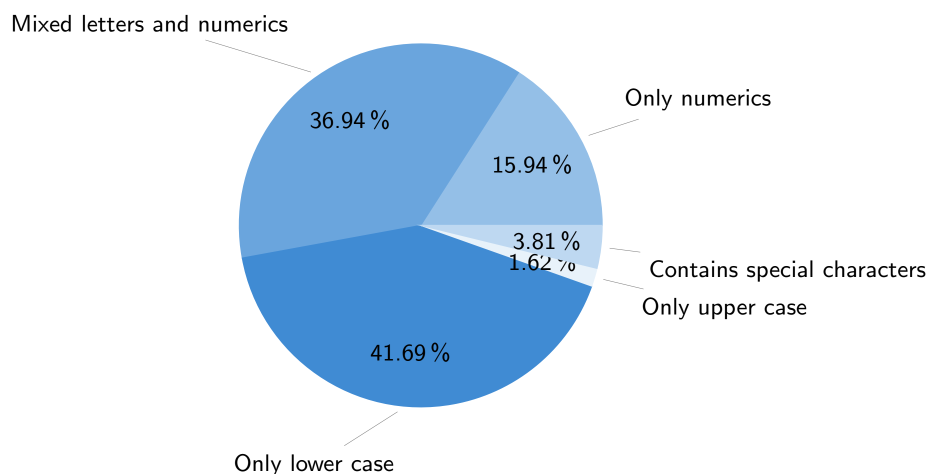


Fig. 1: RockYou password character type distribution

It was also measured that 40-50% of compromised password hashes can be recovered within a short period of time using brute-force attack. It is well-known that most of the users are not using complex or random passwords, they are choosing passwords often from dictionary. Exploiting this human behavior one can decrypt approximately 70-85% of the passwords within few days in a single notebook using dictionary attack with some (sometimes sophisticated) rules. What about the unrecovered passwords? Are these passwords really secure? The aim of this and the next papers are to describe an efficient password cracking methodology based on more than three years of experimental researches conducted by the National Security Authority of Hungary (hereafter: HuNSA). The methodology based on regression analyses of cleartext passwords. It has turned out that the approach can be used to crack encrypted passwords more efficiently than brute-force attacks or dictionary attacks. Most of the cases we were able to decrypt more than 90% of the encrypted passwords irrespectively of the applied cryptographic algorithm.

2 Encrypting cleartext passwords

Most companies use strict privacy standards and assert that all private data is stored securely. However, the reality does not justify that sense. In lots of cases the compromised data is stored in cleartext, including passwords. Storing passwords in cleartext is the worst security method ever, but storing the passwords in encrypted form is often not much better. Our main idea is based on the following

Observation. *Different password databases have similar behavior regardless of their size: password length distribution, entropy, letter frequencies show similar characteristics in every examined database.*

Based on this observation, it can be concluded that there is no significant difference between hashed and cleartext password databases. The only difference is that we can not analyze the uncracked passwords as cleartexts. In order to be able to find common patterns in these unrecovered passwords we reverse the research method. Let us encrypt a cleartext database and try to decrypt it with brute-force and dictionary attacks. In other words we simulate a real encrypted database. The remaining uncracked passwords have our main interests. We are able to analyze the length-distributions, complexity of the passwords, etc. of the remaining cases because we have the full cleartext database. During more than three years of experience we were able to find new patterns in cleartext password databases which can be used against hashed password databases.

2.1 Password strength

Password strength is a measure of the effectiveness of a password standing up against guessing and brute-force attacks. It is a function of length, complexity, and unpredictability. The effectiveness of a password of a given strength is strongly influenced by the design and implementation of the authentication system. There are two factors to consider in determining password strength: the average number of guesses the attacker must perform to figure out the correct password, and the ease with which an attacker is able to check the validity of each. The first factor is determined by the length of the passwords, by the alphabet size they are drawn from and by the randomness or predictability of the password creation process. The second factor is specified by how the password is stored and used. This factor is regulated by the design of the password management system and beyond the control of the user. Passwords can be created (1) automatically (applying some randomization), (2) by humans, or (3) by a mixture of them. Humans choose passwords guided by restricted set of rules or suggestions. In this case only estimates of the strength are possible since humans tend to follow some kind of mental patterns. Some of these patterns are collected into dictionaries even in various human languages. Analyzing the uncracked passwords lead us to design more sophisticated algorithms and rules in order to achieve 95-98% of success ratio in cracking against encrypted password databases.

2.2 Password length distribution

If one would like to design an algorithm that can generate human-like passwords the first step is to analyze the password length distributions (hereafter: PLD). Password length distributions of databases are more complex than a simple normal distribution. For example, in 96% of password databases there are more 6 and 8 character long passwords than 7 character long ones. This interesting behavior can be observed in non-english databases as well (see e.g: [21]). It seems that there is a break in the distribution, and can not be approximated by a simple function. By analyzing more than 100 different databases with more than 150 millions of passwords it has turned out that the PLD function can be approximated by a mixture of two modified Gaussian distribution function. Let us define the following password distribution (hereafter PWD) function based on empirical analyses and researches:

$$\mathrm{PWD}(x, D_n, \sigma, \mu_1, \mu_2) = D_n \cdot \frac{1}{2\sqrt{2\pi}}\left(e^{\frac{-(x-\mu_1)^2}{\sigma^2}} + e^{\frac{-(x-\mu_2)^2}{\log(\sigma)/1.5}}\right) \tag{1}$$

where D_n is the number of password in the database, σ is the standard deviation from the average password length with an ϵ error, μ_1, μ_2 are the most common password lengths in the database (e.g. $\mu_1 = 6, \mu_2 = 8$). RockYou database is a good starting point to test the correctness of $\mathrm{PWD}(x, D_n, \sigma, \mu_1, \mu_2)$. In our particular example $\mathrm{PWD}(x, 32603388, 2.1, 8, 6)$ is applied. Figure 2 shows the real password length distribution (PLD) and the approximated distribution function (PWD) together.

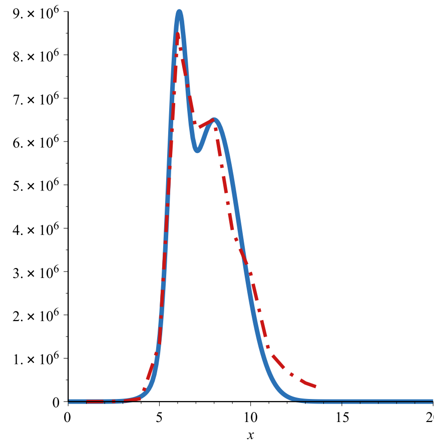
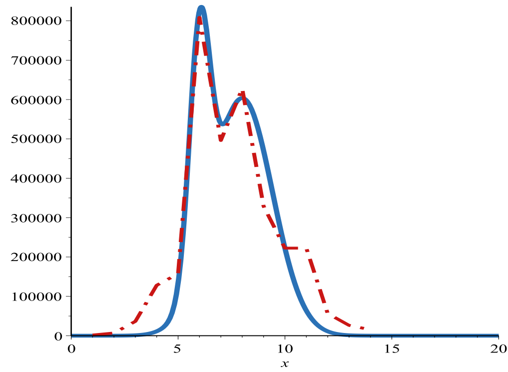
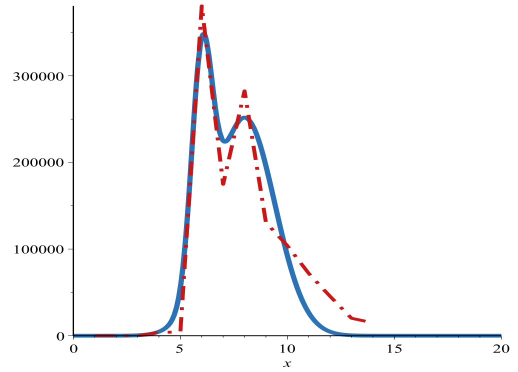


Fig. 2: Rockyou real PLD (red) and approximated PWD (blue).

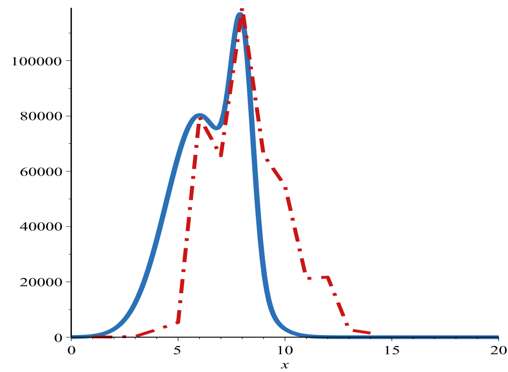
It has turned out that the PWD function can be used to approximate PLD with appropriate accuracy (≈ 95% of the cases). Without going into the details one can see 6 more examples in Figure 3.



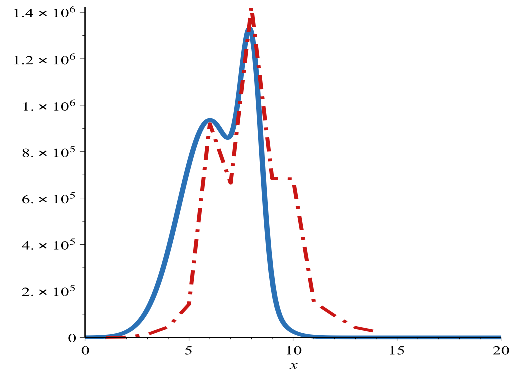
(a) Youporn



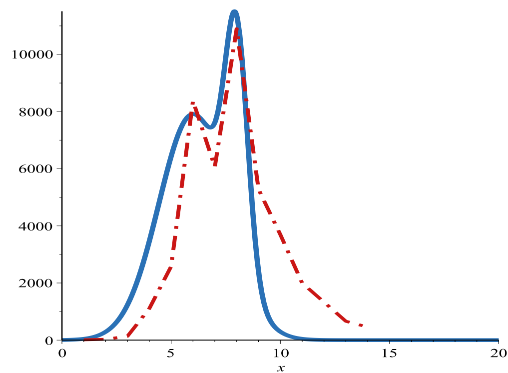
(b) Yandex



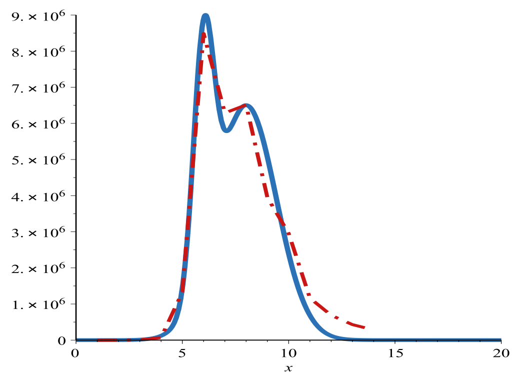
(c) Yahoo



(d) Google



(e) Unpublished



(f) RockYou

Fig. 3: Password length distributions of various databases and distribution data.

2.3 Letter frequency distribution

It is well-known that letter frequency distribution is not uniformly distributed because passwords are not chosen randomly. Table 1 shows the first 10 most common characters in different databases:

Database	**The first 10 most common letter**
Google	$a, e, 1, i, n, r, o, s, 2, l$
Yahoo	$a, e, i, o, 1, r, n, s, l, t$
RockYou	$a, e, 1, i, o, n, r, l, s, 0$
YouPorn	$a, e, 1, o, i, r, n, s, 2, 1$
Unpublished	$a, e, i, o, s, k, 1, r, n, t$

Table 1: Letter frequencies in various databases.

If passwords were truly random, each character should occur with probability $1/N$ where N denotes the number of different characters in the database. It can be seen in Figure 4 that the actual password occurrence is similar to some inverse logarithmic distribution ($1/\log N$). Analyzing more than 50 databases

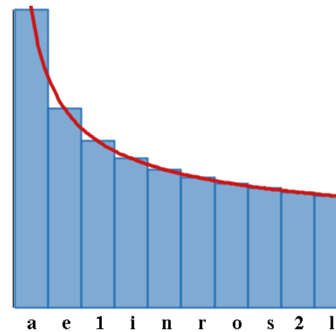
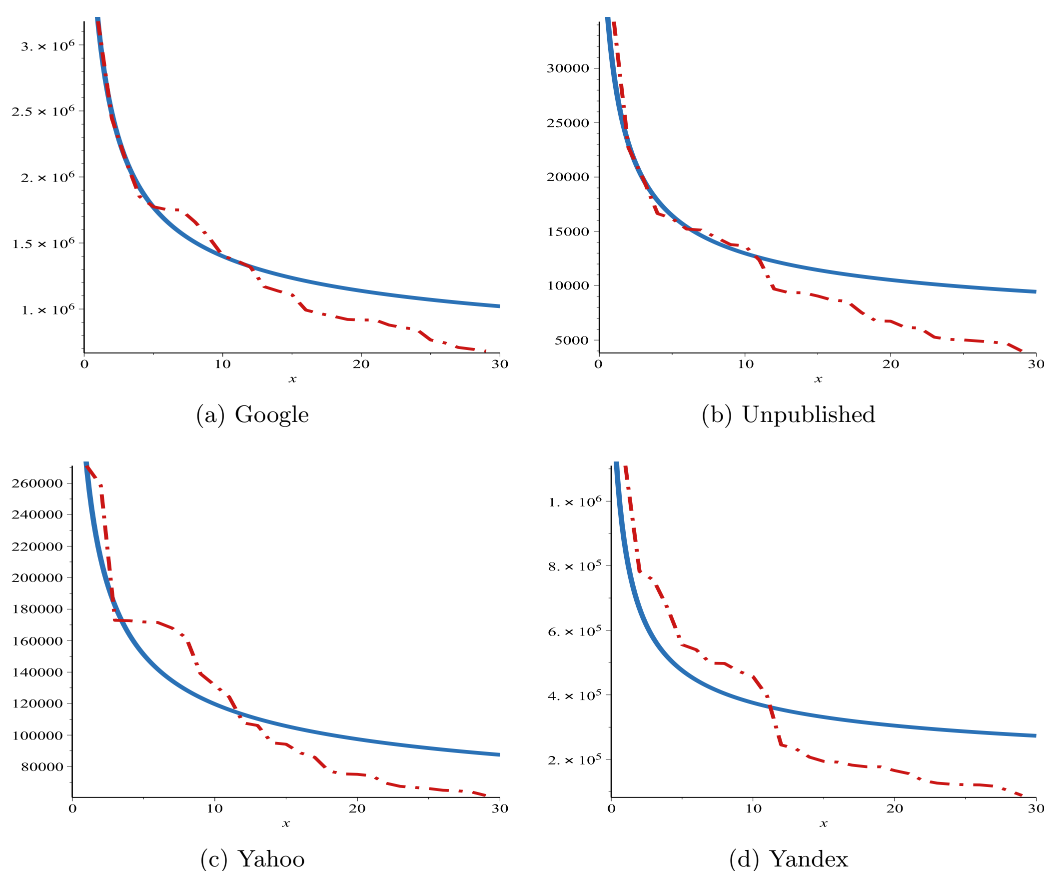


Fig. 4: Letter frequency distribution

it has turned out that letter frequency distribution can be approximated by the following function:

$$LF(x, D_n, B) = \frac{D_n}{B \cdot \ln(x+2)} - \frac{x^3}{B \cdot \ln(x+2)} \quad (2)$$

where D_n is the number of passwords in the database and B is an appropriate constant between 1 and 1.5. Most of the cases the best approximation can be achieved by $B \approx 1.255$. Without going into the details one can see 4 more examples in Figure (5).



(a) Google (b) Unpublished

(c) Yahoo (d) Yandex

Fig. 5: Letter frequency distribution of various databases

Letter frequency distribution can be used to speed-up the brute-force cracking method. Take into consideration the first N most common characters one can apply a much more sophisticated brute-force attack.

2.4 Entropy distribution

Password strength is specified in terms of information entropy, measured in bits. Password entropy predicts how difficult a given password would be to

crack through guessing, brute-force cracking, dictionary attacks or other common methods. There are different approaches for password entropy computation. It is known that the NIST SP800-63 document does not provide a valid metric for measuring the security provided by password creation policies [22].

The information entropy H_0 of a random password can be given by the formula

$$H_0 = \log_2 N^L = L \log_2 N \tag{3}$$

where N is the number of possible symbols and L is the number of symbols in the password. For example, using 95 printable ASCII characters in the symbol space the entropy per symbol is 6.57 bits. Recently, statistical metrics for individual password strength were proposed [22]. These enable rating the strength of a password given together with a large sample distribution without assuming anything about password semantics.

Most of the cases the database charset is unknown, so one should calculate entropy based on the occurrence of the observed characters. Given a string S of length n where $P(s_i)$ is the relative frequency of each character, the entropy of a string (in bits) can be calculated as

$$H_1 = -\sum_{i=0}^{n} P(s_i) \log(P(s_i))\,. \tag{4}$$

For example $H_1(password)$ is 2.75 bits. We calculated H_1 for each password in more than 50 databases. The result can be seen in Figure 6. Most of the

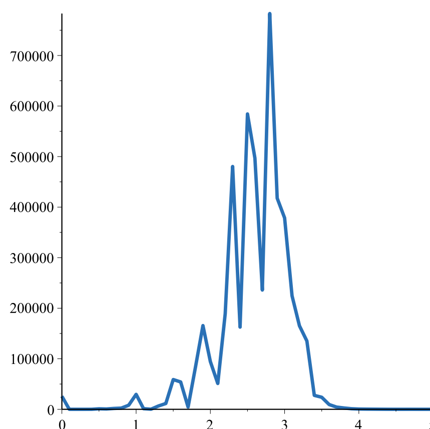


Fig. 6: Entropy distribution of password databases

passwords have approximately 2.75 entropy in each examined databases. This means that most of the passwords in the databases apply 7 different characters

independently of their size. The 8 character long passwords are using 7 different characters in $\approx 65\%$. Table 2 shows some examples for $H_1 = 2.75$ bits passwords.

*mantra*06	*passw*0*rd*
*liza*0522	*Buddy*608
Nicepass	*gingerz*2
*odai*7*ibt*	*dk*239*dhg*
snickers	*v*8*splash*

Table 2: Passwords with $H_1 = 2.75$ bits entropy

2.5 The most common passwords

The most common passwords are well studied in the last few years by several authors. Table 3 shows the most common passwords in the leaked Gmail database.

1.	123456	6.	12345678
2.	*password*	7.	111111
3.	123456789	8.	*abc*123
4.	12345	9.	123123
5.	*qwerty*	10.	1234567

Table 3: Top 10 most common passwords in the Gmail database

Analyzing more than 50 databases and more than 150 millions of passwords it has turned out that the accumulated occurence of the first most common passwords are approximately logarithmic. The most common password in the Gmail database can be found 47757 times. The first 10 most common passwords occur 102030 times in the database, the first 100 passwords occur 19257 times, and so on. This can be approximated by the following function:

$$M(x, D_n, r) = \log(x)^2 \cdot \sqrt{D_n} \cdot r^2 + x \,, \tag{5}$$

where r is an appropriate constant close to 1 or 2 depending on the database type. Figure 7 shows the cumulative sum of the first 100 000 passwords of the Gmail database, where $M(x, 4927290, 2.1)$ was applied.

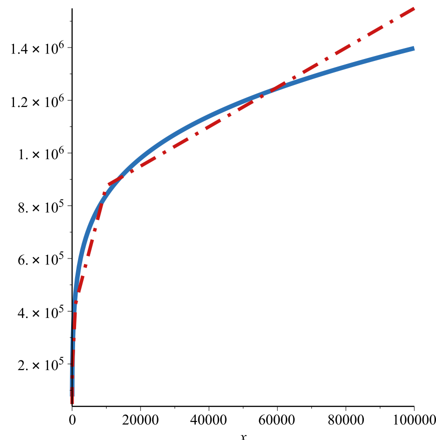


Fig. 7: Cumulative common password distribution

3 Conclusion and Future Work

The observations mentioned in the previous sections can be used against encrypted databases. In order to make the cracking method more efficient one can apply a special brute-force attack based on these behaviors.

Combining the characteristics of password length distribution, letter frequency distribution and entropy distribution functions a special brute-force attack can be designed and implemented. The entropy distribution of a normal brute-force attack distributes uniformly among strings.

For testing purposes HP EliteBook 8570w workstation class laptops were used which have mid-high range dedicated GPU called AMD FirePro M4000 built on GCN-architecture. For cracking oclHashcat 1.31 was used with the recommended AMD Catalyst 14.9 VGA driver package on x64 Windows 8.1 Pro. We used the Gmail cleartext database with SHA-3 (Keccak) to create a test database with the original passwords. AMD FirePro M4000 was able to decrypt approximately 29.8M passwords/sec with brute-force attack under Hashcat. We implemented a special brute-force attack (SBFA) that take entropy and letter distribution into consideration at the same time [3]. The modified SBFA is approximately $11-13\%$ slower than an unconditional brute-force attack. Analyzing more than 50 databases and analyzing uncracked passwords distributions, and patterns it has turned out that using this slower SBFA, approximately $23-25\%$ overall performance improvements can be achieved.

As the result of the research shows, the special brute-force attack is much more efficient than conventional. This is especially important if there is a specific time window to decrypt the passwords. The methods of attack always evolve, and the speed of decrypting is constantly rising, but the attacker has definite

[3] The exact C++ source code will be published soon.

time to decrypt the encoded passwords, so the result of this research is always relevant.

First, as a continuation of the research we want to prepare a special module that can be used in the most popular password-cracking applications. Secondly, we believe that there are further opportunities for this research; the speed of decrypt can be further optimized, so we will continue our research in this direction. Thirdly, based on the results of the research we want to create an optimal level of dictionaries with different size, that provide a much faster decryption within the specified time window.

Choosing appropriate dictionary attack and brute-force attack is important against secure hash algorithms like Bcrypt.

Acknowledgment

We would like to thank the opportunity to the Hungarian NSA to grant access to unpublished hungaraian databases that can not be found on the internet. Databases are originated from different vulnerability assessments and security hardening projects conducted by the National Security Authority of Hungary. For this project only truncated databases were provided. It means that databases do not contain any additional personal information about users, so it cannot be determined to whom the password belongs to. Password analysis and publication of this article is made with the authorization of the Hungarian NSA.

References

1. http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/ (last visited: 23 October 2014)
2. http://thenextweb.com/google/2014/09/10/4-93-million-gmail-usernames-passwords-published-google-says-evidence-systems-compromised (last visited: 23 October 2014)
3. http://www.theguardian.com/technology/2014/jun/16/dominos-pizza-ransom-hack-data (last visited: 23 October 2014)
4. http://blog.aol.com/2014/04/28/aol-security-update/ (last visited: 23 October 2014)
5. http://bgr.com/2014/05/27/ebay-hack-145-million-accounts-compromised/ (last visited: 23 October 2014)
6. http://www.wired.co.uk/news/archive/2013-11/13/mac-rumours-forums-hacked (last visited: 23 October 2014)
7. http://www.theverge.com/2013/11/7/5078560/over-150-million-breached-records-from-adobe-hack-surface-online (last visited: 23 October 2014)
8. http://arstechnica.com/security/2013/07/hack-exposes-e-mail-addresses-password-data-for-2-million-ubuntu-forum-users/ (last visited: 23 October 2014)
9. http://arstechnica.com/security/2013/05/drupal-org-resets-login-credentials-after-hack-exposes-password-data/ (last visited: 23 October 2014)

10. http://nakedsecurity.sophos.com/2013/04/05/scribd-worlds-largest-online-library-admits-to-network-intrusion-password-breach/ (last visited: 23 October 2014)
11. http://nakedsecurity.sophos.com/2013/04/27/livingsocial-hacked-50-million-affected/ (last visited: 23 October 2014)
12. http://www.wired.co.uk/news/archive/2013-03/04/evernote-hacked (last visited: 23 October 2014)
13. http://www.wired.co.uk/news/archive/2013-02/02/twitter-hacked (last visited: 23 October 2014)
14. Florencio D., Herley C. A large-scale study of web password habits, Proceedings of the 16th International Conference on World Wide Web, Association for Computing Machinery, Banff, Alberta, Canada, (2007) 657–666.
15. Shay, R., Komanduri, S., Kelley, P.G., Leon, P.G., Mazurek, M.L., Bauer, L., Christin, N., and Cranor, L.F.,: Encountering Stronger Password Requirements: User Attitudes and Behaviors. In SOUPS 10: Proceedings of the 6th Symposium on Usable Privacy and Security. ACM, (2010)
16. Kelley, P.G., Komanduri, S., Mazurek, M.L., Shay, R., Vidas, T., Bauer, L., Christin, N.,, Cranor, L.F., and Lopez, J.: Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. Carnegie Mellon University, Tech. Rep. CMU-CyLab-11-008, (2011)
17. A brief analysis of 40 000 leaked MySpace passwords, Blog post: http://www.the-interweb.com/serendipity/index.php?/archives/94-A-brief-analysis-of-40,000-leaked-MySpace-passwords.html (last visited: 23 October 2014).
18. Bonneau, J.: The science of guessing: analyzing an anonymized corpus of 70 million passwords. 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, (2012)
19. Weir, M., Aggarwal, S., Collins, M., Stern, H.: Testing metrics for password creation policies by attacking large sets of revealed passwords. In ACM Conference on Computer and Communications Security (2010) 162–175.
20. DellAmico M., Michiardi P., Roudier Y. Password strength: An empirical analysis. Proceedings of the 29th Conference on Information Communications, San Diego, USA, (2010) 983–991.
21. Norbert Tihanyi, Comparison of two hungarian password databases, Pollack Periodica, Vol. 8, No. 2, (2013) 179-186.
22. Bonneau, J.: Statistical metrics for individual password strength. In: SP'12 Proceedings of the 20th international conference on Security Protocols, University of Cambridge, UK, (2012) 76–86.

Gathering and Analyzing Identity Leaks for Security Awareness

David Jaeger, Hendrik Graupner, Andrey Sapegin,
Feng Cheng, and Christoph Meinel

Hasso Plattner Institute, University of Potsdam, Germany
{firstname}.{lastname}@hpi.de

Abstract. The amount of identity data leaks in recent times is drastically increasing. Not only smaller web services, but also established technology companies are affected. However, it is not commonly known, that incidents covered by media are just the tip of the iceberg. Accordingly, more detailed investigation of not just publicly accessible parts of the web but also deep web is imperative to gain greater insight into the large number of data leaks. This paper presents methods and experiences of our deep web analysis. We give insight in commonly used platforms for data exposure, formats of identity related data leaks, and the methods of our analysis. On one hand a lack of security implementations among Internet service providers exists and on the other hand users still tend to generate and reuse weak passwords. By publishing our results we aim to increase awareness on both sides and the establishment of counter measures.

Keywords: identity leak, data breach, password, security awareness

1 Introduction

Data leaks, e.g. from Adobe[1], happen regularly on the Internet and affect millions of users. The impact of such data leakage events is often underestimated by users, even if covered in media with proper attention. Many users use the same e-mail address to complete registrations for hundreds of accounts on different sites in the web, while only a few of them maintain records or use unique passwords for such accounts. Accounts may contain sensitive information, which can be used to gain access to other accounts of the same user, especially if the password is reused.

Knowledge about the existence of data leaks is hard to gather, since they are often published in small pieces through a variety of hacker forums or personal web pages. We developed a service for tracking identity leaks, where we have solved following not-trivial problems.

– Automated discovery and collection of leaked data sets on the Internet
– Normalization of data sets
– High-speed search through billions of leaked records
– Security and privacy issues

In this paper we describe solutions for the problems mentioned above. Besides that, we provide password statistics as well as recommendations for both regular Internet users and enterprise security departments. We hope that offered measures could help to reduce the impact of the data leaks on the security and privacy of end-user accounts all over the Internet.

The rest of this paper is organized as follows. Section 2 covers related work and Section 3 presents our implementation. Section 4 provides details on data gathering, Section 5 shows information represented in leaks and categorizes leaks by their origin, Section 6 gives an evaluation of used passwords in the analyzed leaks. Finally, we outline future work in Section 7 and then conclude in Section 8.

2 Related Work

The relevance of finding and analyzing publicly available identity data leaks on the web and deep web can be conducted from numerous reports on Internet security. Our goal is to increase people's awareness of their potentially leaked data and develop statements about the quantity and quality of existing identity leaks.

In their annual Internet security threat report[2], a well-known IT security company showcases a drastic increase of identity data leaks caused by hacking of databases. In 2013, there were 253 breaches containing more than 552 million identities[2, p.13]. Another report[3] by Risk Based Security, sponsor of the DataLossDB[1], even estimates 822 million affected records from 2,163 incidents.

The scientific community is strongly focused on developing and evolving technologies to prevent and detect data leaks in companies. This is a fast developing field involving a lot of interesting publications. However, for those innovations to be taken over into practice we found lack of existing awareness regarding security risks. Filling this gap is where we see this work's contribution to Internet security.

[1] DataLossDB - `http://datalossdb.org/`

2.1 Scientific Publications

Scientific work related to password and identity leaks mostly focuses on breach prevention from an enterprise point of view (e.g. [4]). This paper is related to the cases where prevention mechanisms failed and user identity or password data was leaked by internal or external attackers. Hence, relevant topics are the security of stored passwords after they were exposed to the public, such as leak detection on the Internet and password security in general.

The latter is of special importance for us since a securely generated and obfuscated password should be practically useless to an attacker, even if stolen from a web server. Unfortunately, a large amount of leaked passwords can be reversed, guessed, or brute forced fairly easy. The main reasons are weak user-generated passwords and insecure obfuscation mechanisms[5]. Additional threats on the Internet are reuse of passwords[6] and publicly available information[7].

Our mechanisms of web monitoring regarding identity leak data and normalization are self-developed, since no relevant publications in this area exist. To prevent the spreading of potentially abusable source code we forbear to publish any implementation details that enhances the progress of leak data collection.

2.2 Products and Services

Due to rising security awareness, enterprises as well as individual persons are more and more interested in what happens to their data and who has access to it. Some companies have identified this gap in the market and provide several products to sell information about existing identity data leaks to end users. In addition non-commercial solutions by government agencies (e.g. BSI[2]) and private programmers exist to aid a public interest.

We noticed that products with commercial interest such as Survela[3], BreachAlarm[4], or PwnedList[5] are based on much larger data sets than other services. This follows from their continuous crawling of the web and deep web. However, it is hard to measure the quality of their methods, because in general they are not revealed to the public. Besides, the quality of data leaks itself is questionable as well, since cracking and testing the correctness of a user's password for a service or website would constitute a criminal offense.

[2] BSI Security Check - `https://www.sicherheitstest.bsi.de/`
[3] Survela - `https://survela.com/`
[4] BreachAlarm - `https://breachalarm.com/`
[5] PwnedList - `https://pwnedlist.com/`

3 Identity Leak Checker Service

Our *Identity Leak Checker*[6] is a free-of-charge Internet service that allows users to search for occurrences of their e-mail-address in publicly accessible identity leaks. As part of the service, we find, collect and normalize leaked data sets, partially automatically, to make high-speed and up-to-date leak checking available to Internet users. The service was started in May 2014 and has more than 1.2 million requests and has warned more than 120,000 users of identity loss (Aug. 11th, 2014). There are 25 leaked databases with more than 172 million records incorporated into the service. Additionally, more than 3,000 leaks have been collected, which we plan to integrate into the service in the future.

The service has a search page with an input field, where users can enter their e-mail-addresses. As soon as the search is submitted, the e-mail is checked against the data set and a response e-mail is sent out to inform the user about the search result. The e-mail provides categories and approximate publication dates of the leaked data. This way no sensitive information is revealed to potentially compromised accounts.

4 Gathering of Identity Leaks

The locations in the cyberspace where identity information is published by cyber criminals are manifold. We use these locations as sources for data collection and categorize them by their purpose, i.e. *leak provision* and *leak announcement*.

4.1 Leak Provision

These sites on the Internet are usually only hosting the leaks, meaning they store and deliver the leaks, but are not aware that they are hosting this critical data. Because these providers can host any kind of data, it is hard to monitor them for new appearing leaks. The following list gives an overview of provider types and whether they are general-purpose providers or used exclusively for leak distribution.

General-Purpose Providers

– **Paste pages** are web sites that allow anyone on the Internet to publish small text snippets. While these pages are often used for distributing code snippets, event log files and texts, a selection of these pages is

[6] HPI Identity Leak Checker - `https://sec.hpi.de/leak-checker`

also massively used to distribute leaks. A famous example in this category is PasteBin[7][8]. Paste pages are used to distribute small-sized leaks with around 100 - 10,000 identities. On a single day, we can find around 10 - 100 leaks distributed over these pages.

- **File Hosting Providers** are a common place to distribute leaks, which are too big in size for paste pages or are organized in directories. A popular site in this category is AnonFiles[8], which allows people to anonymously upload and share files. Leaks distributed over file hosting providers have usually around 10,000 - 10,000,000 identities. On a single day, we can find up to 10 leaks distributed in this way.
- The **BitTorrent** ecosystem is used to distribute leaks that are of major interest to the public, being downloaded many times. These leaks usually contain more than one million identities. We found this kind of distribution quite infrequent, i.e. we monitored around 2-3 publications in six months. The Adobe leak is one famous example with around 150 million user accounts[9].

Leak-Only Providers

- There are special **paste pages** by hacker groups, that are dedicated to the publishing of their leaks. The idea is similar to the general-purpose paste pages. The leak sizes are slightly larger, because the page is specifically built to publish day-to-day leaks.
- **Drop zones** are remote anonymous shares for looted data of cyber criminals. These drop zones are either manually maintained by the criminals or automatically populated by botnet clients performing phishing and keylogging. There have been multiple cases in the past, where these drop zones have been taken over by companies or researchers[10][11][12][13]. Often they contain big repositories of collected user data, such as credentials and financial details. Getting access to private drop zones is very challenging.
- **Leak database shares** are servers on the Internet, where various previously published leaks are shared between members of a community. The cases we encountered are all hosted in the Onion network[14][15] and contain hundreds of earlier and bigger leaks.
- Some hacker groups maintain their own **leak distribution pages**, which are subpages of their main website and host all their looted data. These pages should show the experience and success of the attackers and provide proof of their hacks.

[7] PasteBin - `http://pastebin.com`
[8] AnonFiles - `https://anonfiles.com`

- **Hacker Forums** are sometimes used to publish identities directly in the posts of a forum thread. In this case, the leaks are hosted by the forum. Due to limitations in the length of posts, the number of identities leaked in one post is limited to a few thousands.

4.2 Leak Announcement

These locations on the Internet do not directly host leaks, but they give directions where leaks can be found or even give direct links to leaks. Cyber criminals use these places to announce their achievements and get recognition from the community.

- **Hacker forums** are a popular source for hackers to publish any kind of information about hacking, like vulnerabilities, exploits and others, e.g. leaks. The forums are mostly free, but sometimes require registration. If there are sections in the forums specifically for leaks, often links to leak providers are given.
- **Leak announcement pages** are web sites where individuals can post links to their own leaks or are referring to leaks they found somewhere on the Internet.
- **Leak monitoring pages** are web sites that check various sources for new upcoming leaks. There are implementations that monitor paste pages and file hosters for incoming files and try to find potential leaks in the posted data. Once interesting data is found, notifications are posted to their page or on their social media accounts. Examples in this category are LeakedIn[9] or BreachAlarm.
- **Social media** web sites like *Twitter* and *Facebook* are massively used to announce leaks. The authors of the announcements are typically *Hacker or hacker groups*, *security researchers/journalists*, e.g. Brian Krebs or Troy Hunt, *automated leak monitors*, e.g. Dump Monitor, or leak *security news* pages. The posts are mostly referring to paste pages and file hosting sites, as listed in Section 4.1.

4.3 Automated Gathering of Identity Leaks

Automated gathering of leaks from the listed sources can be difficult, because the leak providers usually do not provide an interface that can be accessed programmatically. The most challenging provider type is file hosters, because they have access restrictions and protect their data with

[9] LeakedIn - `http://www.leakedin.com/`

Captchas. Paste pages on the other hand are easy to access programmatically and even have APIs sometimes. Because of these limitations, programmatically collected leaks are often small- to middle-sized.

5 Anatomy of an Identity Leak

Looking at the previously gathered identity leaks, different information sources and types of information can be identified.

5.1 Information in Identity Leaks and Data Model

The information in identity leaks varies from e-mail addresses with login credentials to any imaginable information about a person, from home address to credit-card data. In order to manage the information being found in different leaks, we created a data model that covers the most interesting and common information. The data model is object-oriented and can be seen in Figure 1.

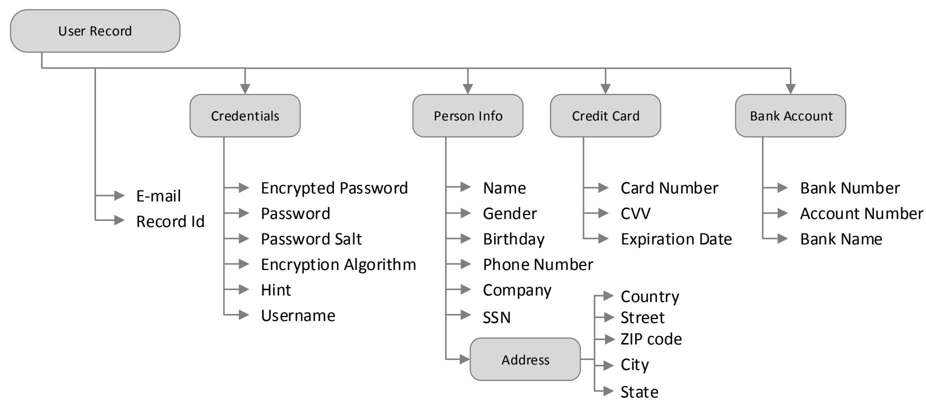


Fig. 1: Data-model for identity leak information

There is a main object `User Record` that holds information about a single record from a leaked database. Each of these objects holds the `E-mail` as the main identifier for an identity and the `Record Id` as reference to the id within the leaked database.

`User Record` has four sub-objects, covering various facets of a user, such as credentials, credit card and bank account data, and personal information, like home addresses, birthday and phone number. Each user

record can have multiple sub-objects of the same type. This happens frequently, when users have to specify multiple phone numbers or have a home and company address.

There is additional information in leaks that is currently not covered by our data model, because it is only rarely included, but could be of interest for special purposes. Such information includes IP addresses, personal websites or instant messaging usernames from AIM, Skype, or ICQ.

5.2 Types of Identity Leaks

There are multiple sources where cyber criminals obtain identity information, which is also reflected by the kind of information and structure a specific leak contains. Looking at forums dedicated to leak announcement confirms this view. They use specialized terms for leaks from certain sources, which we call leak types. We encountered following types of leaks:

Database Dumps (Dumps) A user or customer database of a web service has been hacked and downloaded, e.g. by SQL injection on the web frontend of the service. This kind of leak is most common and contains between 1,000 up to millions of records.

Phishing-/Trojan-Data (Logs) The data is collected individually from each user. This covers the collection of user data from following sources:

- Malware (e.g. trojans, keyloggers, botnet clients) that is installed on a user's computer and collects information the user enters,
- phishing websites a user visits to enter his secret information, and
- password sniffers that are used in public networks and are tailored to extract identity information from network traffic.

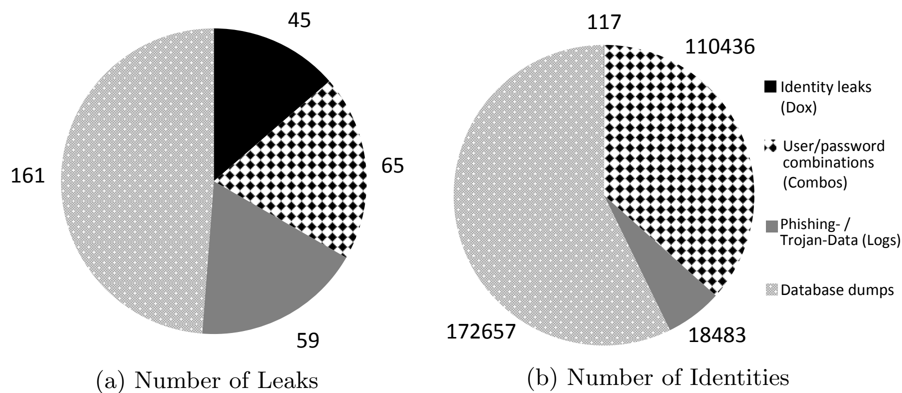
The leaks have around 100 - 10,000 records.

Full Identity Leaks (Dox) Detailed information about the identity of a single person and related people. The information is gathered from publicly available sources, such as database dumps, social networks, or personal and company websites. This kind of leak contains the most complete information about a person and is therefore also most dangerous for victims.

User/password combinations (Combos) A list of combinations of user identifiers, such as usernames or e-mail addresses, and passwords. The source of the information is mostly unknown and is either a database dump or phishing-/trojan-data. Usually, combo-lists lack information about identities' origins.

The company *High-Tech Bridge* partially confirms these origins of leaks in their analysis[8] of PasteBin leaks.

To show the distribution of leak origins, we have analyzed automatically collected leaks from May 2014. The results are shown in Figure 2. Figure 2a shows how many leaks have been found for each category. It confirms that database dumps are most common with as many as 161 leaks. Logs and full identity leaks are less often encountered. However, it should be noted that leaks with combos can also count in either dumps or logs.



(a) Number of Leaks

(b) Number of Identities

Fig. 2: Statistics for automatically collected leaks in May 2014

Figure 2b gives a rough idea on how many identities, which are counted by e-mail addresses, are affected for the different leak types. In this case most identities where leaked with dumps with around 1,072 identities per leak. On the second position are combos with around 1,700 identities per leak. Here it is also important to note, that automatically detected leaks are usually smaller in size, as outlined in Section 4.3 and therefore the identities per leak for dumps is rather low. On the other hand, the collected combos could be originating from bigger dumps. Identity leaks are least frequent and contain around 2.5 identities per leak, where leaks with multiple identities additionally cover relatives and family members.

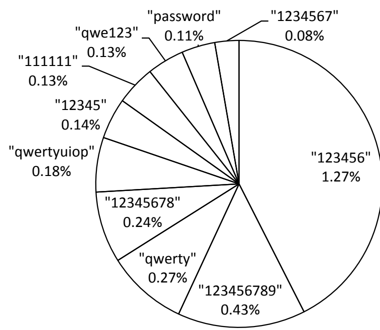
6 Evaluation

6.1 Password Distribution

We have calculated the most popular passwords used in 28 of the biggest data leaks. However, the distribution of popular passwords often dif-

fers across databases. Some passwords are presented in only one or two databases, but still could have the highest absolute frequency (among all leaked sources), due to the very high number of records in these databases. For example, 'adobe123' is the most popular password in the Adobe leak. Due to its popularity for Adobe users and the very large size of the leaked data (150 million records), the fraction of this password among all leaked data sets remains high enough to put it on the first place for all data sets.

To avoid this problem, we have sanitized our data to filter out database-specific passwords. First, we calculate the password frequency for each password in each leak source and select the top n passwords ordered by average frequency in the leak source. Second, for each of the selected top n passwords, we calculate the variance coefficient based on the distribution of source-wise password frequencies. Finally, we exclude passwords with the variance coefficient falling into the top 10% of the min-max coefficient range. The sanitized password statistic is presented in Figure 3.



(a) Top 1 - 10 Passwords

Password	Avg. frequency per source
123123	0,08%
1234567890	0,07%
11111111	0,06%
pokemon	0,06%
000000	0,06%
1qaz2wsx	0,06%
1q2w3e4r	0,05%
qazwsx	0,05%
1234	0,05%
123321	0,05%

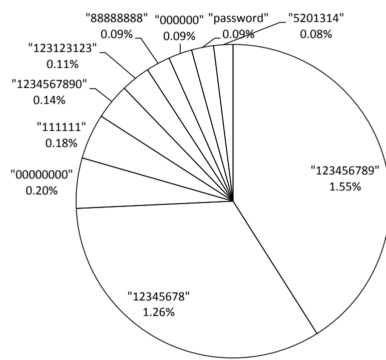
(b) Top 11 - 20 Passwords

Fig. 3: Most popular passwords by average password frequency in leak sources

Although almost every leak source contains user credentials with very weak passwords, Figure 3 implies that the aggregated fraction of the top 10 weak passwords is not that high (2.85%). Moreover, the top 20 most popular passwords make only 3.31% of all passwords on average. All in all, weak passwords are being used by about 5% of the users in an average data set, and we hope that this fraction will be reduced in the future by further dissemination of password policies.

Regional Password Distribution On top of the distribution of passwords in general, further investigation can be done on passwords originating from identities with different backgrounds, like their origin country or culture and language. By looking at these, it can be determined, whether there are factors influencing the choice of passwords. We have identified three sources in our collected leaks, that can be clearly attributed to Chinese services. All sources should contain mostly passwords from Chinese identities.

Taking the analysis methods from before, we analyzed the password distribution only for Chinese leaks. The results are shown in Figure 4.



(a) Top 1 - 10 Chinese Passwords

Password	Avg. frequency per source
7758258	0,05%
888888	0,05%
666666	0,05%
123321	0,05%
987654321	0,05%
147258369	0,04%
1314520	0,04%
111111111	0,04%
iloveyou	0,03%
1qaz2wsx	0,03%

(b) Top 11 - 20 Chinese Passwords

Fig. 4: Most popular Chinese passwords by average password frequency in leak sources

Comparing the distributions of passwords in general against Chinese passwords, it turns out that the top 2 passwords are the same, namely the number sequences 123456, 123456789. One thing that stands out of the lists, is that the Chinese passwords are mostly numerical, whereas the general passwords contain many alphabetic characters. Actually, the password *password*, *iloveyou* and *1qaz2wsx* are the only alphabetic password in the top 20. Another thing that stands out is a higher prevalence of the number 8 and 6, i.e. in the passwords *88888888* (8x8), *666666* (6x6) and *888888* (6x8) and the special, seemingly random, numbers *5201314*, *7758258* and *1314520*. However, looking up these numbers in relation to China, we could find that 6 and 8 are numbers with a very positive meaning in China and that the sequence *5201314/1314520* is pronounced like *I love you forever* and *7758258* like *kiss and love me* in Chinese. So these

passwords are quite special to Chinese people, but have low significance for non-Chinese people.

All in all, the analysis of passwords from only Chinese identities shows that there are special passwords for different regions of the world. There are passwords that are specific to the language of the region, like *5201314* or that are derived from cultural backgrounds, like the positive number 8. Surely, the analysis of regional passwords can be an interesting topic for future research.

6.2 Password Encryption/Hashing Methods

While we have analyzed passwords from various leaks, we also found a big amount of passwords that are not immediately obtainable in cleartext format. This is because they are obfuscated with different types of hash functions, such as MD5, SHA-1 or PHPass[10]. Some of the stored hash values involve salts and/or multiple iterations of a single hash function. We have analyzed the hash routines, i.e. the specific usage of hash functions, salts and iterations on all our normalized leaked databases and were able to find the distribution in Figure 5. The notation we used for describing the routines defines *$p* as *password*, *$s* as hashing *salt*, and *$u* as *username*. In one occasion, the username was used as a unique hashing salt. The diagram shows that cleartext passwords are still used in every third password database, although security experts are warning for years to properly secure passwords.

Another fact we can deduce from this diagram is that MD5, a hash function widely considered insecure[16, 17], is still extremely popular for service providers to store passwords. One third of the databases use MD5 without a salt. Passwords stored this way are highly susceptible to rainbow table attacks[18] and can be reversed extremely fast with tools like *hashcat*[11] and *John the Ripper*[12] or online hash reversing pages, e.g. `hashkiller.co.uk` providing more than 43 billion reversed MD5 hashes. 14% of databases are using multiple iterations of MD5 hashing. These hashes are far more secure than a single application of MD5 due to the fact, that effective reversing based on dictionaries is very unlikely. However, many passwords can still be reversed with rather small computational overhead. Only a small number of databases are using hash functions other than MD5. We found one source using SHA-1, another hash

[10] PHPass - `http://www.openwall.com/phpass/`
[11] hashcat - `http://hashcat.net/hashcat/`
[12] John the Ripper - `http://www.openwall.com/john/`

function that is discouraged to use and rather fast to reverse[19]. Only hash routines with a high computational overhead can be considered as sufficiently resistant against brute force attacks. Such routines use a high number of iterations of a hash function or they are intensive to calculate. Routines like *bcrypt($p.$s)*[13] and *PHPass($p)* fall under this category of hash routines with high computational overhead.

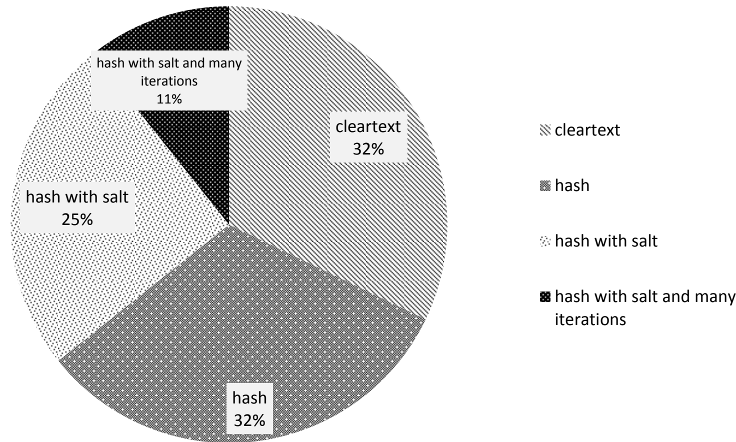


Fig. 5: Distribution of different obfuscation types of password database leaks. The parameters used are defined as follows: $p = password, $s = salt, $u = username. The most common way of storing passwords among the analyzed sources was *f = $p*, i.e. cleartext.

Figure 6 summarizes the distribution of different levels of password hashing techniques. Cleartext passwords are unprotected and hash routines involving strong hash functions, salts, and a high number of iterations are considered most secure. It is notable, that for hash routines a negative correlation, between password security and the amount of databases utilizing it, exists. However, the only strong hash routine we found, i.e. *bcrypt* and *PHPass*, was used in as little as 11% of the cases[17].

[13] bcrypt library - `http://bcrypt.sourceforge.net/`

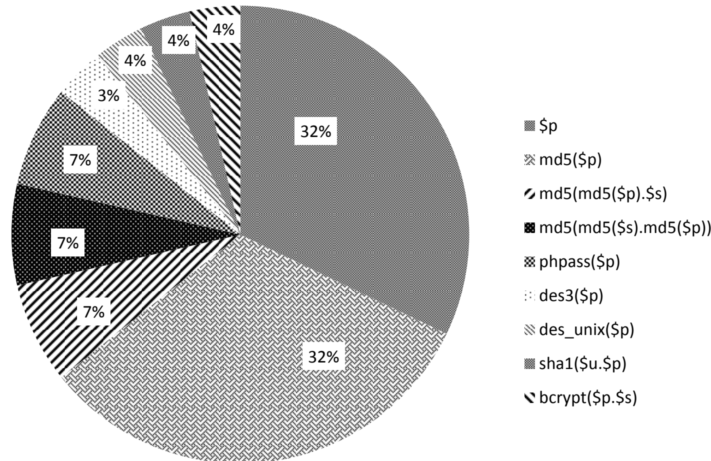


Fig. 6: Distribution of different types of password hashing techniques.

7 Future Work

Follow-up work could focus on finding today's unknown password leak sources, their monitoring, and normalization of arising data. In the current state of our project, data is being normalized semi-automatically, resulting in a model which requires a lot of time and effort. It is desirable to minimize human interaction during web monitoring and normalization of leaks, based on the various recurring data formats (e.g. filtering of false positive results of automatic data gathering). Efficiency can be increased by developing mostly automated processes, using technologies such as content analysis, machine learning, etc.

Another area of research, based on identity leak gathering, is advanced password analysis on anonymized data records. Being able to reverse specific password hashes in leaks allows insight into information that is potentially available to hackers. A controlled research environment is able to generate statistics relevant to Internet password security. E.g. the publication of top used passwords lists enables service providers to warn users to avoid specific terms in their passwords or actively prevent the creation of weak passwords. Hence, we believe this type of analysis has potential to increase Internet security in general.

Often, the passwords in leaks are hashed or encrypted and it is rarely obvious which routine of hashing has been used. We aim to enhance the classification of password databases in terms of the used hash routines. Due to the big impact of the usage of obfuscation standards on password

security, the results will be telling to automatically determine the used routine and give an estimation on the leak's password security. Hence, it allows conclusions to security of stored passwords of Internet service databases in general. This information can be used to raise security awareness, especially on the side of database owners.

8 Conclusion

The most challenging part of analyzing a large data set of identity data leaks is the normalization of numerous different formats. In some occasions big data leaks in an easy readable format are provided. However, the highest amount of sources contain only 100-1,000 or more identities and have custom formats and a questionable quality of content. Even though those sources seem less significant due to the low number of entries, in reality they are very important for the goal of our work. The media primarily focuses on big breach events where millions of identities have been revealed. This may give end users a false sense of security if they are not affected by those breaches, since they will potentially never know about their data being leaked in one of the countless small breaches.

Another challenge is the increased efficiency of identification and monitoring of leak sources. This is especially important for the search in the deep web, where usually no indexing exists. In addition some web sites (e.g. PasteBin) inhibit our efficiency of crawling by restricting the number of requests based on IP addresses.

Our work covers an increasing base of identity data leaks. Using this data we can provide up-to-date analysis and statistics of identity data breaches, insecure passwords, and the security of web content providers' password management. The results of these analyses have been made available to users by our Identity Leak Checker Service[14]. We propose this service to increase public awareness and help users to implement countermeasures against misuse of their data.

References

[1] *Important Customer Security Announcement.* `http://blogs.adobe.com/conversations/2013/10/important-customer-security-announcement.html`.
[2] Symantec Corporation. *Internet Security Threat Report.* 2014.
[3] *Data Breach QuickView: An Executive's Guide to 2013 Data Breach Trends.* Presentation. Risk Based Security, Feb. 2014.

[14] HPI Identity Leak Checker - `https://sec.hpi.de/leak-checker`

[4] Bryan Parno, Jonathan M. McCune, et al. "CLAMP: Practical Prevention of Large-Scale Data Leaks". In: *2013 IEEE Symposium on Security and Privacy* 0 (2009), pp. 154–169. ISSN: 1081-6011. DOI: http://doi.ieeecomputersociety.org/10.1109/SP.2009.21.

[5] Dennis Mirante and Justin Cappos. *Understanding Password Database Compromises*. Tech. rep. Technical Report TR-CSE-2013-02, Department of Computer Science and Engineering Polytechnic Institute of NYU, 2013.

[6] Blake Ives, Kenneth R. Walsh, and Helmut Schneider. "The Domino Effect of Password Reuse". In: *Commun. ACM* 47.4 (Apr. 2004), pp. 75–78. ISSN: 0001-0782. DOI: 10.1145/975817.975820. URL: http://doi.acm.org/10.1145/975817.975820.

[7] C. Castelluccia, A. Chaabane, et al. "When Privacy meets Security: Leveraging personal information for password cracking". In: *ArXiv e-prints* (Apr. 2013). arXiv:1304.6584 [cs.CR].

[8] High-Tech Bridge. *300,000 Compromised Accounts Available on Pastebin: Just the Tip of Cybercrime Iceberg*. Web site. https://www.htbridge.com/news/300_000_compromised_accounts_available_on_pastebin.html (Accessed Jul. 2014). Feb. 2014. URL: https://www.htbridge.com/news/300_000_compromised_accounts_available_on_pastebin.html.

[9] Brian Krebs. *Adobe Breach Impacted At Least 38 Million Users*. Web site. http://krebsonsecurity.com/2013/10/adobe-breach-impacted-at-least-38-million-users/ (Accessed Jul. 2014). Oct. 2013. URL: http://krebsonsecurity.com/2013/10/adobe-breach-impacted-at-least-38-million-users/.

[10] Thorsten Holz, Markus Engelberth, and Felix Freiling. "Computer Security - ESORICS 2009". In: vol. 5789. Lecture Notes in Computer Science. Springer, 2009. Chap. Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones, pp. 1–18.

[11] Brian Krebs. *Adobe To Announce Source Code, Customer Data Breach*. Web site http://krebsonsecurity.com/2013/10/adobe-to-announce-source-code-customer-data-breach/ (Accessed Jul. 2014). Oct. 2013. URL: http://krebsonsecurity.com/2013/10/adobe-to-announce-source-code-customer-data-breach/.

[12] Yacin Nadji, Manos Antonakakis, et al. "Beheading Hydras: Performing Effective Botnet Takedowns". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. CCS '13. New York, NY, USA: ACM, 2013, pp. 121–132.

[13] Brett Stone-Gross, Marco Cova, et al. "Your Botnet is My Botnet: Analysis of a Botnet Takeover". In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS '09. New York, NY, USA: ACM, 2009, pp. 635–647.

[14] Roger Dingledine, Nick Mathewson, and Paul Syverson. "Tor: The Second-Generation Onion Router". In: *Proceedings of the 13th USENIX Security Symposium*. Aug. 2004.

[15] The Tor Project. *Tor: Hidden Service Protocol*. Web Site https://www.torproject.org/docs/hidden-services.html.en (Accessed Jul. 2014). URL: https://www.torproject.org/docs/hidden-services.html.en.

[16] Xiaoyun Wang and Hongbo Yu. "How to Break MD5 and Other Hash Functions". In: *EUROCRYPT*. 2005.

[17] Jens Christian Hillerup. "Cryptanalysis and its Applications to Password Hashing". MA thesis. KTH Information and Communication Technology, 2013.

[18] Arvind Narayanan and Vitaly Shmatikov. "Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. 2005.

[19] Chad Teat and Svetlana Peltsverger. "The Security of Cryptographic Hashes". In: *Proceedings of the 49th Annual Southeast Regional Conference*. Mar. 2011.

Passwords - Divided they Stand, United they Fall

Harshal Tupsamudre, Vijayanand Banahatti and Sachin Lodha

TCS Innovation Labs - TRDDC, India
firstname.lastname@tcs.com

Abstract. It is evident from the breached password databases that in the absence of any composition policy, passwords are created predominantly using lowercase letters or digits or combination of both. Considerable portion of these passwords can be broken offline using dictionary attacks, which exploit the fact that some passwords are more probable than the others. Remaining portion of these passwords can be recovered using a brute-force attack since the passwords do not have sufficient length. To counter these attacks, password policies enforce users to create passwords from a larger search space. One way to achieve this is to enforce the use of a large alphabet set while the other way is to increase the length of passwords. Both of these strategies link the increase in the search space to the increase in the security of passwords. However, we refute the claim that merely increasing the search space results in secure passwords. We hypothesize that with the human generated passwords, the search space will remain highly biased and increasing the search space might just result in a different dictionary for the attacker. We also believe that the system can play a role in changing this biased distribution without resorting to the assignment of random passwords to the users. And therefore, we propose two schemes which ensure that the attacker has to brute-force search the entire search space to break the password database.

1 Introduction

Today, we are living in an internet age, where most of the services such as banking, computing, insurance, utilities are available online. In order to gain access to these web services, the user must prove his identity to the system. This is done through a process called as authentication. There are various ways of authenticating users, however the most common and inexpensive way is to use the *text-based passwords*. In this authentication mechanism, every user selects a unique user-id along with a password. The user-id is assumed to be public but the password should remain secret and difficult to guess. If the password is easy to guess, then depending upon the nature of the service, the compromise of the password can affect the user substantially. Today, almost every website implements the lock out policy or the CAPTCHA mechanism to ensure that the online brute-force attacks do not succeed in a short time. However, with the number of password database breaches increasing [1],[2],[3],[4], the strong threat to the weak passwords comes from the offline attacks. In these attacks, the attacker steals the password database which we assume to be protected by a one-way hash function. Now, since the attacker possesses the hashed password database, there is no restriction on the number of attempts that the attacker might try to break these passwords. And therefore, if the passwords set by the users are easy to guess *i.e.* from an English dictionary or

small enough to brute-force search [5], then the attacker can break such passwords in no time.

Websites force users to choose their passwords from a large search space, so that the brute-force search becomes infeasible. The available search space S_S depends upon two factors; the size of an alphabet set γ from which the password is derived and the length n of the password. Specifically, the search space S_S can be measured using the formula:

$$Search\ Space\ S_S\ = |\gamma|^n \tag{1}$$

Now, the above equation suggests that there are two ways to increase the search space, one is to create passwords of minimum length but using a large alphabet set and the other is to create longer passwords without any restriction of the alphabet set. However, the former strategy seems to be quite popular among the websites. Today, most websites enforce the minimum length restriction in addition to the use of at least one lowercase, uppercase, digit and special character for creating passwords. But, some researchers claim that increasing the length of passwords to say 16 without enforcing the use of the large alphabet set results in more secure and usable passwords [6]. Generally, the strength of the password is measured using entropy, a concept which is more popular in the information theory. If the password p is derived from a uniformly distributed search space S_S, then its entropy E_p [7] is given by:

$$\therefore Entropy\ E_p = log_2(|\gamma|^n)\ bits \tag{2}$$

However, in reality the search space is not uniformly distributed and therefore, the above equation gives upper bound on the strength of the password p. Measuring the exact strength of the password is a non-trivial task and therefore, claiming the security of one scheme over another is non-trivial either. To draw conclusions based upon the survey conducted on few thousand users might be misleading. It is quite evident from the millions of passwords available from the leaked databases that in the absence of any composition rule, the passwords set by users are easy to break (common English words). These passwords are created predominantly using lowercase letters. If the same kind of behaviour is observed even for the longer passwords, the attacker can create a more sophisticated dictionary to mount the attack.

The theoretical search space S_S provides only upper bound on the strength of the password but measuring its actual strength seems to be a difficult task. For most of the purposes, finding the lower bound on the password strength will be sufficient, however, this also seems to be an arduous task. There are different strength meters deployed on different websites that evaluate passwords strength on different criteria [8]. For some strength meters, longer passwords are stronger while for some strength meters the use of a large alphabet set creates stronger passwords. The current strength meters cannot distinguish between the passwords created by the random and the non-random process. The passwords created by the random process are secure against the offline attacks as the only way to break such passwords is to perform the brute-force search. However, such passwords are not considered usable [9].

Contribution. The main objective of this paper is to contradict the claim that

forcing users to derive passwords from a large search space result in secure passwords. For this purpose, we partition the search space into *bins.* Then, we analyse the passwords from the Rockyou database [10] and use the data from the existing surveys [11],[12] to demonstrate that increasing the search space does not result in the creation of secure passwords. We show that enforcing the composition rules results in a non-uniform distribution on the partitions or *bins* which can lead to more sophisticated offline attacks than the trivial brute-force search. We also show that, merely increasing the password length requirement results in longer but weaker passwords.
Finally, we propose two schemes *viz.*, *random bin* scheme and *random phrase* scheme, that utilizes the search space uniformly. The *random bin* scheme aims to create more secure passwords by utilizing every partition or *bin* uniformly while the *random phrase* scheme aims to create more secure passwords by utilizing every word in the *bin* uniformly. The purpose of both these schemes is to cause an exponential increase in the effort of an offline attacker.

Notation. In the rest of the discussion we have used the following notations.
L - An alphabet representing the set $\{a, \ldots, z\}$ of lowercase letters.
U - An alphabet representing the set $\{A, \ldots, Z\}$ of uppercase letters.
D - An alphabet representing the set $\{0, \ldots, 9\}$ of decimal digits.
S - An alphabet representing the set of 33 special symbols such as $, \&, \#, \{, \}$ and so on.
+ - denotes 1 or more occurrences of the alphabet.
***** - denotes 0 or more occurrences of the alphabet.
? - denotes 0 or 1 occurrence of the alphabet.
[i,j] - denotes at least i and at most j occurrences of the alphabet, where $0 \leq i < j$.
$|\boldsymbol{\alpha}|$ - represents the number of elements in a set α.

Organization. The organization of this paper is as follows. In section 2, we provide a brief explanation of the related work. Then, in the subsequent sections we describe our contribution in detail. In section 3, we partition the search space into *bins.* In section 4, we show that increasing the search space either by forcing the use of the large alphabet set or by increasing the length of the passwords might not achieve the expected results. In section 5, we propose two schemes that increases the effort of an offline attacker exponentially. In section 6, we discuss the implications of these schemes on the future. Finally, in section 7, we conclude the paper.

2 Related Work

User habits in creating passwords have been studied since 1979 [5]. The authors of [5] analysed 3289 passwords and found that 86% of them are weak either due to their prevalence in the dictionary or due to their short length and containing only lowercase letters or digits. Later in 1990, Klein [13] successfully broke 25% of the passwords in use, on the unix system using the brute-force attack. In 1999, Moshe and William [14] based on their survey of 997 participants found that 80% of the passwords were derived only using lowercase and uppercase letters. In 2006, the leak of 34,000 Myspace passwords revealed that passwords such as “password”are popular choices among the users [15]. In 2007, Florencio and Herley

[7] studied passwords of nearly 5 million users and found that most of the passwords are created using either lowercase letters or digits. The leak of 32 million Rockyou passwords in 2009 revealed that 0.9% of these passwords is a numeric string "123456"[16]. These data suggest that there is a non-uniform distribution on the search space and therefore, the resulting passwords are not secure.

The presence of composition rules is believed to create secure passwords [17] while [6] and [18] suggest that longer passwords are no less secure than those created under composition rules. There are also studies which suggest that the presence of strength meters result in stronger passwords [19]. However, all these results are based upon the survey of not more than few thousand users and there is no real data available to study passwords created in the presence of composition rules or strength meters. Moreover, determining the actual strength of the password is not an easy task. NIST provides the guidelines for measuring the strength of the password [20], however this approach is not based on the large empirical data.

In the following sections, we describe our contribution in detail. First, we partition the search space into *bins* and show that increasing the search space does not guarantee the creation of secure passwords. Subsequently, we describe two schemes which result in the creation of secure passwords.

3 Partitioning the Search Space into Bins

Consider the alphabet set $\sigma = \{L, U, D, S\}$ consisting of 4 alphabets. We refer to n length strings derived from the alphabet set σ as *password bins* or just *bins*, e.g. L^8, U^1L^7, $S^1U^1L^5D^1$ are all 8 length bins. The collection of all n length bins forms a bin space. Therefore, the size of the bin space is 4^n. Basically, every bin represents a class of passwords, *e.g.* the bin L^8 represents 8 length passwords composed entirely of lowercase letters while the bin U^1L^7 represents 8 length passwords that begin with an uppercase letter followed by 7 lowercase letters. Every such bin depending upon its alphabetic composition has a certain capacity associated with it. This capacity is defined as the number of passwords represented by the bin. For instance, the capacity of the bin L^8 composed of all lowercase letters is $|L|^8 = 26^8$. Since the collection of all passwords form the search space for an attacker, the size of the search space is the sum of capacity of all bins.

$$\begin{aligned}\sum_{i=1}^{4^n} C_i &= (|L| + |U| + |D| + |S|)^n \\ &= (26 + 26 + 10 + 33)^n \\ &= 95^n\end{aligned}$$

where C_i is the capacity of the ith bin. The number and capacity of bins increases exponentially with the increase in the length.

Studies [5],[14],[7] suggest that in the absence of any composition rule, users create their passwords preferably from the bins composed entirely of lowercase letters (L^+) or digits (D^+). Thus, if no restriction is imposed during the password creation, very few bins are used extensively for deriving the passwords which results in a non-uniform distribution on the bin space. To identify the bins that are more

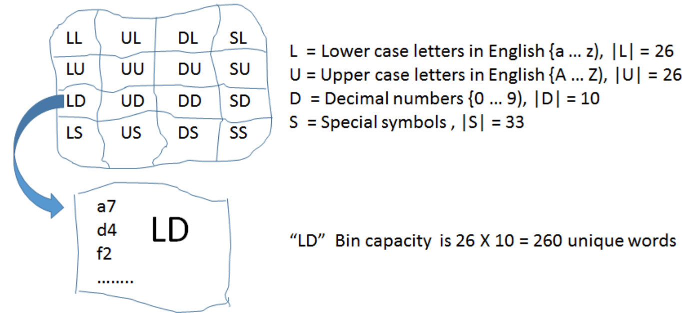


Fig. 1: Illustration of a Bin Space. The number of $n = 2$ length bins derived using the alphabet set $\sigma = \{L, U, D, S\}$ is $4^n = 4^2 = 16$.

frequently used for the password creation, we studied nearly 14 million unique passwords in the Rockyou database [10] that were breached in 2009. However, upon analysis, we found that nearly 13 million Rockyou passwords which comprise 93% of the breached database, have maximum length of 12. Therefore, for the current purpose, we focus our attention on these passwords only.

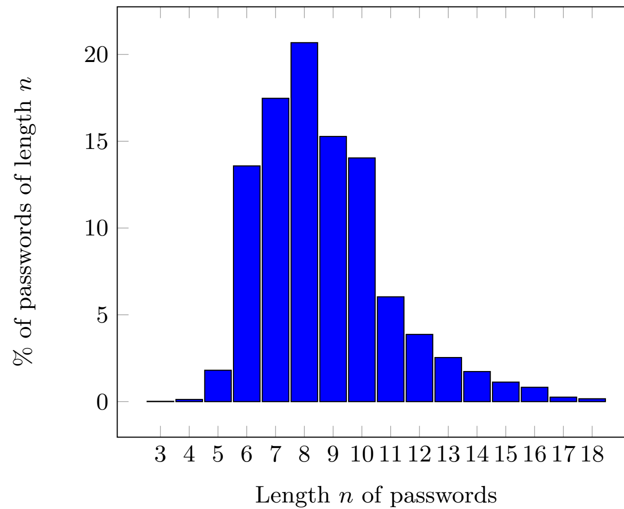


Fig. 2: Length-wise distribution of 14 million unique passwords found in the Rockyou Database.

Analysing 13 million unique passwords revealed the form of the popular bins which are enlisted in the Table 1 along with their descriptions. The most popular bins are of the form L^+D^+, that begin with at least one lowercase letter and end with at least one digit. Nearly 31.32% of the passwords were derived from these bins. Counting the number of bins of any particular form depends upon its composition and the password length n. For instance, the number of bins of the form L^+D^+ and length 3 is 2 (LD^2,L^2D). Table 1 also enumerates the count of the

most popular bins found in the Rockyou database as the function of the password length n.

Table 1: Analysing 13 million passwords of length $n \leq 12$ in the Rockyou database reveals the most popular *bins*.

Popular *bins*	Bin Description	Number of such bins of length n	% of passwords found
U^+	composed entirely of uppercase letters	1	1.51
U^+D^+	starting with at least 1 uppercase letter and ending with at least 1 digit	$n-1$	2.20
D^+L^+	starting with at least 1 digit and ending with at least 1 lowercase letter	$n-1$	3.93
D^+	composed entirely of digits	1	16.36
L^+	composed entirely of lowercase letters	1	24.30
L^+D^+	starting with at least 1 lowercase letter and ending with at least 1 digit	$n-1$	31.32

The total number of popular bins that constitutes nearly 80% of 13 million Rockyou passwords can be found by adding the entries in the third column of the Table 1 while varying the password length n from 1 to 12.

$$\begin{aligned}\sum_{n=1}^{12} 1+n-1+n-1+1+1+n-1 &= \sum_{n=1}^{12} 3\cdot n \\ &= 3\cdot\sum_{n=1}^{12} n \\ &= 234 \end{aligned} \tag{3}$$

The number of bins having length $n \leq 12$ is $\sum_{n=1}^{12} 4^n$. However, analysing the 13 million unique passwords in the Rockyou database revealed that nearly 80% of these passwords can be found in a tiny fraction $234/\sum_{n=1}^{12} 4^n$ of the total bins. This clearly indicates that the bin space is highly biased. Now, the offline attacker can exploit the fact that few bins are extremely popular and as a result, searching the popular bins will break the protected password database. Given the fact that 56-bit DES key can be searched exhaustively, if the Rockyou password database was protected with a one way hash function, the motivated attacker could have performed the brute-force search on 234 popular bins of the length $n \leq 12$ and recovered nearly 80% of passwords. The capacity of any of the 234 bins does not exceed $log_2(26^{12}) \approx 56$ bits. The actual search space is enormous $log_2(95^{12}) \approx 79$ *bits*. However, the attacker can take advantage of the fact that only a fraction of the bins are highly probable and skip the search of the unpopular bins. Also, there is no need to mount the dictionary attack as the brute-force search of few popular

bins is sufficient for the attack to succeed.

Remark 1 : *The entire search space can be partitioned into bins. When no restriction is imposed during the password creation, the bin space becomes highly biased and as a consequence a tiny fraction of bins become very popular. Therefore, the effort of the offline attacker is reduced drastically as the considerable portion of the password database can be recovered by targeting only these popular bins.*

4 Increasing the Search Space

In this section, we show that increasing the search space still results in a non-uniform distribution on the bin space, which can be exploited by the offline attacker. As mentioned previously, the search space can be increased either by enforcing the composition rules or by enforcing the minimum length restriction during the password creation. We discuss both the cases.

4.1 Effect of Enforcing the Composition Rules

The composition rules deny the use of bins composed of only one alphabet and enforce users to choose the bins composed of at least 2 or 3 alphabets. In this case, the search space is too large (95^n) to carry out the brute-force search. However, merely increasing the search space does not imply that all available bins are used uniformly. Surveys [11],[12] and breached databases suggest that the bins of the form $\{S, D\}^p U^1 L^{n-2p-1} \{S, D\}^p$, $p > 0$, that begin with at least one digit or symbol followed by at least one uppercase letter and then lowercase letters and ending with at least one digit or symbol are more popular. We analysed 3 types of minimum 8 length passwords from the Rockyou database *viz.*; alpha-numeric, alpha-symbolic and the passwords composed of at least one letter from each alphabet L, U, D and S. These passwords best reflect the ones that are created due to the compostion rules widely in use today. Analysing these Rockyou passwords provides more insights into the bins that become highly probable in the presence of the composition rules.

It is evident from the Table 2. that enforcing composition rules might not achieve security even against the brute-force attack. For instance, analysing the alpha-numeric passwords in the Rockyou database reveals that nearly 56% of these passwords are derived from the bins $U^1 L^+ D^+$ that begin with an uppercase letter followed by lowercase letters and ending with digits. The number of such bins are few and depends upon the length. In general, if the length of the bin is n, then the number of bins that begin with an uppercase letter and end with digits is $n - 2$. Since, the number of available bins is 4^n, the considerable portion (56%) of Alpha-Numeric passwords fall in a very tiny fraction ($(n - 2)/4^n$) of bins. Moreover, if n is not large *i.e.* $n \leq 12$ which typically is, then the popular bins can be easily brute-force searched for passwords. Nearly 93% of 14 million unique passwords in the Rockyou database have length at most 12.

Now, if the attacker can identify such high probable bins then only those need to be searched, instead of searching the entire search space. The non-uniform distribution on the bins suggests that enforcing composition rules does not result in a proper utilization of the available search space of 95^n. Today, most of the organizations need users to set at least 8 length passwords derived from the set σ. If

Table 2: Analysing the composition rules compliant passwords of length $n \geq 8$ in the Rockyou database reveals the most popular *bins*.

Password Type	Number of passwords	Popular *bins*	Bin Description	Number of n length bins	% of Rock-you pass-words
alpha-numeric $\{L, U, D\}^+$	293932	$U^1L^+D^+$	starting with 1 uppercase letter followed by 1 or more lowercase letters and ending with 1 or more digits	$n-2$	56
alpha-symbolic $\{L, U, S\}^+$	36730	$U^?L^*S^1U^?L^*$	consisting of exactly 1 symbol and starting and ending with at most 1 uppercase letter followed by 0 or more occurrences of lowercase letter	$4 \cdot n$	41.5
All $\{L, U, D, S\}^+$	44726	$UL^+D^*S^1D^*$	consisting of exactly 1 symbol sandwiched between the digits and starting with 1 uppercase letter followed by at 1 or more occurences of lowercase letter	$2 \cdot n$	30

the bin of length 9 and of the form $U^1L^5S^1D^2$ becomes most popular, then to recover the majority of passwords, the offline attacker has to brute-force search only $26 * 26^5 * 33 * 10^2 \approx 2^{39.5}$ combinations which is quite feasible with the currently available computing power. With this kind of distribution, the attacker does not have to brute-force the relatively enormous search space of $95^9 \approx 2^{59}$ and hence, the minimum 8 length requirement is not enough.

Remark 2 : *The passwords created due to the composition rules might not be secure even against the brute-force attack. Again, this is because of the non-uniform distribution on the bin space. Some bins are more frequently used for deriving the passwords. For instance, if users are enforced to create alpha-numeric passwords, the analysis of Rockyou database suggests that bins of the form* $U^1L^+D^+$ *can become more popular. Therefore, the offline attacker without resorting to any dictionary attack can break considerable fraction of the protected password database by searching only the popular bins. This defeats the purpose of enforcing the composition rules.*

4.2 Effect of Increasing the Password Length

To render the brute-force search infeasible, passwords should be created from the large capacity bins. Using the large capacity bins implies increasing the password length n. Another advantage of increasing the password length is that we can get rid of the restriction of using a large alphabet set for deriving the passwords. This is because increasing the length results in an exponential increase in the capacity

of the bin which makes the brute-force search infeasible. In the absence of any alphabet set restriction, the bins of the form L^n are more preferred for creating the passwords [5],[14],[7]. If the search space with 75 bits entropy is considered as infeasible to mount a brute-force attack then the minimum length restriction should be at least $n = 16$. This is because, the capacity of the bin L^{16} is $|L|^{16} = 26^{16}$ and provides $log_2(26^{16}) \approx 75$ bits of entropy. However, there is not only a non-uniform distribution on the bins but even words in the bin are highly biased. This behaviour is exploited by the attacker in the form of sophisticated dictionaries. The words in the bins that are more probable are the common English words, misspelled English words or common English phrases.

Table 3: Breaking L^n passwords in Rockyou database v/s Random database using the phrase dictionary of size 2^{56}.

Password length n	Number of L^n passwords in Rockyou database	% of Rockyou passwords broken using the phrase dictionary	% broken if the Rockyou passwords were random
13	128695	79.36	$100 \cdot 2^{56}/26^{13} \approx 2.9$
14	86632	68.26	$100 \cdot 2^{56}/26^{14} \approx 0.11$
15	59796	58.12	$100 \cdot 2^{56}/26^{15} \approx 0.004$
16	36416	47.82	$100 \cdot 2^{56}/26^{16} \approx 0.00016$
17	14138	39.5	$100 \cdot 2^{56}/26^{17} \approx 0.0000063$
18	8970	32.5	$100 \cdot 2^{56}/26^{18} \approx 0.0000002$

Analysing the longer passwords in the Rockyou database revealed that the sizeable chunk of these passwords are merely concatenation of most common passwords and most common words used in the English dictionary. To verify this, we selected nearly $16000 \approx 2^{14}$ words comprising the common passwords, common names, common words and analysed passwords of the form L^n, where $13 \leq n \leq 18$. We created a phrase dictionary with the help of the $16000 \approx 2^{14}$ words and observed that by exploring over 2^{56} search space (phrases of at most 4 words), a considerable portion of longer passwords in the Rockyou database can be recovered. However, if passwords were derived randomly from an uniform distribution, then a very tiny fraction of passwords would have been broken due to our phrase dictionary. For instance, the number of Rockyou passwords derived from the bin L^{13} that can be broken by exploring a search space of 2^{56} is nearly 79% while the number of random passwords derived from the bin L^{13} that can be broken is nearly $100 * 2^{56}/26^{13} \approx 2.9 \ll 79.36$. The results are shown in the Table 3. We achieved the similar results for passwords of other lengths too. These results cleary indicate that the words in the bin are also non-uniformly distributed, which drastically reduces the search space for an offline attacker. Therefore, increasing the capacity of bins might not solve the password security problem. In other words, longer passwords might not imply secure passwords. If users are forced to create longer passwords and the same behaviour is observed then the motivated attacker can build more sophisticated dictionary and with the available computing power

can mount more serious attack than ours.

Remark 3 : *If users are forced to create longer passwords without any restriction of a large alphabet set, then bins of the form L^n can become more popular. Moreover, if the resulting passwords are concatenation of common English words as evident from the analysis of L^n passwords in the Rockyou database, the offline attacker can construct a dictionary to exploit this bias. As a result, the search space for the attacker is drastically reduced due to the non-uniformly distributed words in the high probable bins. Therefore, merely increasing the password length might not result in the secure passwords.*

5 Application of Bins

As concluded earlier, increasing the search space does not result in the creation of secure passwords. The non-uniform distribution on the bins enables the offline attacker to mount a brute-force attack on few popular bins to recover the majority of passwords. However, if the capacity of popular bins is large enough to make the brute-force search infeasible, then the attacker can resort to sophisticated dictionary attack which exploits the fact that words in the bins are also non-uniformly distributed. In this section, we present two schemes that counter these attacks and result in an exponential increase in the effort of the offline attacker.

5.1 Random Bin Scheme

With the human generated passwords, the non-uniform distribution on the search space cannot be avoided. Therefore, to achieve the desired security, *system assigned passwords* seems to be the best option. These passwords can be seen as generated using the following steps.

1. Randomly select the length n, $n_{min} \leq n$ *e.g.* $n = 12$.
2. Randomly select the bin β of length n, *e.g.* $\beta = L^2D^3L^1S^4U^2$.
3. Randomly select a word in the bin β, *e.g.* ke932c-%?'LQ

However, *system assigned passwords* affects usability [9]. Users have no control over creating their passwords and the resulting passwords are difficult to remember. Therefore, we propose a scheme that provides users with some control over their passwords creation without significantly comprimising the security. In this scheme, we allow the system to decide upon the first two steps. In other words, we allow system to randomly assign the bin β of length n to the users. Now, every user selects the password from the assigned bin. We call this as *random bin* scheme. This scheme ensures that the bin space is uniformly distributed and the brute-force search of few bins does not break the entire password database. The effort of the offline attacker is increased considerably as every bin needs to be searched. However, this scheme provides no guarantee over the distribution of words in the bin, as they are chosen by users. The another advantage of this scheme is that the minimum length of the password requried to thwart the brute-force attack can be calculated precisely. For this purpose, we assume that passwords are created using 95 characters (4 alphabets L, U, D, S). Now, if the entropy $E_{desired}$ is considered to be secure against the brute-force attack, then the minimum length n_{min} of

passwords should be:

$$\begin{aligned} E_{desired} &= log_2(95^{n_{min}}) \\ &= n_{min} \cdot log_2(95) \\ \therefore n_{min} &= E_{desired}/log_2(95) \end{aligned} \tag{4}$$

if $E_{desired} = 75$ bits, then $n_{min} \approx 12$. In this case, there are $4^{n_{min}} = 4^{12}$ bins available which are randomly assigned to the users. Therefore, the attacker has to brute-force search the entire search space of $95^{n_{min}}$ to break the resulting password database.

Alternatively, the system can randomly assign a unique bin to every user. This is mere a variant of *random bin* scheme and we call it as a *unique bin per user* scheme. Assuming that bins are derived from the alphabet set $\sigma = \{L, U, D, S\}$, the length of the password should be at least :

$$\begin{aligned} n_{min} &= log_{|\sigma|}(\textit{number of users}) \\ &= log_4(\textit{number of users}) \end{aligned} \tag{5}$$

Thus, if passwords are derived using 4 alphabets $\sigma = \{L, U, D, S\}$ and the *number of users* is $2^{20} \approx 1$ million then the minimum length of the password should be $log_4(2^{20}) = 10$. But, if passwords are derived using only 2 alphabets L and D, then the minimum length requirement should be $log_2(2^{20}) = 20$ (Fig. 3). In cases, where the *number of users* are few, the system can resort to the *random bin* scheme and compute the n_{min} by fixing the desired entropy $E_{desired}$ to counter the brute force attack.

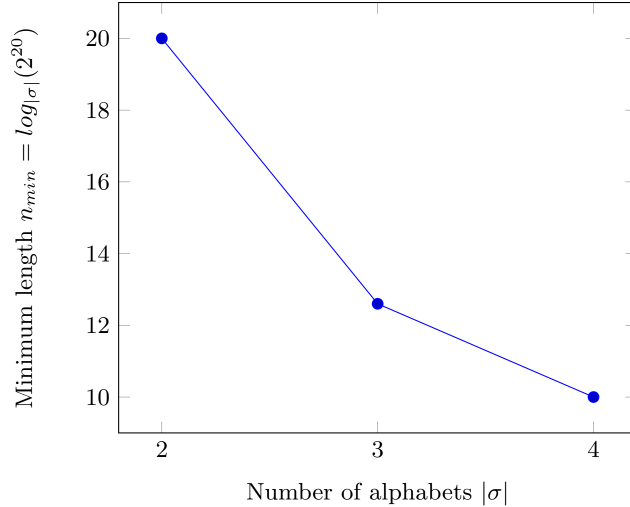


Fig. 3: Using the *unique bin per user* scheme to determine the minimum password length n_{min} for $2^{20} \approx 1$ million users by varying the size of the alphabet set σ.

Strength. The uniform distribution on the bins ensure that the entropy E_{min}

of the password database in a *unique bin per user* scheme is at least:

$$E_{min} = log_2(|\sigma|^{n_{min}})$$
$$= log_2(4^{n_{min}}) \quad (6)$$

$|\sigma|^{n_{min}}$ is the number of possible bins of length n_{min} derived from the alphabet set σ. This scheme ensures that the bins are uniformly distributed and assigned uniquely to every user. The advantage of such system assigned schemes is that the password reuse can be prevented, the password strength can be measured and with the lock out policies in place, online attacks can be easily thwarted. Moreover, the brute-force attack on few bins will not break the password database as the bins are equally likely. Now, offline attackers have to target all $4^{n_{min}}$ bins and hence, the entropy E_{max} of a resuting password database against the brute-force attack is :

$$E_{max} = log_2(95^{n_{min}}) \quad (7)$$

Thus, when bins are randomly assigned to the users, all bins become equally likely. As a result, there is no bias to exploit and therefore, to break the password database, attackers have to brute-force search the entire search space.

Challenges. In the *random bin* scheme, a bin of certain length is randomly assigned to the user. Creating passwords complying with the assigned bins can be cumbersome and might annoy the users. Further, some bins are more usable than the others, e.g. the password bin $U^1L^8D^5S^2$ can be considered as more usable than $S^2L^1D^3L^2S^1U^4D^2L^1$. In such cases, usability will not be any better than the *system assigned passwords*. Moreover, some bins that are more user friendly, might result in weaker passwords. For instance, with the bin $S^{12}L^4$ of length 16, the password set by an user could be $\#^{12}a^4$. Therefore, in such cases, the strength of passwords will be merely due to the randomness of bins and not much due to the letters used in passwords. This fact can be exploited by the dictionary attack.

5.2 Random Phrase Scheme

There is yet another way to increase the security of the password database. The system can allow a non-uniform distribution on the bin space, however, the words in those high probable bins should be uniformly distributed. Now, even if the attacker learns about the high probable bins, any attempt to mount the dictionary attack will be futile as there will not be any bias to exploit. The only way to break the database is to perform the brute-force search on the high probable bins. The password length is chosen such that the capacity of bins makes the brute-force attack infeasible. In the absence of any alphabet set restrictions, the bins of the form L^n become more popular [5],[14],[7] and therefore, we assume that passwords are created using the alphabet L *i.e.* $|L| = 26$ lowercase letters. Again, assuming that the $E_{desired}$ bits of entropy is strong enough to resist the brute-force attack, the minimum length n_{min} of L^n bins should be :

$$E_{desired} = log_2(26^{n_{min}})$$
$$= n_{min} \cdot log_2(26)$$
$$\therefore n_{min} = E_{desired}/log_2(26) \quad (8)$$

If $E_{desired} = 75$ bits, then $n_{min} \approx 16$. In this case, words from the bins L^n, $n \geq n_{min} = 16$, are randomly assigned to the users. Therefore, the attacker has to brute-force search the entire search space of $26^{n_{min}}$ to break the resulting password database.

Strength. The uniform distribution on the words in $L^{n_{min}}$ bins ensures that the entropy of the password database is :

$$\begin{aligned} E_{min} &= log_2(|L|^{n_{min}}) \\ &= log_2(|26|^{n_{min}}) \end{aligned} \tag{9}$$

Thus, when words from a popular bins are randomly assigned to the users, all words become equally likely. Since there is no bias to exploit, attackers have to brute-force search the popular bins to break the password database. However, the capacity of popular bins is large enough to resist the brute-force attack.

Challenges. Finding such usable bins wherein words are easy to remember is a daunting task. Generating system assigned pass-phrase composed of English words can be one way of generating such bins. However, all words in the bin are not pass-phrases. Therefore, we choose a alphabet set ϵ_{words} consisting of English words, from which the pass-phrase can be derived. If $|\epsilon_{words}| = 16000 \approx 2^{14}$ English words are used to create pass-phrases, then to achieve 75 bits of entropy, every user should remember the pass-phrase made up of at least $75/log_2(2^{14}) = 75/14 \approx 5$ words. Remembering 5 words can be very demanding for the users and thus, assigning the random pass-phrases can lead to a cognitive burden.

In the following section, we discuss the further consequences of partitioning the search space into bins and the implications of proposed schemes on the future.

6 Discussion

If we assume that passwords are derived from n length bins composed of 4 alphabets $\sigma = \{L, U, D, S\}$ and if θ^n denotes the average capacity of each bin, then the size of search space S_S and the corresponding entropy E_p is:

$$\begin{aligned} &Search\ Space\ S_S = \theta^n \cdot 4^n \\ &\therefore Entropy\ E_p = n \cdot log_2(\theta) + 2 \cdot n \end{aligned} \tag{10}$$

However, this search space, provides a very loose upper bound on the password strength. Analysis of the Rockyou database revealed that the bin space is highly biased. The number of bins increases exponentially with the increase in the password length n, however, the number of bins that are utilized increases only linearly (Fig. 4). These utilized bins form a tiny fraction which can be exploited by the brute-force attack. The uneven utilization of bins enables us to compute much tighter upper bound on the password strength. If $c \cdot n$, where c is a constant, denotes the number of bins used for deriving n length passwords, then the size of the search space that is actually utilized $S_{utilized}$ and the corresponding entropy $E_{utilized}$ is given by:

$$\begin{aligned} &Search\ Space\ S_{utilized} = \theta^n \cdot c \cdot n \\ &\therefore Entropy\ E_{utilized} = n \cdot log_2(\theta) + log_2(c \cdot n) \end{aligned} \tag{11}$$

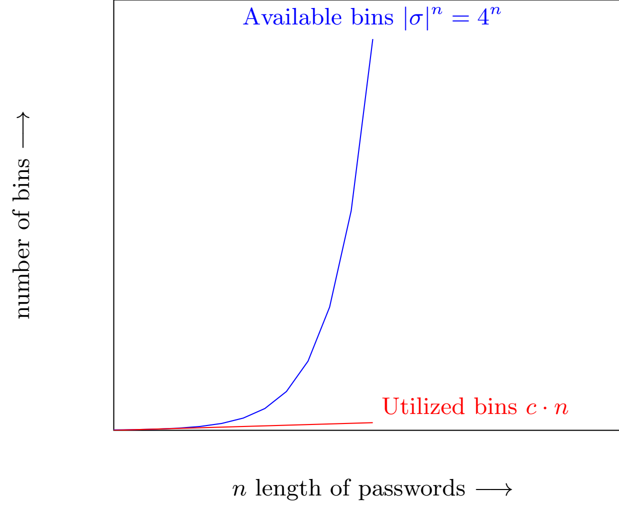


Fig. 4: Available Bin Space v/s Actual Bin Utilization in the Rockyou Database as a function of password length n.

Hence, by targeting only the popular bins, the effort of an attacker is reduced by at least $E_p - E_{utilized}$ bits,

$$E_p - E_{utilized} = 2 \cdot n - log_2(c \cdot n) \tag{12}$$

And therefore, the fraction F of the search space that can be skipped by the attacker is at least:

$$\begin{aligned} Fraction\ F &= (E_p - E_{utilized})/E_p \\ &= (2 \cdot n - log_2(c \cdot n))/(n \cdot log_2(\theta) + 2 \cdot n) \end{aligned} \tag{13}$$

e.g. if $\theta = 26$, $n = 12$ and $c = 5$, then $E_p \approx 12 \cdot 4.5 + 12 \cdot 2 = 78$ bits and $E_{utilized} \approx 12 \cdot 4.5 + 6 = 60$ bits. In this case, the effort is reduced by at least $78 - 60 = 18$ bits and therefore, the fraction of the search space that can be skipped is $18/78 \approx 0.23$. Today, the search of 78 bits search space is nearly infeasible but searching 60 bits space is not an impossible task. This indicates that just increasing the search space is not enough.

Also, until now determining the minimum length requirement of the password was not an exact science. However, by using the concept of a bin, we can not only determine the minimum length requirement but also achieve the desired level of security without using the *system assigned random* passwords. More specifically, the *random bin* and the *random phrase* schemes enable us to address these issues by precisely computing the minimum length requriement of the passwords for any alphabet set σ and enforcing the uniform distribution on the search space and therefore, eliminating the difference between the actual search space E_p and the utilized search space $E_{utilized}$.

The passwords yet have to play a more crucial role in achieving the desired level of security by thwarting the unauthorised access. With the Cisco, predicting nearly 50 *billion* $\approx 2^{35.5}$ internet connected devices by 2020 [21] and with no alternative usable and secure authentication mechanism in sight, the importance of the text-based passwords will increase tremendously. Suppose that a random password of

length 10 is used for protecting every device. Since, there are only $|\sigma|^{10} = 4^{10}$ bins of length 10, searching any bin will reveal nearly $2^{35.5}/4^{10} \approx 2^{15.5} = 46340$ passwords. This suggests that in the future even if the passwords are derived randomly, the number of passwords that can be recovered by searching any bin can prove fatal. However, with a *unique bin per user* scheme or in the correct context it is more appropriate to call it as a *unique bin per device* scheme, the minimum length of the passwords to prevent such attacks can be percisely calculated. The minimum length of the passwords required to protect 50 *billion* $\approx 2^{35.5}$ devices against the brute-force attack is $n_{min} = log_4(number\ of\ devices) = log_4(35.5) \approx 18$.

7 Conclusion

Most websites prefer the use of composition rules to create secure passwords. However, some researchers claim that just increasing the password length create more secure passwords. We hypothesized that both these approaches do not ensure the security of the resulting passwords. We partitioned the search space into *bins* and showed that some bins are more probable than the others. These popular bins can be exhaustively searched to recover the considerable portion of passwords without searching the entire available space. Futher, we showed that the longer passwords are mostly derived by concatenating common English words and can be broken using the phrase dictionary. To circumvent these attacks, we proposed *random bin* and *random phrase* schemes which ensure that either the bins are uniformly distributed or words in bins are uniformly distributed. Further, these schemes allow us to determine the minimum length requirement of passwords as a function of *number of users* and alphabet set. Because of the *random bin* scheme, every bin becomes equally likely and hence there are no more popular bins. Now, the attacker cannot expect to get the majority of passwords by a brute-force search of few bins which is possible today with the composition rules in place. The attacker has to search the entire search space to break the password database. In the *random phrase* scheme, every word in a bin has equal probability of being a password which renders the dictionary attack futile whereas the brute-force attack is countered by choosing the bin of a large capacity. Both these schemes ensures that the effort of the attacker is increased exponentially and the resulting password database is secure against the brute-force attack.

References

1. CNBC. 5 million gmail passwords leaked: Is yours one of them. http://www.cnbc.com/id/101989542, accessed on 21 October 2014.
2. BBC. Adobe hack: At least 38 million accounts breached. http://www.bbc.com/news/technology-24740873, accessed on 21 October 2014.
3. CBS NEWS. 6.5 million linkedin passwords reportedly leaked on russian hacker site. http://www.cbsnews.com/news/65-million-linkedin-passwords-reportedly-leaked-on-russian-hacker-site/, accessed on 21 October 2014.
4. The Telegraph. Playstation hack: Sony users urged to change passwords. http://www.telegraph.co.uk/technology/sony/8478404/PlayStation-hack-Sony-users-urged-to-change-passwords.html, accessed on 21 October 2014.
5. Robert Morris, Robert Morris, Ken Thompson, and Ken Thompson. Password security: A case history. *Communications of the ACM*, 22:594–597, 1979.

6. Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. Of passwords and people: measuring the effect of password-composition policies. In *Proceedings of the 2011 Annual ACM Conference on Human Factors in Computing Systems*, CHI '11, pages 2595–2604, New York, NY, USA, 2011. ACM.
7. Dinei Florencio and Cormac Herley. A large-scale study of web password habits. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 657–666, New York, NY, USA, 2007. ACM.
8. Xavier de Carné de Carnavalet and Mohammad Mannan. From very weak to very strong: Analyzing password-strength meters. In *Network and Distributed System Security (NDSS) Symposium 2014*. Internet Society, February 2014.
9. Michael D. Leonhard and V. N. Venkatakrishnan. A comparative study of three random password generators. In *EIT'07: Proc. 2007 IEEE International Conference on Electro/Information Technology*, 2007.
10. https://wiki.skullsecurity.org/Passwords, accessed on 21 October 2014.
11. Richard Shay, Saranga Komanduri, Patrick Gage Kelley, Pedro Giovanni Leon, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Encountering stronger password requirements: User attitudes and behaviors. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, SOUPS '10, pages 2:1–2:20, New York, NY, USA, 2010. ACM.
12. Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and Xiaofeng Wang. The tangled web of password reuse. In *Network and Distributed System Security (NDSS) Symposium 2014*. Internet Society, February 2014.
13. Daniel V. Klein. "foiling the cracker": A survey of, and improvements to, password security. Usenix Security Workshop, 1990.
14. Moshe Zviran and William J. Haga. Password security: An empirical study. *J. of Management Information Systems*, 15(4):161–186, 1999.
15. Bruce Schneier. Schneier on security: Real-world passwords. http://www.schneier.com/blog/archives/206/12/realworld_passw.html, accessed on 21 October 2014.
16. http://www.imperva.com/docs/WP_Consumer_Password_worst_Practices.pdf, accessed on 21 October 2014.
17. Wayne C. Summers and Edward Bosworth. Password policy: The good, the bad, and the ugly. In *Proceedings of the Winter International Synposium on Information and Communication Technologies*, WISICT '04, pages 1–6. Trinity College Dublin, 2004.
18. Robert W. Proctor, Mei-Ching Lien, Kim-Phuong L. Vu, E.Eugene Schultz, and Gavriel Salvendy. Improving computer security for authentication of users: Influence of proactive password restrictions. *Behavior Research Methods, Instruments, & Computers*, 34(2):163–169, 2002.
19. Serge Egelman, Andreas Sotirakopoulos, Ildar Muslukhov, Konstantin Beznosov, and Cormac Herley. Does my password go up to eleven?: The impact of password meters on password selection. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2379–2388, New York, NY, USA, 2013. ACM.
20. NIST. Electronic authentication guideline. http://csrc.nist.gov/publications/nistpubs/800-63-1/SP-800-63-1.pdf, accessed on 21 October 2014.
21. Cisco. The internet of things. http://share.cisco.com/internet-of-things.html, accessed on 21 October 2014.

Overview of the Candidates for the Password Hashing Competition

And their Resistance against Garbage-Collector Attacks

Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel
<first name>.<last name>@uni-weimar.de

Bauhaus-Universität Weimar

Abstract. In this work we provide an overview of the candidates of the Password Hashing Competition (PHC) regarding to their functionality, e.g., client-independent update and server relief, their security, e.g., memory-hardness and side-channel resistance, and its general properties, e.g., memory usage and flexibility of the underlying primitives. Furthermore, we formally introduce two kinds of attacks, called Garbage-Collector and Weak Garbage-Collector Attack, exploiting the memory management of a candidate. Note that we consider all candidates which are not yet withdrawn from the competition.

Keywords: Password Hashing Competition, Overview, Garbage-Collector Attacks

1 Introduction

Typical adversaries against password-hashing algorithms (also called password scramblers) try plenty of password candidates in parallel, which becomes a lot more costly if they need a huge amount of memory for each candidate. On the other hand, the defender (the honest party) will only compute a single hash, and the memory-cost parameters should be chosen such that the required amount of memory is easily available to the defender.
But, memory-demanding password scrambling may also provide a completely new attack opportunity for an adversary, exploiting the handling of the target's machine memory. We introduce the two following attack models: (1) Garbage-Collector Attacks, where an adversary has access to the internal memory of the target's machine **after** the password scrambler terminated; and (2) Weak Garbage-Collector Attacks, where the password itself (or a value derived from the password using an efficient function) is written to the internal memory and almost never overwritten during the runtime of the password scrambler. If a password scrambler is vulnerable in either one of the attack models, it is likely to significantly reduce the effort for testing a password candidate.

Up to now, there exist two basic strategies of how to design a memory-demanding password scrambler:

Type-A: Allocating a huge amount of memory which is rarely overwritten.
Type-B: Allocating a reasonable amount of memory which is overwritten multiple times.

The primary goal of the former type of algorithms is to increase the cost of dedicated password-cracking hardware, i.e., FPGAs and ASICs. However, algorithms following this approach do not provide high resistance against garbage-collector attacks, which are formally introduced in this work. The main goal of the second approach is to thwart GPU-based attacks by forcing a high amount of cache misses during the computation of the password hash. Naturally, algorithms following this approach provide some kind of built-in robustness against garbage-collector attacks.

Remark 1. For our theoretic consideration of the proposed attacks, we assume a natural implementation of the algorithms, e.g., that some possible mentioned overwriting of the internal state **after** the invocation of an algorithm is neglected due to optimization.

2 (Weak) Garbage-Collector Attacks and their Application to ROMix and `scrypt`

In this section we first provide a definition of our attack models, i.e., the Garbage-Collector (GC) attack and the Weak Garbage-Collector (WGC) attack. For illustration, we first show that ROMix (the core of `scrypt` [19]) is vulnerable against a GC attack (this was already shown in [11], but without a formal definition of the GC attack), and second, we show that `scrypt` is also vulnerable against a WGC attack.

2.1 The (Weak) Garbage-Collector Attack

The basic idea of these attacks is to exploit the memory management of password scramblers based on the handling of the internal state or some single password-dependent value. More detailed, the goal of an adversary is to find a valid password candidate based on some knowledge gained from observing the memory used by an algorithm, whereas the test for validity of the candidate requires significantly less time/memory in comparison to the original algorithm. Next, we formally define the term Garbage-Collector Attack.

Definition 1 (Garbage-Collector Attack). *Let $PS_G(\cdot)$ be a memory-consuming password scrambler that depends on a memory-cost parameter G and let Q be a positive constant. Furthermore, let v denote the internal state of $PS_G(\cdot)$ after its termination. Let $\mathcal{A}$ be a computationally unbounded but always halting adversary conducting a garbage-collector attack. We say that $\mathcal{A}$ is successful if some knowledge about v reduces the runtime of $\mathcal{A}$ for testing a password candidate x from $\mathcal{O}(PS_G(x))$ to $\mathcal{O}(f(x))$ with $\mathcal{O}(f(x)) \lll \mathcal{O}(PS_G(x))/Q, \forall x \in \{0,1\}^*$.*

Algorithm 1 The algorithm `scrypt` [19] and its core operation ROMix.

`scrypt`	ROMix
Input:	**Input:**
pwd {Password}	x {Initial State}
s {Salt}	G {Cost Parameter}
G {Cost Parameter}	**Output:** x {Hash value}
Output: x {Password Hash}	20: **for** $i = 0, \dots, G-1$ **do**
10: $x \leftarrow \text{PBKDF2}(pwd, s, 1, 1)$	21: $v_i \leftarrow x$
11: $x \leftarrow \text{ROMix}(x, G)$	22: $x \leftarrow H(x)$
12: $x \leftarrow \text{PBKDF2}(pwd, x, 1, 1)$	23: **end for**
13: **return** x	24: **for** $i = 0, \dots, G-1$ **do**
	25: $j \leftarrow x \bmod G$
	26: $x \leftarrow H(x \oplus v_j)$
	27: **end for**
	28: **return** x

In the following we define the Weak Garbage-Collector Attack (WGCA).

Definition 2 (Weak Garbage-Collector Attack). *Let $PS_G(\cdot)$ be a password scrambler that depends on a memory-cost parameter G, and let $F(\cdot)$ be an underlying function of $PS_G(\cdot)$ that can be efficiently computed. We say that an adversary $\mathcal{A}$ is successful in terms of a weak garbage-collector attack if a value $y = F(pwd)$ remains in memory during (almost) the entire runtime of $PS_G(pwd)$, where pwd denotes the secret input.*

An adversary that is capable of reading the internal memory of a password scrambler during its invocation, gains knowledge about y. Thus, it can reduce the effort for filtering invalid password candidates by just computing $y' = F(x)$ and checking whether $y = y'$, where x denotes the current password candidate. Note that the function F can also be given by the identity function. Then, the plain password remains in memory, rendering WGC attacks trivial (see Section 2.2 for a trivial WGC attack on `scrypt`).

2.2 (Weak) Garbage-Collector Attacks on `scrypt`

Garbage-Collector Attack on ROMix. Algorithm 1 describes the necessary details of the `scrypt` password scrambler together with its core function ROMix. The pre- and post-whitening steps are given by one call (each) of the standardized key-derivation function PBKDF2 [15], which we consider as a single call to a cryptographically secure hash function. The function ROMix takes the initial state x and the memory-cost parameter G as inputs. First, ROMix initializes an array v of size $G \cdot n$ by iteratively applying a cryptographic hash function H (see Lines 20-23), where n denotes the output size of H in bits. Second, ROMix accesses the internal state at randomly computed points j to update the password hash (see Lines 24-27).

It is easy to see that the value v_0 is a plain hash (using PBKDF2) of the original secret *pwd* (see Lines 10 and 21 for $i = 0$). Further, from the overall structure of `scrypt` and ROMix it follows that the internal memory is written once (Lines 20-23) but never overwritten . Thus, all values $v_0, \dots, v_{G-1}$ can be accessed by a garbage-collector adversary $\mathcal{A}$ after the termination of `scrypt`. For each password candidates pwd', $\mathcal{A}$ can now simply compute $x' \leftarrow \text{PBKDF2}(pwd')$ and check whether $x' = v_0$. If so, pwd' is a valid preimage. Thus, $\mathcal{A}$ can test each possible candidate in $\mathcal{O}(1)$, rendering an attack against `scrypt` (or especially ROMix) practical (and even memory-less).
As a possible countermeasure, one can simply overwrite $v_0, \dots, v_{G-1}$ after running ROMix. Nevertheless, this step might be removed by a compiler due to optimization, since it is algorithmically ineffective.

Weak Garbage-Collector Attack on `scrypt`. In Line 12 of Algorithm 1, `scrypt` invokes the key-derivation function PBKDF2 the second time using again the password *pwd* as input again. Thus, *pwd* has to be stored in memory during the entire invocation of `scrypt`, which implies that `scrypt` is vulnerable to WGC attacks.

3 Overview

Before we present the tables containing the comparison of the candidates for the Password Hashing Competition (PHC), we introduce the necessary notions (see Table 1) to understand the tables.

Identifier	Description
	Primitives/Structures
BC	Block cipher
SC	Stream cipher
PERM	Keyless permutation
HF	Hash function
BRG	Bit-Reversal Graph
DBG	Double-Butterfly Graph
	General Properties
CIU	Supports client-independent update
SR	Supports server relief
KDF	Usable as Key-Derivation Function (requires outputs to be pseudorandom)
FPO	Using floating-point operations
Flexible	Underlying primitive can be replaced
Iteration	Algorithm is based on iterations/rounds
	Security Properties
GCA Res.	Resistant against garbage-collector attacks (see Definition 1)
WGCA Res.	Resistant against weak garbage-collector attacks (see Definition 2)
SCA Res.	Resistant against side-channel attacks.
ROM-port	Special form of memory hardness [8].
Shortcut	Is it possible to bypass the main (memory and time) effort of an algorithm by knowing additional parameters, e.g., the Blum integers p and q for Makwa which are used to compute the modulo n.

Table 1. Notations used in Tables 2, 3, and 4.

Comments for Table 2. The values in the column "Memory" come from the authors recommendation for password hashing or are marked as '○' if no recommendation exists. The entry "A (CF)" denotes that only the compression function of algorithm A is used. An entry $A(XR)$ denotes that an algorithm A is reduced to X rounds. The `scrypt` password scrambler is just added for comparison. If an algorithm can only be partially be computed in parallel, we marked the corresponding entry with 'part.'. Note that POMELO and *schvrch* do not depend on an existing underlying primitive but on an own construction.

Algorithm	**Based On**	**Iteration**	**Memory Usage**	**Parallel**	**Underlying Primitive**		**Underlying Mode**
					BC/SC/PERM	**HF**	
AntCrypt		✓	32 kB	part.	-	SHA-512	-
ARGON	AES	✓	1 kB - 1 GB	✓	AES (5R)	-	-
battcrypt		✓	128 kB - 128 MB	part.	Blowfish-CBC	SHA-512	-
CATENA	BRG/DBG	✓	8 MB	part.	-	BLAKE2b	-
CENTRIFUGE		✓	2 MB	-	AES-256	SHA-512	-
EARWORM		✓	2 GB (ROM)	✓	AES (1R)	SHA-256	PBKDF2 $_{\text{HMAC}}$
Gambit	Sponge	✓	50 MB	-	Keccak_f	-	-
Lanarea DF		✓	256 B	-	-	BLAKE2b	-
Lyra2	Sponge	✓	400 MB - 1 GB	-	BLAKE2b (CF)	-	
MAKWA	Squarings	✓	negl.	✓	-	SHA-256	HMAC
MCS_PHS		✓	negl.	-	-	MCSSHA-8	-
ocrypt	`scrypt`	✓	1 MB - 1 GB	-	ChaCha	CubeHash	-
Parallel		✓	negl.	✓	-	SHA-512	-
PolyPassHash	Shamir Sec. Sharing	-	negl.	-	AES	SHA-256	-
POMELO		✓	(8 KB, 8 GB)	part.	-	-	-
Pufferfish	Blowfish/`bcrypt`	✓	4 - 16 kB	-	Blowfish	SHA-512	HMAC
Rig	BRG	✓	15 MB	part.	-	BLAKE2b	-
`scrypt`		✓	1 GB	-	Salsa20/8	-	PBKDF2
schvrch		✓	8 MB	part.	-	-	-
`Tortuga`	Sponge & rec. Feistel	✓	○		Turtle	-	-
SkinnyCat	BRG	✓	○	-	-	SHA-*/BLAKE2*	-
TwoCats	BRG	✓	○	✓	-	SHA-*/BLAKE2*	-
Yarn		✓	○	part.	BLAKE2b (CF), AES	-	-
yescrypt	`scrypt`	✓	3 MB (RAM)/3 GB (ROM)	part.	Salsa20/8	SHA-256	PBKDF2 $_{\text{HMAC}}$

Table 2. Overview of PHC Candidates and their general properties (Part 1).

Comments for Table 3. Even if the authors of a scheme do not claim to support client-independent update (CIU) or server relief (SR), we checked for the possibility and marked the corresponding entry in the table with '✓' or 'part.' if possible or possible under certain requirements, respectively. Note that we say that an algorithm does not support SR when it requires the whole state to be transmitted to the server. Moreover, we say that an algorithm does not support CIU if any additional information to the password hash itself is required. Note that Catena refers to both instantiations, i.e., Catena-BRG and Catena-DBG.

Algorithm	**CIU**	**SR**	**FPO**	**Flexible**
AntCrypt	✓	-	✓	part.
ARGON	✓	✓	-	✓
battcrypt	✓	-	-	part.
Catena	✓	✓	-	✓
CENTRIFUGE	-	-	-	✓
EARWORM	-	✓	-	-
Gambit	-	✓	opt.	part.
Lanarea DF	-	✓	-	✓
Lyra2	✓	✓	-	part.
Makwa	part.	-	-	✓
MCS_PHS	-	✓	-	part.
ocrypt	-	-	-	✓
Parallel	✓	✓	-	✓
PolyPassHash	✓	-	-	✓
POMELO	✓	-	-	-
Pufferfish	-	✓	-	part.
Rig	✓	✓	-	✓
`scrypt`	-	-	-	✓
schvrch	-	-	-	-
`Tortuga`	-	-	-	-
SkinnyCat	-	✓	-	✓
TwoCats	✓	✓	-	✓
Yarn	-	✓	-	-
yescrypt	-	✓	-	✓

Table 3. Overview of PHC Candidates and their general properties (Part 2).

Comments for Table 4. The column "Type" specifies which type of a memory-demanding design a certain algorithm satisfies. The types "A" and "B" are as described in Section 1 and marking an algorithm by "-" denotes that it is not designed to be memory-demanding. An entry supplemented by '*' (as for Memory-Hardness and Security Analysis), denotes that there exists not sophisticated analysis or proofs for the given claim/assumption. For SCA Res., 'part.' (partial) means that only one or more parts (but not all) provide resistance against side-channel attacks.

Algorithm	**Type**	**Memory-Hardness**	**KDF**	**GCA Res.**	**WGCA Res.**	**SCA Res.**	**Security Analysis**	**Shortcut**
AntCrypt	B	✓	✓	✓	✓	✓	✓*	-
ARGON	B	✓	✓	✓	✓	-	✓	-
battcrypt	B	✓	✓	✓	-	✓	✓*	-
Catena-BRG	B	✓	✓	-	✓	✓	✓	-
Catena-DBG	B	λ	✓	✓	✓	✓	✓	-
CENTRIFUGE	A	✓*	-	-	-	✓	✓*	-
EARWORM	B	ROM-port	-	✓	-	✓	✓	-
Gambit	B	✓*	✓	✓	✓	✓	✓*	-
Lanarea DF	B	✓*	✓	✓	✓	part.	✓*	-
Lyra2	B	✓	✓	✓	✓	part.	✓	-
Makwa	-	-	✓	✓	✓	part.	✓	✓
MCS_PHS	-	-	✓	✓	✓	✓	-	-
ocrypt	B	✓*	✓	✓	✓	-	✓*	-
Parallel	-	-	✓	✓	-	✓	✓*	-
PolyPassHash	-	-	-	-	-	-	✓	✓
POMELO	B	-	-	✓	✓	part.	✓*	-
Pufferfish	B	✓*	✓	✓	✓	-	✓*	-
Rig	B	λ	✓	✓	✓	✓	✓	-
`scrypt`	A	sequential	✓	-	-	-	✓	-
schvrch	B	-	-	✓	✓	✓	✓*	-
`Tortuga`	B	✓*	✓	✓	✓	✓	✓*	-
SkinnyCat	A	sequential	✓	-	-	part.	✓	-
TwoCats	B	sequential	✓	✓	✓	part.	✓	-
Yarn	B	✓*	-	✓	-	-	✓*	-
yescrypt	A	ROM-port, sequential	✓	-	-	-	✓*	-

Table 4. Overview over the security properties of PHC candidates.

Remark 2. Note that we do not claim completeness for Table 4. For example, we defined a scheme not to be resistant against side-channel attacks if it maintains a password-dependent memory-access pattern. Nevertheless, there exist several other types of side-channel attacks such as those based on power or acoustic analysis.

4 Resistance of PHC Candidates against (W)GC Attacks

In this section we briefly discuss potential weaknesses of each PHC candidate regarding to garbage-collector (GC) and weak-garbage collector (WGC) attacks or argue why it provides resistance against such attacks. Note that we assume the reader to be familiar with the internals of the candidates since we only concentrate on those parts of the candidates that are relevant regarding to GC/WGC attacks.

***AntCrypt* [9].** The internal state of AntCrypt is initialized with the secret *pwd*. During the hashing process, the state is overwritten multiple times (based on the parameter `outer_rounds` and `inner_rounds`), which thwarts GC attacks. Moreover, since *pwd* is used only to initialize the internal state, WGC attacks are not applicable.

ARGON *[3].* First, the internal state derived from *pwd* is the input to the padding phase. After the padding phase, the internal state is overwritten by applying the functions `ShuffleSlices` and `SubGroups` at least L times. Based on this structure, and since *pwd* is used only to initialize the state, ARGON is not vulnerable against GC/WGC attacks.

***battcrypt* [24].** Within battcrypt, the plain password is used only once, namely to generate a value $key = \text{SHA-512}(\text{SHA-512}(salt \mid\mid pwd))$. The value *key* is then used to initialize the internal state, which is expanded afterwards. In the *Work* phase, the internal state is overwritten `t_cost`×`m_size` times using password-dependent indices. Thus, GC attacks are not applicable.

Note that the value *key* is used in the three phases *Initialize blowfish*, *Initialize data*, and *Finish*, whereas it is overwritten in the phase *Finish* the first time. Note that the main effort for battcrypt is given by the *Work* phase. Thus, one can assume that one iteration of the outer loop (iterating over *t_cost_upgrade*) lasts long enough for a WGC adversary to launch the following attack:
For each password candidates x and the known value *salt*, compute $key' = \text{SHA512}(\text{SHA512}(salt \mid\mid x))$ and check whether $key' = key$. If so, mark x as a valid password candidate.

Catena *[11].* Catena has two instantiations Catena-BRG and Catena-DBG, which are based on a (G, λ)-Bit-Reversal Graph and a (G, λ)-Double-Butterfly Graph, respectively. Both instantiations use an array of G elements

each as their internal state. This state is overwritten $\lambda - 1$ times for Catena-BRG and $(2\log_2(G)-1)\cdot\lambda+2\log_2(G)-2$ times for Catena-DBG. Hence, when considering Catena-BRG, a GC adversary with access to the state can reduce the effort for testing a password candidate by a factor of $1/\lambda$. When considering Catena-DBG, the reduction of the computational cost of an adversary is negligible. The authors mention this fact by recommending Catena-DBG when considering GC attacks.

For Catena-BRG as well as Catena-DBG, the password *pwd* is used only to initialize the internal state. Thus, both instantiations provide resistance against WGC attacks.

CENTRIFUGE [1]. The internal state M of size `p_mem`×`outlen` byte is initialized with a seed S derived from the password and the salt as follows: $S = H(s_L \mid\mid s_R)$, where $s_L \leftarrow H(pwd \mid\mid len(pwd))$ and $s_R \leftarrow H(\text{salt} \mid\mid len(\text{salt}))$. Furthermore, S is used as the initialization vector (IV) and the key for the CFB encryption. The internal M is written once and later only accessed in a password-dependent manner. Thus, a GC adversary can launch the following attack:

1. receive the internal state M (or at least $M[1]$) from memory
2. for each password candidate x:
   (a) initialization (seeding and S-box)
   (b) compute the first table entry $M'[1]$ (during the *build table* step)
   (c) check whether $M'[1] = M[1]$

The final step of CENTRIFUGE is to encrypt the internal state, requiring the key and the IV, which therefore must remain in memory during the invocation of CENTRIFUGE. Thus, the following WGC attack is applicable:

1. Compute $s_R \leftarrow H(salt \mid\mid len(salt))$
2. For every password candidate x:
   (a) Compute $s'_L \leftarrow H(x \mid\mid len(x))$ and $S' = H(s'_L \mid\mid s_R)$, and compare if $S' = IV$
   (b) If yes: mark x as a valid password candidate
   (c) If no: go to Step 2

EARWORM ***[12].*** EARWORM maintains an array called *arena* which consists of $2^{m_cost} \times L \times W$ 128-bit blocks, where $W = 4$ and $L = 64$ are recommended by the authors. This read-only array is randomly initialized (using an additional secret input which has to be constant within a given system) and used as AES round keys. Since the values within this array do not depend on the secret *pwd*, knowledge about *arena* does not help any malicious garbage collector. Within the main function of EARWORM (WORKUNIT), an internal state *scratchpad* is updated multiple times using password-dependent accesses to *arena*. Thus, a GC adversary cannot profit from knowledge about *scratchpad*, rendering GC attacks not applicable.

Within the function WORKUNIT, the value *scratchpad_tmpbuf* is derived directly from the password as follows:

$$scratchpad_tmpbuf \leftarrow \text{EWPRF}(pwd, 01 \mid\mid salt, 16W),$$

where EWPRF denotes $\text{PBKDF2}_{\text{HMAC-SHA256}}$ with the first input denoting the secret key. This value is updated only at the end of WORKUNIT using the internal state. Thus, it has to be in memory during almost the whole invocation of EARWORM, rendering the following WGC attack possible: For each password candidate x and the known value *salt*, compute $y = \text{EWPRF}(x, 01 \mid\mid salt, 16W)$ and check whether $scratchpad_tmpbuf = y$. If so, mark x as a valid password candidate.

***Gambit** [21].* Gambit bases on a duplex-sponge construction [2] maintaining two internal states S and Mem, where S is used to subsequently update Mem. First, password and salt are absorbed into the sponge and after one call to the underlying permutation, the squeezed value is written to the internal state Mem and processed r times (number of words in the ratio of S). The output after the r steps is optionally XORed with an array lying in the ROM. After that, Mem is absorbed into S again. This step is executed t times, where t denotes the time-cost parameter. The size of Mem is given by m, the memory-cost parameter. Continuously updating the states Mem and S thwarts GC attacks. Moreover, since *pwd* is used only to initialize the state within the sponge construction, WGC attacks are not applicable.

Lanarea DF *[18].* Lanarea DF maintains a matrix (internal state) consisting of $16 \cdot 16 \cdot m_cost$ byte values, where m_cost denotes the memory-cost parameter. After the password-independent setup phase, the password is processed by the internal pseudorandom function producing the array $(h_0, \ldots, h_{31})$, which determines the positions on which the internal state is accessed during the core phase (thus, allowing cache-timing attacks). In the core phase, the internal state is overwritten $t_cost \times m_cost \times 16$ times, rendering GC attacks impossible. Moreover, the array $(h_0, \ldots, h_{31})$ is overwritten $t_cost \times m_cost$ times which thwarts WGC attacks.

***Lyra2** [14].* The Lyra2 password scrambler (and KDF) is based on a duplex sponge construction maintaining a state H, which is initialized with the password, the salt, and some tweak in the first step of its algorithm. The authors indicate that the password can be overwritten from this point on, rendering WGC attacks impossible. Moreover, Lyra2 maintains an internal state M, which is overwritten (updated using values from the sponge state H) multiple times. Thus, GC attacks are not applicable for Lyra2.

Makwa *[22].* Makwa has not been designed to be a memory-demanding password scrambler. Its strength is based on a high number of squarings modulo a composite (Blum) integer n. The plain (or hashed) password is used twice to initialize the internal state, which is then processed by squarings modulo n. Thus, neither GC nor WGC attacks are applicable for Makwa.

***MCS_PHS* [17].** Depending on the size of the output, MCS_PHS applies iterated hashing operations, reducing the output size of the hash function by one byte in each iteration – starting from 64 bytes. Note that the memory-cost parameter m_cost is used only to increase the size of the initial chaining value T_0. The secret input *pwd* is used once, namely when computing the value T_0 and can be deleted afterwards, rendering WGC attacks not applicable. Furthermore, since the output of MCS_PHS is computed by iteratively applying the underlying hash function (without handling an internal state which has to be placed in memory), GC attacks are not possible.

***ocrypt* [10].** The basic idea of ocrypt is similar to that of `scrypt`, besides the fact that the random memory accesses are determined by the output of a stream cipher (ChaCha) instead of a hash function cascade. The output of the stream cipher determines which element of the internal state is updated, which consists of $2^{17+m_{cost}}$ 64-bit words. During the invocation of ocrypt, the password is used only twice: (1) as input to CubeHash, generating the key for the stream cipher and (2) to initialize the internal state. Neither the password nor the output of CubeHash are used again after the initialization. Thus, ocrypt is not vulnerable to WGC attacks.
The internal state is processed $2^{17+t_{cost}}$ times, where in each step one word of the state is updated. Since the indices of the array elements accessed depend only on the password and not on the content, GC attacks are not possible by observing the internal state after the invocation of ocrypt.

Remark 3. Note that the authors of ocrypt claim side-channel resistance since the indices of the array elements are chosen in a password-independent way. But, as the password (beyond other inputs) is used to derive the key of the underlying stream cipher, this assumption does not hold, i.e., the output of the stream cipher depends on the password, rendering (theoretical) cache-timing attacks possible.

***Parallel* [25].** Parallel has not been designed to be a memory-demanding password scrambler. Instead, it is highly optimized to be comuted in parallel. First, a value *key* is derived from the secret input *pwd* and the salt by

$$key = \text{SHA-512}(\text{SHA-512}(salt) \mid\mid pwd).$$

The value *key* is used (without being changed) during the Clear work phase of Parallel. Since this phase defines the main effort for computing the password hash, it is highly likely that a WGC adversary can gain knowledge about *key*. Then, the following WGC attack is possible: For each password candidate x and the known value *salt*, compute $y = \text{SHA-512}(\text{SHA-512}(salt) \mid\mid x)$ and check whether $key = y$. If so, mark x as a valid password candidate. Since the internal state is only given by the subsequently updated output of SHA-512, GC attacks are not applicable for Parallel.

***PolyPassHash* [5].** PolyPassHash denotes a threshold system with the goal to protect an individual password (hash) until a certain number of correct passwords (and their corresponding hashes) are known. Thus, it aims at protecting

an individual password hash within a file containing a lot of password hashes, rendering PolyPassHash not to be a password scrambler itself. The protection lies in the fact that one cannot easily verify a target hash without knowing a minimum number of hashes (this technical approach is referred to as PolyHashing). In the PolyHashing construction, one maintains a (k, n)-threshold cryptosystem, e.g., Shamir Secret Sharing. Each password hash $h(pwd_i)$ is blinded by a share $s(i)$ for $1 \leq i \leq k \leq n$. The value $z_i = h(pwd_i) \oplus s(i)$ is stored in a so-called PolyHashing store at index i. The shares $s(i)$ are not stored on disk. But, to be efficient, a legal party, e.g., a server of a social networking system, has to store at least k shares in the RAM to on-the-fly compare incoming requests on-the-fly. Thus, this system only provides security against adversaries which are only able to read the hard disk but not the volatile memory (RAM).

Since the secret (of the threshold cryptosystem) or at least the k shares have to be in memory, GC attacks are possible by just reading the corresponding memory. The password itself is only hashed and blinded by $s(i)$. Thus, if an adversary is able to read the shares or the secret from memory, it can easily filter wrong password candidates, i.e., makeing PolyPassHash vulnerable against WGC attacks.

POMELO [27]. POMELO contains three update functions $F(S, i), G(S, i, j)$, and $H(S, i)$, where S denotes the internal state and i and j the indices at which the state is accessed. Those functions update at most two state words per invocation. The functions F and G provide deterministic random-memory accesses (determined by the cost parameter t_cost and m_cost), whereas the function H provides random-memory accesses determined by the password, rendering POMELO at least partially vulnerable to cache-time attacks. Since the password is used only to initialize the state, which itself is overwritten about $2^{2 \cdot t_cost} + 2$ times, POMELO provides resistance against GC and WGC attacks.

Pufferfish [13]. The main memory used within Pufferfish is given by a two-dimensional array consisting of 2^{5+m_cost} 512-bit values, which is regularly accessed during the password hash generation. The first steps of Pufferfish are given by hashing the password. The result is then overwritten $2^{5+m_cost} + 3$ times, rendering WGC attacks not possible. The state word containing the hash of the password ($S[0][0]$) is overwritten 2^{t_cost} times. Thus, there does not exist a shortcut for an adversary, rendering GC attacks impossible.

Rig [6]. Rig maintains two arrays a (sequential access) and k (bit-reversal access). Both arrays are iteratively overwritten $r \cdot n$ times, where r denotes the round parameter and n the iteration parameter. Thus, rendering Rig resistant against GC attacks. Note that within the setup phase, a value α is computed by

$$\alpha = H_1(x) \quad \text{with} \quad x = pwd \mid\mid len(pwd) \mid\mid \ldots,$$

Since the first α (which is directly derived from the password) is only used during the initialization phase, WGC attacks are not applicable.

schvrch ***[26].*** The password scrambler *schvrch* maintains an internal state of $256 \cdot 64$-bit words (2 kB), which is initialized with the password, salt and their corresponding lengths, and the final output length. After this step, the password can be overwritten in memory. This state is processed t_cost times by a function $revolve()$, which affects in each invocation all state words. Next, after applying a function $stir()$ (again, changing all state entries), it expands the state to m_cost times the state length. Each part (of size state length) is then processed to update the internal state, producing the hash after each part was processed. Thus, the state word initially containing the password is overwritten $t_cost \cdot m_cost$ times, rendering GC attacks impossible. Further, neither the password nor a value directly derived from it is required during the invocation of *schvrch*, which thwarts WGC attacks.

`Tortuga` [23]. GC and WGC attacks are not possible for `Tortuga` since the password is absorbed to the underlying sponge structure, which is then processed at least two times by the underlying keyed permutation (Turtle block cipher [4]), and neither the password nor a value derived from it has to be in memory.

SkinnyCat and TwoCats [7]. SkinnyCat is a subset of the TwoCats scheme optimized for implementation. Both algorithms maintain a 256-bit state *state* and an array of 2^{m_cost+8} 32-bit values (mem). During the initialization, a value PRK is computed as follows:

$$PRK = Hash(len(pwd), len(salt), \ldots, pwd, salt).$$

The value PRK is used in the initialization phase and first overwritten in the forelast step of SkinnyCat (when the function *addIntoHash()* is invoked). Thus, an adversary that gains knowledge about the value PRK is able to launch the following WGC attack: For each password candidates x and the known value $salt$, compute $PRK' = Hash(len(x), len(salt), \ldots, x, salt)$ and check whether $PRK = PRK'$. If so, mark x as a valid password candidate.

Within TwoCats, the value PRK is overwritten at an early state of the hash value generation. TwoCats maintains consists of a garlic application loop from $startMemCost = 0$ to $stopMemCost$, where $stopMemCost$ is a user-defined value. In each iteration, the value PRK is overwritten, rendering WGC attacks for TwoCats not possible.

Both SkinnyCat and TwoCats consist of two phases each. The first phase updates the first half of the memory (early memory) $mem[0, \ldots, memlen/(2 \cdot blocklen) - 1]$, where the memory is accessed in a password-independent manner. The second phase updates the second half of the memory $mem[memlen/(2 \cdot blocklen), \ldots, memlen/blocklen - 1]$, where the memory is accessed in a password-dependent manner. Thus, both schemes provide only partial resistance against cache-timing attacks. For SkinnyCat, the early memory is never overwritten, rendering the following GC attack possible:

1. Obtain $mem[0, \ldots, memlen/(2 \cdot blocklen) - 1]$ and PRK from memory
2. Create a state $state'$ and an array mem' of the same size as $state$ and mem, respectively
3. Set $fromAddr = slidingReverse(1) \cdot blocklen$, $prevAddr = 0$, and $toAddr = blocklen$
4. For each password candidate x:
   (a) Compute PRK' as described using the password candidate x
   (b) Initialize $state'$ and mem' as prescribed using PRK'
   (c) Compute $state'[0] = (state'[0] + mem'[1]) \oplus mem'[fromAddr + +]$
   (d) Compute $state'[0] = \mathrm{ROTATE_LEFT}(state'[0], 8)$
   (e) Compute $mem'[blocklen + 1] = state'[0]$
   (f) Check whether $mem'[blocklen + 1] = mem[blocklen + 1]$
   (g) If yes: mark x as a valid password candidate
   (h) If no: go to Step 4.

Note that this attack does not work for TwoCats since an additional feature in comparison to SkinnyCat is that the early memory is overwritten.

Yarn [16]. Yarn maintains two arrays *state* and *memory*, consisting of *par* and 2^{m_cost} 16-byte blocks, respectively. The array *state* is initialized using the salt. Afterwards, *state* is processed using the BLAKE2b compression function with the password *pwd* as message, resulting in an updated array *state1*. This array has to be stored in memory since it is used as input to the final phse of Yarn. The array *state* is expanded afterwards and further, it is used to initialize the array *memory*. Next, *memory* is updated continuously. Both *memory* and *state* are overwritten continuously. The array *state1* is overwritten at the lastest in the final phase of Yarn. Thus, GC attacks are not possible for Yarn. Nevertheless, the array *state1* is directly derived from *pwd* and stored until the final phase occurs. Thus, the following WGC attack is possible:

1. Compute $h \leftarrow$ Blake2b_GenerateInitialState($outlen, salt, pers$) as in the first phase of Yarn
2. For each password candidate x:
   (a) Compute $h' \leftarrow$ Blake2b_ConsumeInput(h, x)
   (b) Compute *state1'*$\leftarrow$ Truncate($h', outlen$) and check whether *state1'* = *state1*

yescrypt [20]. The yescrypt password scrambler maintains two lookup tables V and $VROM$, where V is located in the RAM and $VROM$ in the ROM. Depending on the flag `YESCRYPT_RW`, the behaviour of the memory management in RAM can be switched from "write once, read many" to "read-write". Nevertheless, yescrypt does not completey overwrite the memory in RAM, rendering similar GC attacks as for `scrypt` possible (see Section 2.2). But, such an attack would require a higher effort in comparison the attack on `scrypt` since yescrypt at least partially overwrites the RAM locations.

When considering WGC attacks, one has to differ between two variants of yescrypt depending whether it runs in the `scrypt` compatibility mode or not. In `scrypt` compatibility mode, obviously the same WGC as for `scrypt` is applicable (see Section 2.2). If not running in `scrypt` compatibility mode, yescrypt uses the results of the initial call to PBKDF2 in the last step. Thus, the value which has to remain in memory is given by HMAC-SHA-256(SHA-256(pwd), $salt$). Since it is also possible to compute HMAC and SHA-256 efficiently, yescrypt does not provide resistance against WGC attacks.

5 Conclusion

In this work we provided an overview of the first-round candidates of the Password Hashing Competition, which are not yet withdrawn. Further, we analyzed each algorithm regarding to its vulnerability against garbage-collector and weak garbage-collector attacks. Even if both attacks require access to the memory on the target's machine, they show a potential weakness, which should be taken into consideration. As a results, we have shown GC attacks on Catena-BRG, CENTRIFUGE, PolyPassHash, `scrypt`, SkinnyCat, and yescrypt. Additionally, we have shown that WGC attacks are applicable to battcrypt, CENTRIFUGE, EARWORM, Parallel, PolyPassHash, `scrypt`, SkinnyCat, Yarn, and yescrypt.

6 Acknowledgement

Thanks to A. Peslyak, J. M. Gosney, H. Wu, B. Cox, D. Khovratovich, and all contributors to the PHC mailing list for providing us with valuable comments and fruitful discussions.

References

1. Rafael Alvarez. CENTRIFUGE – A password hashing algorithm. `https://password-hashing.net/submissions/specs/Centrifuge-v0.pdf`, 2014.
2. Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography*, volume 7118 of *Lecture Notes in Computer Science*, pages 320–337. Springer, 2011.
3. Alex Biryukov and Dmitry Khovratovich. ARGON v1: Password Hashing Scheme. `https://password-hashing.net/submissions/specs/Argon-v1.pdf`, 2014.
4. Matt Blaze. Efficient Symmetric-Key Ciphers Based on an NP-Complete Subproblem, 1996.
5. Justin Cappos. PolyPassHash: Protecting Passwords In The Event Of A Password File Disclosure. https://password-hashing.net/submissions/specs/PolyPassHash-v0.pdf, 2014.
6. Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Rig: A simple, secure and flexible design for Password Hashing. https://password-hashing.net/submissions/specs/RIG-v2.pdf, 2014.

7. Bill Cox. TwoCats (and SkinnyCat): A Compute Time and Sequential Memory Hard Password Hashing Scheme. https://password-hashing.net/submissions/specs/TwoCats-v0.pdf, 2014.
8. Solar Designer. New developments in password hashing: ROM-port-hard functions. http://distro.ibiblio.org/openwall/presentations/New-In-Password-Hashing/ZeroNights2012-New-In-Password-Hashing.pdf, 2012.
9. Markus Dürmuth and Ralf Zimmermann. AntCrypt. `https://password-hashing.net/submissions/AntCrypt-v0.pdf`, 2014.
10. Brandon Enright. Omega Crypt (ocrypt). https://password-hashing.net/submissions/specs/OmegaCrypt-v0.pdf, 2014.
11. Christian Forler, Stefan Lucks, and Jakob Wenzel. The Catena Password-Scrambling Framework. `https://password-hashing.net/submissions/specs/Catena-v2.pdf`, 2014.
12. Daniel Franke. The EARWORM Password Hashing Algorithm. https://password-hashing.net/submissions/specs/EARWORM-v0.pdf, 2014.
13. Jeremi M. Gosney. The Pufferfish Password Hashing Scheme. https://password-hashing.net/submissions/specs/Pufferfish-v0.pdf, 2014.
14. Marcos A. Simplicio Jr, Leonardo C. Almeida, Ewerton R. Andrade, Paulo C. F. dos Santos, and Paulo S. L. M. Barreto. The Lyra2 reference guide. https://password-hashing.net/submissions/specs/Lyra2-v1.pdf, 2014.
15. B. Kaliski. RFC 2898 - PKCS #5: Password-Based Cryptography Specification Version 2.0. Technical report, IETF, 2000.
16. Evgeny Kapun. Yarn password hashing function. https://password-hashing.net/submissions/specs/Yarn-v2.pdf, 2014.
17. Mikhail Maslennikov. PASSWORD HASHING SCHEME MCS_PHS. https://password-hashing.net/submissions/specs/MCS_PHS-v2.pdf, 2014.
18. Haneef Mubarak. Lanarea DF. https://password-hashing.net/submissions/specs/Lanarea-v0.pdf, 2014.
19. Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. presented at BSDCan'09, May 2009, 2009.
20. Alexander Peslyak. yescrypt - a Password Hashing Competition submission. https://password-hashing.net/submissions/specs/yescrypt-v0.pdf, 2014.
21. Krisztián Pintér. Gambit – A sponge based, memory hard key derivation function. https://password-hashing.net/submissions/specs/Gambit-v1.pdf, 2014.
22. Thomas Pornin. The MAKWA Password Hashing Function. https://password-hashing.net/submissions/specs/Makwa-v0.pdf, 2014.
23. Teath Sch. Tortuga – Password hashing based on the Turtle algorithm. https://password-hashing.net/submissions/specs/Tortuga-v0.pdf, 2014.
24. Steve Thomas. battcrypt (Blowfish All The Things). `https://password-hashing.net/submissions/specs/battcrypt-v0.pdf`, 2014.
25. Steve Thomas. Parallel. https://password-hashing.net/submissions/specs/Parallel-v0.pdf, 2014.
26. Rade Vuckovac. *schvrch*. https://password-hashing.net/submissions/specs/Schvrch-v0.pdf, 2014.
27. Hongjun Wu. POMELO: A Password Hashing Algorithm. https://password-hashing.net/submissions/specs/POMELO-v1.pdf, 2014.

Cryptographic module based approach for password hashing schemes

Donghoon Chang, Arpan Jati, Sweta Mishra, Somitra Kumar Sanadhya

Indraprashtha Institute of Information Technology, Delhi (IIIT-Delhi), India
{donghoon,arpanj,swetam,somitra}@iiitd.ac.in

Abstract. Password Hashing is the technique of performing one-way transformation of the password. One of the requirements of password hashing algorithms is that it should be memory demanding to provide defense against hardware attacks. In practice, most Cryptographic designs are implemented inside a Cryptographic module, as suggested by NIST in a set of standards (FIPS 140). A cryptographic module has a limited amount of memory and this makes it challenging to implement a password hashing algorithm inside it.

In this work, we propose a novel approach to allow a limited memory cryptographic module to be used in the implementation of a high memory password hashing algorithm. We also analyze all the entries of the Password Hashing Competition (PHC) to evaluate the suitability of the submitted algorithms to be implemented in a Cryptographic module. We show that all the submissions to the PHC can be securely implemented in a crypto-module following our suggestion. To the best of our knowledge, this is the first attempt in the direction of secure implementation of password hashing algorithms.

Keywords: Password, Password hashing, Cryptographic module, Cryptographic module based password hashing.

1 Introduction

Passwords are the most common technique for user authentication currently. Specifically, a password is a secret word or string of characters which is used by a principal to prove her identity as an authentic user to gain access to a resource. In order to ensure the confidentiality of the secret password, only its one-way transformation is stored and not the password itself. The technique used for this one-way transformation is called 'Password Hashing'. The password hashing transforms the password into another string, called the 'hashed' password which is usually stored in the database of the server (or authenticator of the system). Following Moore's Law [23], the hardware is becoming more and more powerful with time at a constant cost factor. Multi-core GPUs and ASIC modules are commonly used to implement brute force attack over possible password choices. These hardware perform highly efficiently when the password hashing computation requires negligible storage. However, these attacks are comparatively inefficient when the computation requires relatively large amount of memory. Therefore, consumption of comparatively huge memory by the password hashing design is a design goal of such schemes to provide defense against hardware threats.

The most common threat of password hashing is that the server database which contains the password hashes is vulnerable to adversaries. Since the user chosen passwords usually have low entropy, it is quite feasible for the adversary to guess the passwords and to compute the corresponding password hashes. These computed hashes are then matched with the compromised database to retrieve the actual password. This is termed the "offline attack" against the password hashing scheme. There can be "online attacks" as well in which the attacker actively measures information leakage at the time the password hash algorithm is performing its computations. While evaluating the security, we can not even trust the server. The natural question in this case is if it is feasible to provide security even when the server is compromised. It is challenging to provide a system which provides the same level of security as could be achieved when the server was secure, however, it is indeed possible to increase the level of difficulty of the adversary to reveal the sensitive data. With the goal of restricting the active adversary as well as the off-line attacker (who compromises password hash file to reveal passwords), the approach taken in this paper is the use of a cryptographic module for the password hash computation. In a series of standards (FIPS 140), NIST already recommends the use of a cryptographic module for stream ciphers, block ciphers, authenticated encryption, key generation algorithms etc.

Password hashing algorithms are a natural candidate for the use of a cryptographic module to implement them and we feel that these modules will be required, at least for some use cases, in the future.

A cryptographic module is a device (plus the requisite software to make it work) with a secret key stored inside it and it is assumed that the device itself is secure against all types of adversaries. Cryptographic modules are usually secured physically and the attacker is not allowed access to the input of the device. The only part where the attacker can mount an attack is the output of the module. In general, it is hard to keep a big and bulky device from attackers and hence such a device usually contains a small amount of memory. This memory could range from few bytes for tiny devices to few kilobytes.

As mentioned above, a major limitation of a cryptographic module is that it has a limited memory. However, by design, all password hashing schemes require comparatively large memory. Therefore it is not always feasible to implement a complete password hash computation on a cryptographic module. Some schemes can still be easily fitted into a small memory and yet require large memory while computing the password hash. On the other hand, some other schemes do not directly fit inside a limited memory device. These latter class of schemes need tweaks in them, which may cause the security proof of the original scheme to fail. In this paper we consider possible approaches to provide security of password hashing schemes using cryptographic module. We also analyze all password hashing designs in the ongoing Password Hashing Competition (PHC) with respect to their suitability to be implemented inside a small memory crypto-device.

The rest of the document is organized as follows. In section 2 we explain the term cryptographic module and its importance. This is followed by the general approach towards the design of cryptographic module based password hashing scheme in section 3. The security analysis of this approach is provided in sections 4. Subsequently, the analysis of the submitted PHC designs with respect to cryptographic module based approach is included in sections 5. Finally, we conclude the paper in section 6.

2 Cryptographic Module

The Federal Information Processing Standards (FIPS) specify requirements for cryptographic modules for various security algorithms in FIPS 140 series. The standard FIPS 140-2 [1] defines a cryptographic module as: “the set of hardware, software, firmware, or some combination thereof that implements cryptographic logic or processes, including cryptographic algorithms, and is contained within the cryptographic boundary of the module”.

The standard [1] specifies the security requirements satisfied by a cryptographic module when implemented within a security system protecting sensitive but unclassified information. It provides four increasing, qualitative levels of security that are intended to cover the wide range of potential applications and environments in which cryptographic modules may be employed. The available cryptographic services for a cryptographic module are based on many factors that are specific to the application and the environment. Cryptographic modules should use cryptographic algorithms that have been approved for protecting Federal government sensitive information. The four security levels as defined in [1] are as follows.

1. **Security Level 1:** It provides the lowest level of security by supporting basic security requirements specified for a cryptographic module (e.g., use of at least one approved algorithm or approved security function). This level does not require any specific physical security mechanisms that are beyond the basic requirement for production-grade components.
2. **Security Level 2:** This level enhances the physical security mechanisms of a Security Level 1. This can be provided by adding the requirement for tamper-evidence, which includes the use of tamper-evident coatings or seals or for pick-resistant locks on removable covers or doors of the module.

3. **Security Level 3:** This level enhances the physical security mechanisms required at Security Level 2. Apart from the requirement of tamper-evident, this level attempts to prevent the intruder from gaining access to critical security parameters held within the cryptographic module. This level is intended to provide physical security requirements with high probability of detection and response to attempts at physical access, use or modification of the cryptographic module .
4. **Security Level 4:** This level provides the highest level of security defined in the standard. At this level, the physical security mechanisms provide a complete envelope of protection around the cryptographic module. The implemented mechanisms intent of detecting and responding to all unauthorized attempts at physical access. There is a very high probability of detection of penetration of the cryptographic module from any direction and the detection results in the immediate 'zeroization' [1] of all plaintext critical security parameters. The cryptographic modules of this level are useful for operation in physically unprotected environments.

The Cryptographic Module Validation Program of the NIST actively checks the accuracy of the applied algorithms, therefore we get the assurance that the claimed security is indeed provided and that the cryptographic module can protect the sensitive information and provide the required security.

With the increasing need of securing access to resources, and to protect the user behavior, cryptographic modules are becoming more and more prevalent with time.

3 General approach for cryptographic module based password hashing scheme

From the design restriction of cryptographic module, it is known that we can not implement an algorithm which requires comparatively huge memory for its implementation. Therefore it is ever challenging to provide a complete cryptographic module based approach for password hashing schemes. Considering this challenge we come to the conclusion that the only possibility is to protect the password. The entropy of the password is commonly weak as we can not force the user of the password to choose a strong one. Therefore another requirement is to increase the entropy. This is possible if we use a large secret key (at least 128 bit) with the password. The responsibility of the cryptographic module will be to secure the key and to provide (typically) the server, the one-way transformation of this password operated with key (it can take other parameters as well depending on the algorithm). Then the server can follow further computations of the password hashing algorithm that can be suitably applied with this approach to compute the password hash. The overview of the above procedure is shown in Fig. 1. The steps of this general procedure is specified in the following protocol.

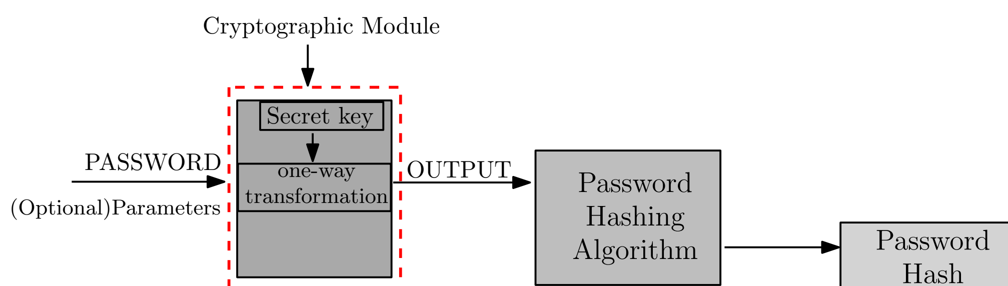


Fig. 1: Cryptographic module used in password hashing

The Protocol for the general cryptographic module based password hashing scheme

1. Password is the input to the cryptographic module (it can take any number of other parameters depending on the password hashing algorithm).

2. Cryptographic module performs a one-way transformation over the password and the secret key (and other parameters if applicable); this output is then provided to the server.
3. Server performs the remaining steps of the password hashing algorithm and produces the final password hash.

4 Security analysis

The security of the overall design depends on the security provided by the cryptographic module to protect the password and the security of the password hashing algorithm. For the resistance from the vulnerability of the adversary we can assume that guessing a large random key (at least 128 bit) is quite challenging. Therefore predicting the output of the cryptographic module will be difficult.

By using the random secret key of the cryptographic module we increase the entropy of the password and also restrict the adversary if it tries to compromise the data during execution of the password hashing algorithm. Therefore this design approach for password hashing provides enhanced security if the underlying password hashing design is secure.

We summarize the overall requirement for the secure implementation of the proposed cryptographic module based approach to claim the security of the system as follows:

- **The input password of the cryptographic module:** The input password of the cryptographic module should be secure. One way to achieve this security is by encrypting the password. This is required to ensure that the password will be secure even if intercepted by the adversary before the computation. In this case we assume that a standard encryption algorithm is used for the encryption of the password.
- **The internal operation of the cryptographic module:** The cryptographic module should perform the one-way transformation operating the secret key with the decrypted password (and with optional parameters if available). The security provided by the cryptographic module as explained in section 2 ensures the security of the internal operation.
- **The input to the password hashing algorithm:** The output of the cryptographic module should be the input to the password hashing algorithm (with other parameters depending on the algorithm used). The level of difficulty to guess this output depends on the algorithm implemented by the cryptographic module and the strength of the secret key. Therefore we can claim that it will be difficult to guess the output. Hence use of a low entropy password with a secret key to perform one-way transformation will definitely provide better security than the plain-text password.
- **The security of the password hashing algorithm:** The password hashing algorithm should fulfill the requirements as mentioned in [2] and based on these criteria we can ensure the level of security provided by the scheme.

From the above explanation, we can state that if a password hashing technique follows the above mentioned approach then the cryptographic module based system will always provide better security than the usual implementation of a password hashing algorithm (without using a crypto-module).

5 Analysis of submitted PHC designs with respect to cryptographic module based approach

In this section we analyze the design of all the PHC submissions [2] to verify whether use of cryptographic module is possible without significant modification of the designs. We try to follow similar approach to provide solutions for all the designs, even though other approaches can also be suitable. We provide the graphical view of the designs and highlight (in red) the area that can be implemented in a cryptographic module. We include the security analysis of cryptographic module implementation for few of the designs (at least one from each category as discussed in section 5.1). It is easy to extend this approach to the remaining designs for which the proof of security is not provided/ commented upon.

– **AntCrypt [9]** The State array of $2^{\mathrm{m_cost}+8}$ bytes of AntCrypt is initialized with salt and password and updated throughout the execution of the algorithm as shown in Fig. 2(a). To implement cryptographic module based approach, we can replace the values (salt and password) highlighted in red in Fig. 2 by the output of the cryptographic module which appends the secret key and performs the one-way transformation over the concatenated values. This little modification does not affect the remaining design of AntCrypt as explained in [9].
Security analysis: The security of the design as claimed by the author in [9] depends on the underlying hash function. The design writes the password to the state in the beginning and immediately overwrites it by the output of the hash function. Following our proposal as shown in Fig. 2(b), the initial salt and password is overwritten with pseudo-random output of the cryptographic module. Remaining design uses this pseudo-random value in place of the (usual low entropy) password. Hence suggested modification does not affect the security of the overall design, even replacing the plain-text password with pseudo-random value enhances the security from information leakage.

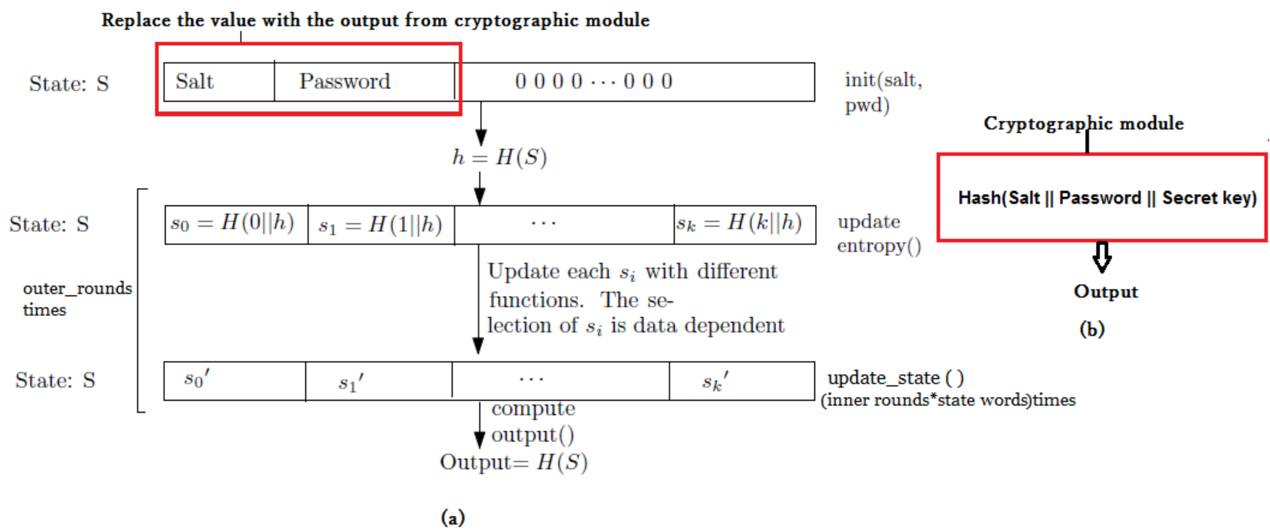


Fig. 2: (a)Overview of the design AntCrypt (b) The cryptographic module implementation

– **ARGON [5]** The algorithm implements four phases. The first 'Padding phase' takes input password, salt, secret value, other parameters and use zero padding to obtain specified length (32(t-4)bytes, see [5] for more detail) and creates block of data. 'Initial round' applies AES to each block. In the next 'Main round' the transformations ShuffleSlices and SubGroups are applied alternatively L times. At the end 'Finalization round' generates the output tag. As mentioned, the padding phase of ARGON is initialized with password, salt and secret key. We suggest to replace this secret key with the secret key of the cryptographic module and to apply the one-way transformation on the values highlighted in red to create the 'Input' as shown in Fig. 3(a). Argon uses the only cryptographic primitive AES. Therefore use of other primitive will modify the design requirement. In that case we suggest to use HMAC where the key will be the secret key of the module and hash function can be any block cipher (AES) based hash function. This modification does not affect the remaining design of ARGON which is explained in [5].
Security analysis: As per the suggestion, the use of HMAC with AES based hash function and the secret key to hash the value inside the cryptographic module provide better security from information leakage than using the parameters in the plain-text form. This modification does not affect the remaining implementation of the design and the overall claimed security of Argon.

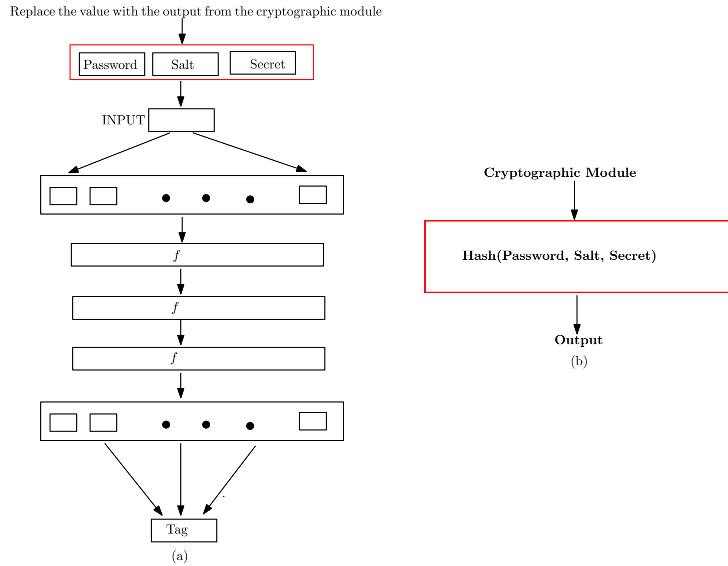


Fig. 3: (a)Overview of the design Argon taken from [5] , (b) The cryptographic module implementation

– **battcrypt [24]** The algorithm initializes the value key by taking hash of salt concatenated with password. We can use cryptographic module to initialize this key by performing one-way transformation over the value appended with the secret key as shown in Fig. 4, highlighted in red. This modification does not affect the remaining design of battcrypt as explained in [24].

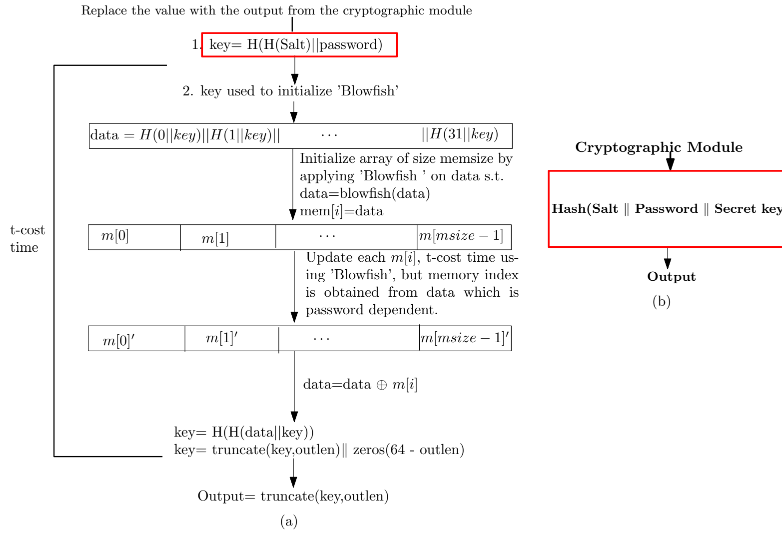


Fig. 4: (a)Overview of the design battcrypt (b) The cryptographic module implementation

Security analysis: At the first step of the algorithm battcrypt, it takes hash of the salt and then hash of the resulting value concatenated with password. We suggest to replace the computation with a single hash as shown in Fig. 4(b). The one-wayness is preserved by this and memory-hardness and other properties of the whole design remains the same. Hence the suggested modification does not affect the security.

– **Catena [11]** The algorithm Catena initializes the value x by performing hash on the concatenation of values tweak t, salt and password respectively. This value x is used to initialize the array of 2^{g_0} elements and g_0 increases by 1 at each following iterations as shown in Fig. 5(a). We can use cryptographic module to initialize this x as shown in Fig. 5(b). This modification does not affect the remaining design of Catena as explained in [11].

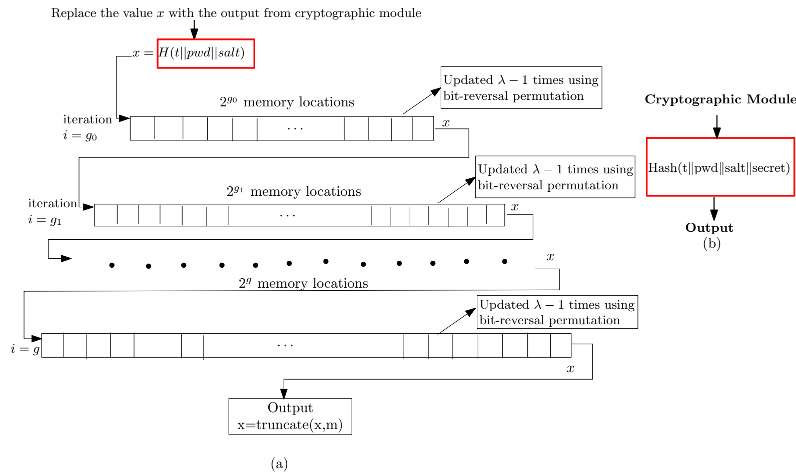


Fig. 5: (a)Overview of the design Catena (b) The cryptographic module implementation

Security analysis: Like the previous design we added one more parameter, secret key for the initial hash computation without making difference to the remaining design as shown in Fig 5. Therefore this modification does not affect the security claimed for the remaining design.

– **CENTRIFUGE [3]** The algorithm initializes the value seed by taking hash of salt and password and the length parameters as shown in Fig. 6. We can replace this seed value with the output of the cryptographic module which highlighted in red in Fig. 6. This modification does not affect the remaining design of CENTRIFUGE as explained in [3].

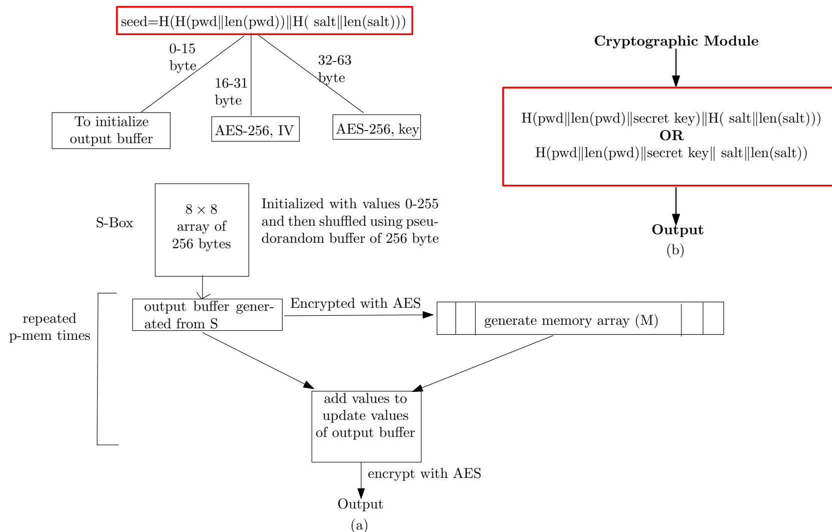


Fig. 6: (a)Overview of the design CENTRIFUGE (b) The cryptographic module implementation

Security analysis: Like the previous constructions CENTRIFUGE performs hash on the password and consumes the result for remaining phases, therefore suggested method does not affect the claimed security.

– **EARWORM [12]** The password and salt is required several times during the whole computation of EARWORM as shown in Fig 7(a). Apart from that the algorithm needs a secret value other than the password. This secret value is used to generate a huge array ARENA. Therefore this computation can not be performed in cryptographic module. To provide the security based on cryptographic module we can compute the hash (or HMAC as used in the algorithm) of the

password with the secret value inside the module and whenever password is required we can use this hash value. Using the hash will be a good alternative from storing the password and reusing it. Therefore EARWORM needs little modification of the design [12].

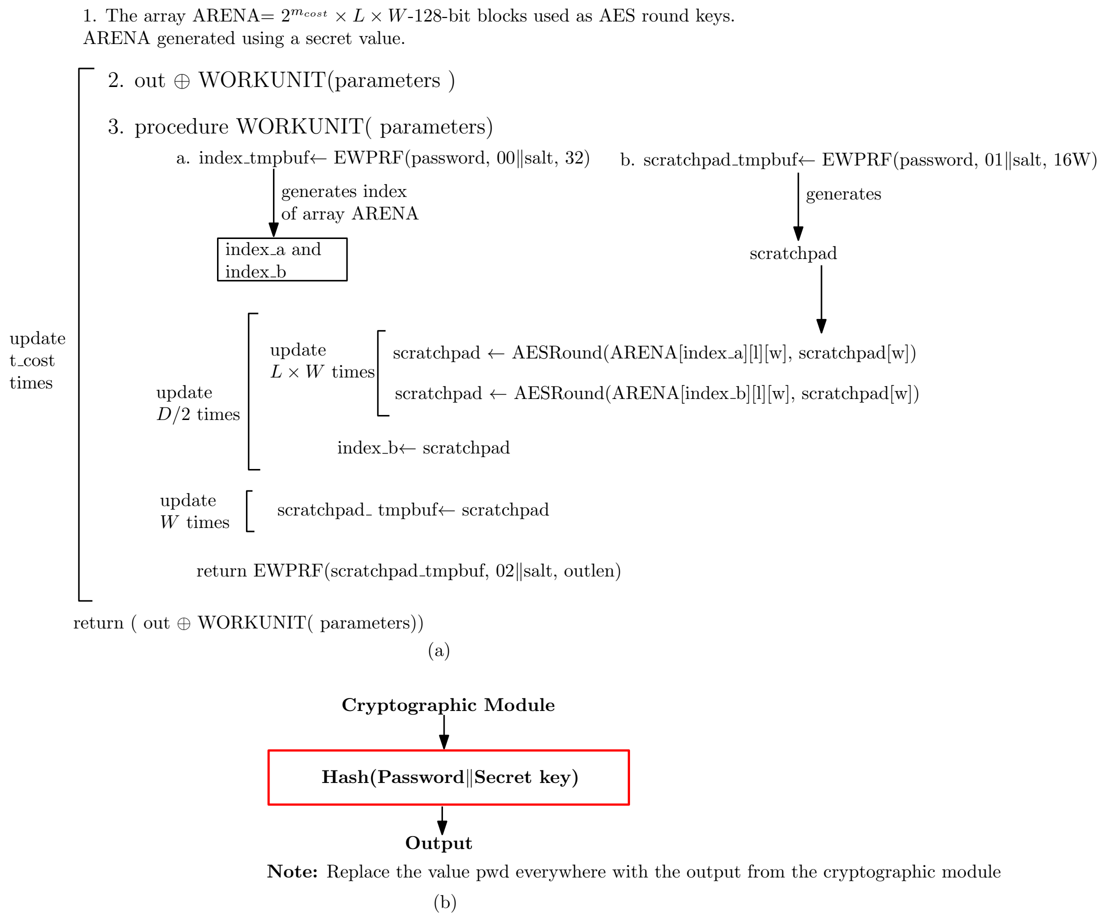


Fig. 7: (a)Overview of the design EARWORM (b) The cryptographic module implementation

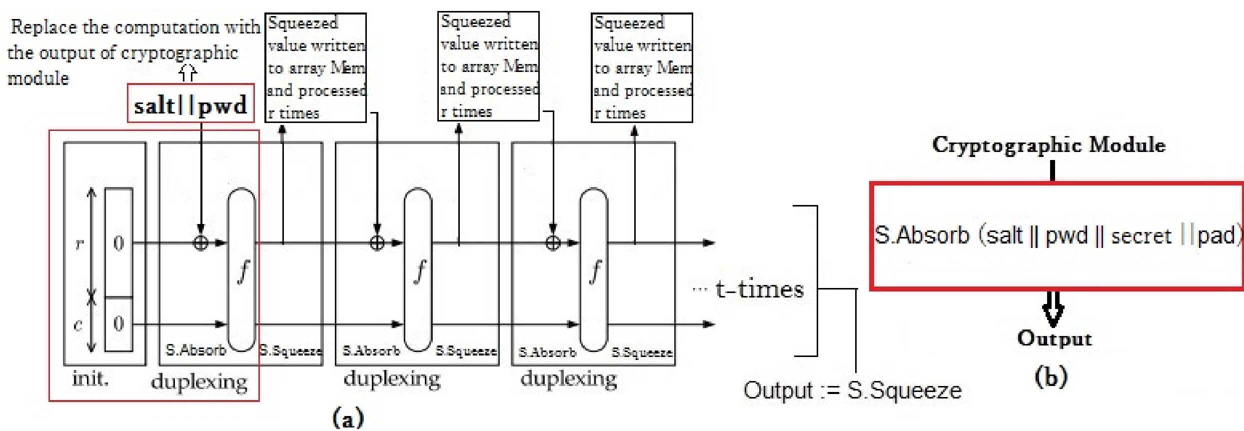


Fig. 8: (a)Overview of the design Gambit taken from [4], (b) The cryptographic module implementation

– **Gambit [20]** The algorithm is based on duplex-sponge construction and initial input of the sponge is the value salt concatenated with password. This initial computation with the value

password can be implemented in the cryptographic module as shown in Fig. 8 highlighted with red. This suggestion is due to preserve the sponge based approach of Gambit. For this approach we assume that the state size of the sponge is small enough to fit all required computations inside the low memory cryptographic module. This modification does not affect the remaining design of Gambit as explained in [20].

– **Lanarea DF [17]** The algorithm has three phases, the setup phase followed by the core phase and the key extraction phase. The setup phase initializes a matrix of $m_cost \times 16 \times 16$ byte which is updated $t_cost \times m_cost \times 16$ times at the core phase. The core phase processes the password and salt through a pseudorandom function and generates an array $(h_0, \cdots, h_{31})$ which determines the access pattern of the matrix. This array generation from password can be replaced with the output of the cryptographic module as shown in the Fig. 9 where F is the pseudorandom function used in the algorithm. Specifically the algorithm uses BLAKE2b and pseudo-random function for its implementation. As suggested before if the pseudo-random function is a block cipher then we suggest to use HMAC inside the cryptographic module to generate the hash, where key will be the secret key and hash will be the block cipher based hash. Therefore use of cryptographic module does not need modification in the remaining design as explained in [17].

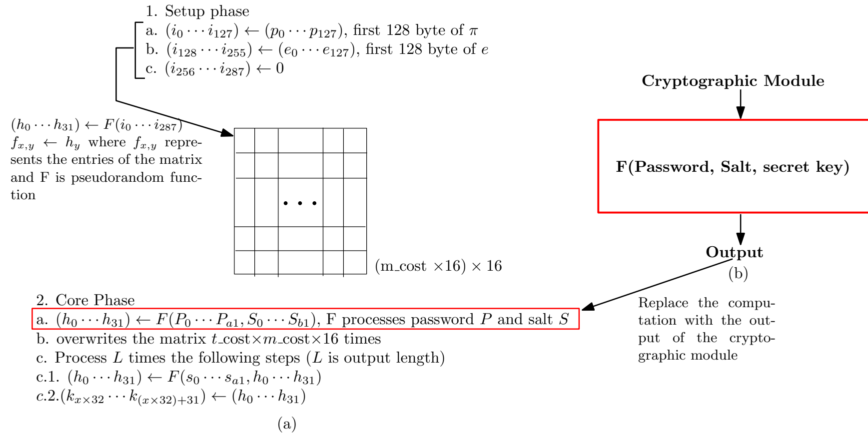


Fig. 9: (a)Overview of the design Lanarea DF (b) The cryptographic module implementation

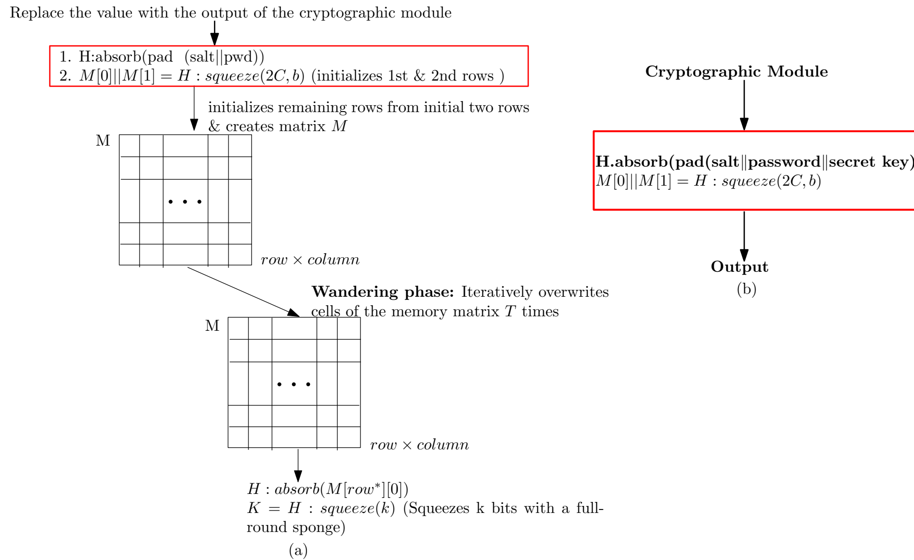


Fig. 10: (a)Overview of the design Lyra2 (b) The cryptographic module implementation

- **Lyra2 [14]** This is a duplex sponge based algorithm which initializes a matrix at the setup phase. The initial computation of this phase performs the sponge absorb with input salt and password. Like the Gambit design, we suggest to implement this initial computation with password to be implemented inside the cryptographic module as shown in Fig. 10(b), highlighted in red. This modification does not affect the remaining design of Lyra2 as explained in [14].
- **Makwa [21]** The algorithm Makwa is not memory demanding therefore it is possible to implement the whole design inside the cryptographic module. Even if we implement the whole design the required modification is to make the pre-hashing computation compulsory for the password (which is optional for the design) as shown in Fig. 11 highlighted in red. In case of complete implementation in cryptographic module, the output will be the output of the algorithm Makwa. Therefore this design as explained in [21] can easily support cryptographic module based approach.

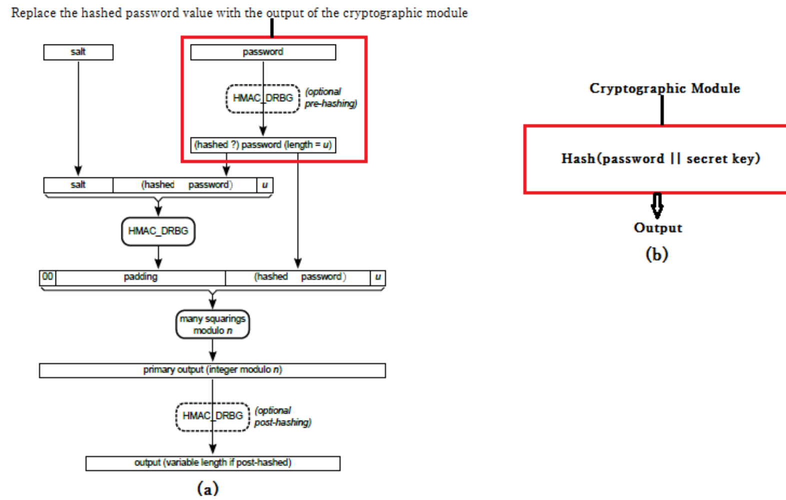


Fig. 11: (a)Overview of the design Makwa taken from [21], (b) The cryptographic module implementation

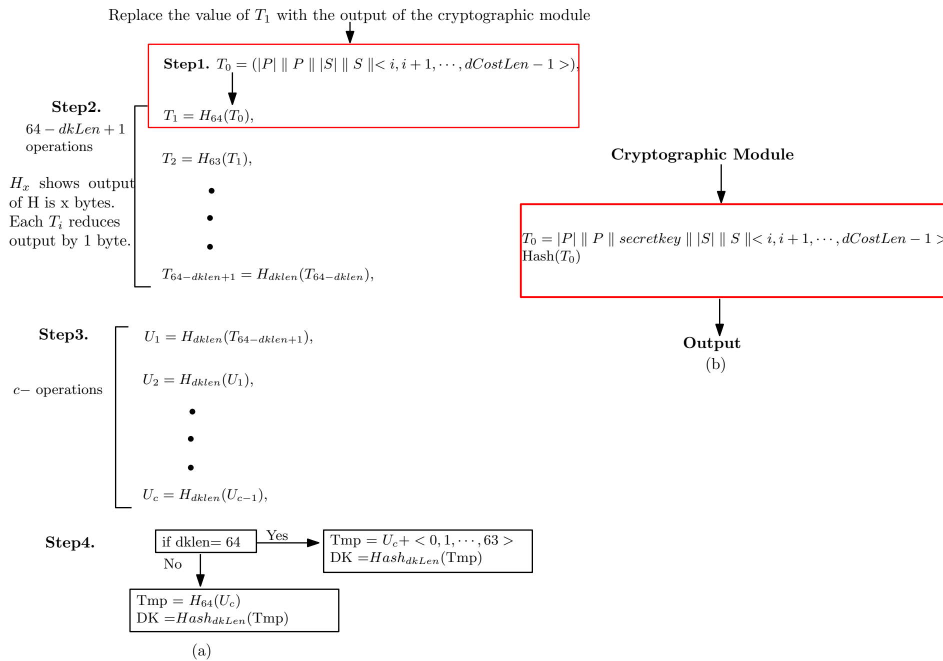


Fig. 12: (a)Overview of the design MCS_PHS where H_k denotes output length of hash is k byte (b) The cryptographic module implementation

– **MCS_PHS [16]** The algorithm implements iterative hash. It initializes the value T_0 by taking concatenation of password, password length, salt, salt length and other parameters as shown in Fig. 12(a). This T_0 is proposed to be a large value but it is the input of a hash function, therefore we can use cryptographic module to calculate T_0 and T_1. In this case we assume T_0 is small enough to fit in the low memory cryptographic module. The suggested implementation does not affect the remaining design of MCS_PHS as explained in [16].
– **Omega Crypt (ocrypt) [10]** The algorithm initializes the value Q using password and salt as shown in Fig. 13(a). The value Q is of length 771 byte. We can implement the initial three steps in cryptographic module as shown in Fig. 13 highlighted in red. Therefore cryptographic module based design does not require modification in the design of ocrypt as explained in [10]. In this case it may happen that size of Q is large with respect to the low memory cryptographic module, but the value 771 byte is not very large to resolve.

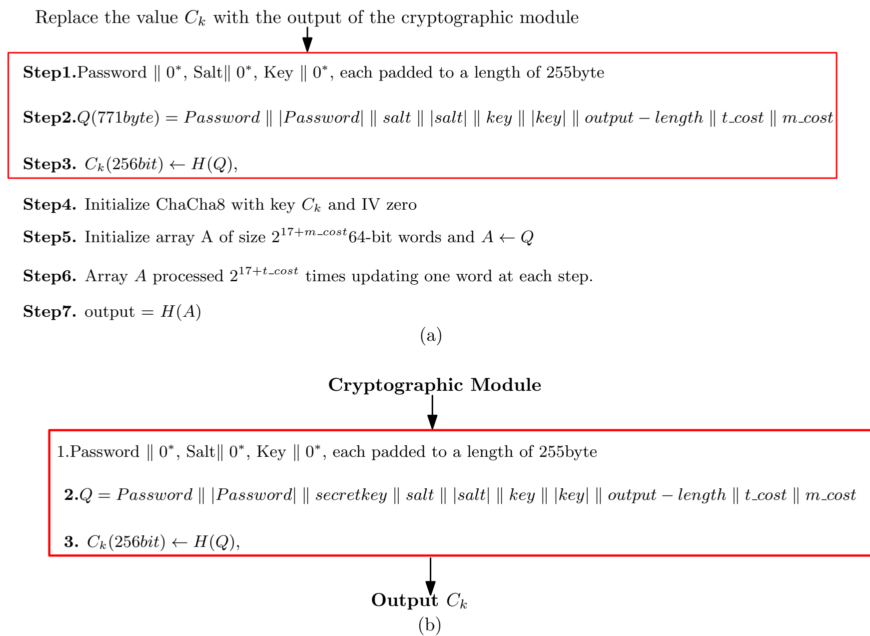


Fig. 13: (a)Overview of the design ocrypt (b) The cryptographic module implementation

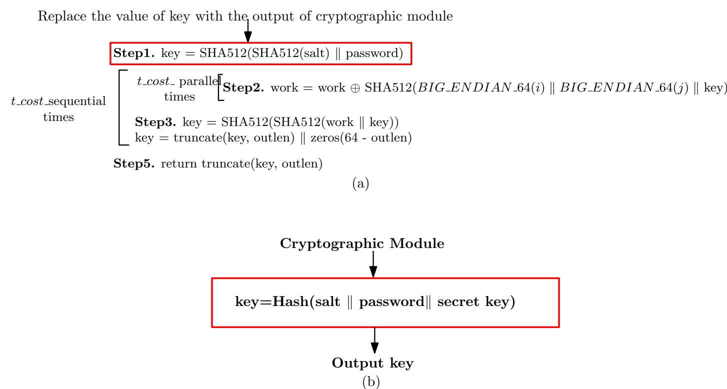


Fig. 14: (a)Overview of the design Parallel (b) The cryptographic module implementation

– **Parallel [25]** The algorithm initializes the value 'key' performing hash over password and salt as shown in Fig. 14. The overall design is not memory demanding. Therefore it is possible to

implement the whole algorithm in cryptographic module. We have shown in the diagram the implementation of the first hash computation in cryptographic module. Therefore cryptographic module based design does not require any modification in the design of Parallel as explained in [25].

- **PolyPassHash [6]** The algorithm provides a threshold cryptosystem based technique to protect password hashes where the password hash can be verified only if a threshold of passwords are known. It basically aims to prevent an attacker from efficiently cracking individual passwords from a stolen password database. Therefore in this work we are not providing the cryptographic module based approach for PolyPassHash [6].
- **POMELO [27]** The algorithm POMELO uses three different state updation functions that are used to update the state array which is first initialized to zero and then filled with password and salt values as shown in Fig. 15(a). To provide cryptographic module based approach we suggest to replace the password and salt values with the output of the cryptographic module as shown in Fig. 15(b) highlighted in red. This suggested modification may change the design requirement by the use of an additional cryptographic primitive which is not preferred. Therefore in this case we should be specific before selection of the function that can be implemented without affecting the overall design requirements. The suggested approach does not affect the remaining design of POMELO as explained in [27].

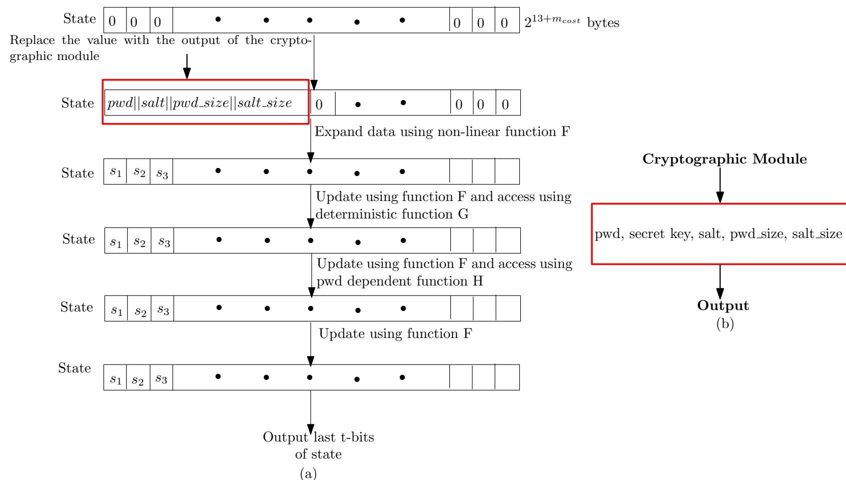


Fig. 15: (a)Overview of the design POMELO (b) The cryptographic module implementation

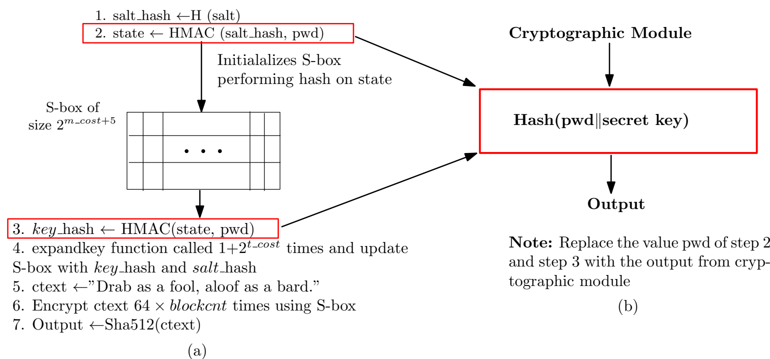


Fig. 16: (a)Overview of the design Pufferfish (b) The cryptographic module implementation

– **Pufferfish [13]** The algorithm initializes the S-box of size 2^{m_cost+5}512 bit processing the salt and the password as shown in Fig. 16(a). The value password is again used to generate the value key_hash. Therefore we suggest to implement the hash of the password to perform inside the cryptographic module and to replace the value password with this hashed value for the computation of the steps 2 and 3 as shown in Fig. 16. Therefore cryptographic module based design does not require modification in the design of Pufferfish as explained in [13].
– **Rig [7]** The initialization phase of the algorithm Rig computes the hash of the value x derived from password, salt and other parameters and produce the hash output α as shown in Fig. 17(a). We suggest to implement this initialization phase in cryptographic module and this approach does not require modification in the design of Rig as explained in [7].

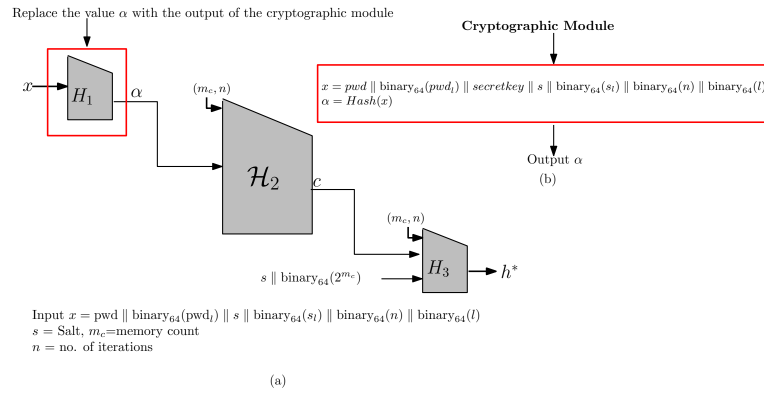


Fig. 17: (a)Overview of the design Rig (b) The cryptographic module implementation

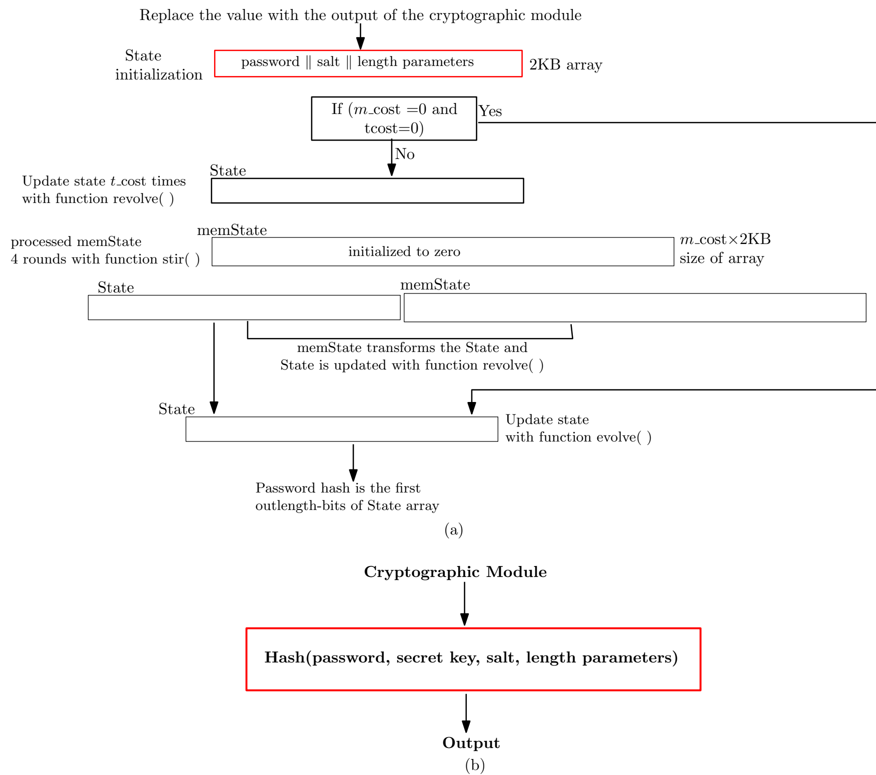


Fig. 18: (a)Overview of the design Schvrch (b) The cryptographic module implementation

– **Schvrch [26]** The algorithm initializes the state array of 2 KB with zero and then fill the state with password, salt and their length parameters and output length. The processing of the steps depends on the values of the m_cost and t_cost. If these values are greater than zero then the state array is updated t_cost time with function revolve otherwise update with function evolve. The whole procedure is shown in Fig. 18(a). For the cryptographic module based approach we suggest the state array to be initialized with the output of the cryptographic module instead of the plain-text values. In that case as suggested in [26] the array can be filled with other parameters other than the length of password but the resulting value should be small enough to be accommodated inside the low memory cryptographic module. The algorithm Schvrch defines different functions inside the design. Therefore it is required to put extra attention for the selection of the function inside the cryptographic module so that the suggested approach would not modify the requirement of the design. Apart from that the cryptographic module based approach does not affect the remaining design as explained in [26].
– **Tortuga [22]** The algorithm is based on Turtle design and specifically the permutation used in this construction is the Turtle algorithm. The steps of the algorithm are enumerated in Fig. 19(a). It uses password at step 2 and reuses at step 3 where both computations use different cryptographic primitives, Turtle cipher and Sponge. To provide the cryptographic module based approach we suggest to compute the hash of the password with the secret key in the module and to use the hashed value where ever password is required. This approach will be secure than using plain-text password repeatedly. Therefore our suggested cryptographic module based design does not require major modification in the design of Tortuga as explained in [22].

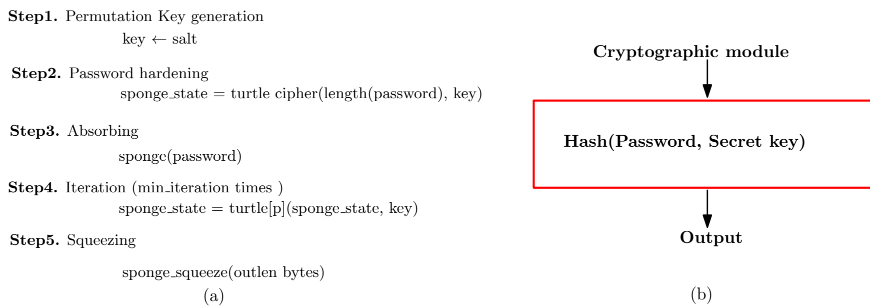


Fig. 19: (a)Overview of the design Tortuga (b) The cryptographic module implementation

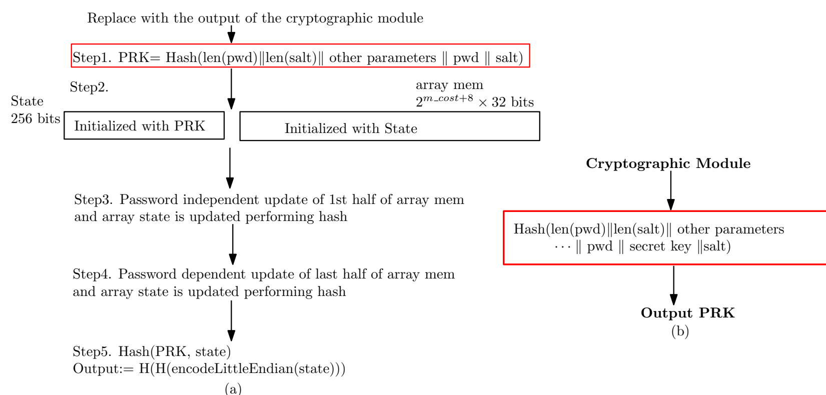


Fig. 20: (a)Overview of the design TwoCats (b) The cryptographic module implementation

– **TwoCats [8]** The PHC [2] submission TwoCats [8] include two variants, TwoCats and SkinnyCat. In this section we provide the overview of the common part of both these variants as our suggestion can easily satisfy both. The algorithm first computes a pseudo-random key PRK from password,salt and other parameters. It initializes a state array of size 256 bits and another array mem of length 2^{m_cost+8}. It uses a two-loop architecture, where the first loop is password independent and update first half of the mem array, and the second loop is password dependent which updates second half of the mem array. Both the loops update the value of the state array performing hash. Fig 20 shows that we can implement the generation of PRK inside the cryptographic module. Therefore cryptographic module based design does not require modification in the design of TwoCats as explained in [8].
– **Yarn [15]** The algorithm Yarn has five phases and the first phase of the algorithm uses password and generates the final state applying Blake2b as shown in Fig. 21(a). The second phase processes the resulting state and expands it to produce pseudo-random data for the subsequent phases. The third phase produces a large array initialized with pseudo-random data. The fourth phase calls AES primitive multiple times and the fifth phase produces the output. Therefore the first phase can be implemented in cryptographic module with a secret key and this approach does not require modification in the design of Yarn [15].

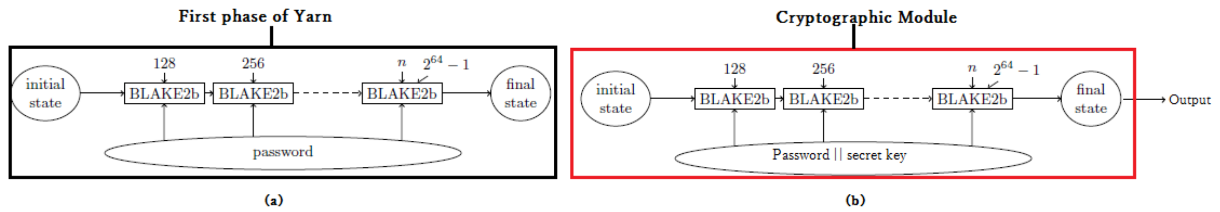


Fig. 21: (a)Overview of the first phase design Yarn taken from [15], (b) The cryptographic module implementation

Security analysis: The security of the algorithm is based on the security of BLAKE2b function and the AES function. At the first phase of the algorithm, it performs repeated hash(Blake2b) on the input password. We suggest to implement this phase inside cryptographic module which appends secret key with the password and then performs the usual computation. This modification enhances the security of password leakage and also preserves the one-wayness. The memory-hardness and other properties of the whole design remain the same. Hence the suggested modification enhance the overall security of the algorithm.

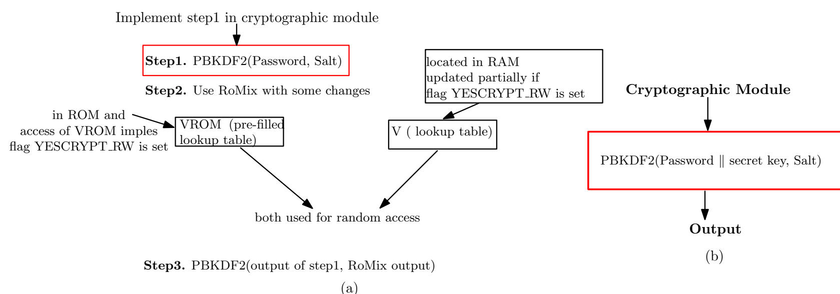


Fig. 22: (a)Overview of the design yescrypt (b) The cryptographic module implementation

– **yescrypt [19]** Yescrypt is based on the design Scrypt [18] and the significant difference is that the design of RoMix in Yescrypt supports an optional pre-filled read-only lookup table (VROM, implemented in ROM). If VROM is accessed then the YESCRYPT_RW flag is set, and the other random-read lookup table (V, located in RAM) is updated partially. Password is the input to the initial call of PBKDF2 and last call of PBKDF2 uses the results of this initial call. Therefore it is required to store the password dependent data in memory. We can implement the initial call to PBKDF2 in cryptographic module appending the secret key with the password as shown in Fig. 22 to enhance the security. Therefore cryptographic module based design does not require modification in the design of Yescrypt as explained in [19].

5.1 Categorization of the schemes

From the analysis of the submitted PHC designs in the last section with respect to their suitability for a crypto-module based implementation, we categorize the approaches taken as follows:

– **Category I:** Use of the secret key with password (and other parameters, if applicable) for the initial hash computation inside the cryptographic module.
– **Category II:** Replacement of the input plain-text password by the hash of the password with the secret key (and other parameters depending on the algorithm) which does not affect the design requirements (in terms of the use of the cryptographic primitive inside the cryptographic primitive).
– **Category III:** Replacement of the initial plain-text password input by the hash of the password with the secret key which may affect the design requirement (in terms of the use of cryptographic primitive inside the cryptographic module).
– **Category IV:** Use of cryptographic module for initial multiple line execution (e.g., repeated use of the cryptographic primitive) of the algorithm.

Following Table 1, lists the algorithms based on the above defined categories:

Table 1: Categorization of the algorithms based on the taken approaches for cryptographic module implementation.

Category I	**Category II**	**Category III**	**Category IV**
battcrypt	AntCrypt	Argon	Yarn
Catena	EARWORM	Lanarea DF	
CENTRIFUGE	MCS_PHS†	POMELO	
Gambit†	Omega Crypt†	Schvrch †	
Lyra2†	Pufferfish		
Makwa	Tortuga		
Parallel			
Rig			
TwoCats			
yescrypt			

†Assuming that the memory requirement for the suggested implementations are small enough to fit inside a low memory crypto-module.

6 Conclusions

In this paper, we provide a novel approach to enhance the security of the implementation of password hashing algorithms using a cryptographic module. We graphically explained how the designs of the PHC candidates fit in this architecture. Considering the emerging threats in the landscape of Cryptographic implementations, we believe our analysis will have significant impact towards the security of password hashing algorithms. We hope that secure implementations of password hashing algorithm would consider this cryptographic module based approach to enhance their security in appropriate use cases.

References

1. FIPS PUB 140-2, Security Requirements for Cryptographic Modules. Federal Information Processing Standards Publication (Supercedes FIPS PUB 140-1, 1994 January 11), 2001. http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf.
2. Password Hashing Competition (PHC), 2014. `https://password-hashing.net/index.html`.
3. Rafael Alvarez. CENTRIFUGE - A password hashing algorithm. Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/Centrifuge-v0.pdf.
4. Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*. Springer, 2011.
5. Alex Biryukov and Dmitry Khovratovich. ARGON v1: Password Hashing Scheme. Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/Argon-v1.pdf.
6. Justin Cappos. PolyPassHash: Protecting Passwords In The Event Of A Password File Disclosure. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/PolyPassHash-v0.pdf.
7. Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Rig: A simple, secure and flexible design for Password Hashing. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/RIG-v2.pdf.
8. Bill Cox. TwoCats (and SkinnyCat): A Compute Time and Sequential Memory Hard Password Hashing Scheme. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/TwoCats-v0.pdf.
9. Markus Dürmuth and Ralf Zimmermann. AntCrypt. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/AntCryptv0.pdf.
10. Brandon Enright. Omega Crypt (ocrypt). Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/OmegaCryptv0.pdf.
11. Christian Forler, Stefan Lucks, and Jakob Wenzel. The Catena Password-Scrambling Framework. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/Catena-v2.pdf.
12. Daniel Franke. The EARWORM Password Hashing Algorithm. Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/EARWORM-v0.pdf.
13. Jeremi M. Gosney. The Pufferfish Password Hashing Scheme. Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/Pufferfish-v0.pdf.
14. Marcos A. Simplicio Jr, Leonardo C. Almeida, Ewerton R. Andrade, Paulo C. F. dos Santos, and Paulo S. L. M. Barreto. The Lyra2 reference guide. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/Lyra2-v1.pdf.
15. Evgeny Kapun. Yarn password hashing function. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/Yarnv2.pdf.
16. Mikhail Maslennikov. Password Hashing Scheme MCS PHS. Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/MCS PHS-v2.pdf.
17. Haneef Mubarak. Lanarea DF. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/Lanarea-v0.pdf.
18. Colin Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCon*, 2009. `http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf`.
19. Alexander Peslyak. yescrypt - a Password Hashing Competition submission. Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/yescrypt-v0.pdf.
20. Krisztián Pintér. Gambit - A sponge based, memory hard key derivation function. Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/Gambit-v1.pdf.
21. Thomas Pornin. The MAKWA Password Hashing Function. Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/Makwa-v0.pdf.
22. Teath Sch. Tortuga - Password hashing based on the Turtle algorithm. Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/Tortuga-v0.pdf.
23. Robert R. Schaller. Moore's law: past, present, and future. *IEEE Spectrum*, June 1997.
24. Steve Thomas. battcrypt (Blowfish All The Things). Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/battcrypt-v0.pdf.
25. Steve Thomas. Parallel. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/Parallel-v0.pdf.
26. Rade Vuckovac. schvrch. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/Schvrch-v0.pdf.
27. Hongjun Wu. POMELO: A Password Hashing Algorithm. Submission to Password Hashing Competition (PHC), 2014. https://passwordhashing.net/submissions/specs/POMELO-v1.pdf.

On Password Guessing with GPUs and FPGAs

Markus Dürmuth, Thorsten Kranz

Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany

Abstract. Passwords are still by far the most widely used form of user authentication, for applications ranging from online banking or corporate network access to storage encryption. Password guessing thus poses a serious threat for a multitude of applications. Modern password hashes are specifically designed to slow down guessing attacks. However, having exact measures for the rate of password guessing against determined attackers is non-trivial but important for evaluating the security for many systems. Moreover, such information may be valuable for designing new password hashes, such as in the ongoing *password hashing competition* (PHC).
In this work, we investigate two popular password hashes, bcrypt and scrypt, with respect to implementations on non-standard computing platforms. Both functions were specifically designed to only allow slow-rate password derivation and, thus, guessing rates. We develop a methodology for fairly comparing different implementations of password hashes, and apply this methodology to our own implementation of scrypt on GPUs, as well as existing implementations of bcrypt and scrypt on GPUs and FPGAs.

Keywords: Password hashing, efficient implementations, bcrypt, scrypt, FPGAs, GPUs

1 Introduction

Passwords are still the most widely used form of user authentication on the Internet (and beyond), despite substantial effort to replace them. Thus, research to improve their security is necessary. One potential risk with authentication in general is that authentication data has to be stored on the login server, in a form that enables the login server to test for correctness of the provided credentials. The database of stored credentials is a high-profile target for an attacker, which is illustrated in recent years by a substantial number of databases leaked by attacks. Even worse, for storage encryption the encryption key, protected by the password using a KDF, is stored on the same machine as the encrypted data, and thus an even easier target. A leak of the password database is a major concern not only because the credentials for that particular site leak, and resetting all passwords for all users of a site in a short time span requires a significant effort. In addition, password re-use, i.e., using one password for more than one site, is

a frequent phenomenon to reduce the cognitive load of a user, causes a single leaked password to compromise a larger number of accounts.

In order to mitigate the adverse effects of password leaks, passwords are typically not stored in plain, but in hashed (and possibly salted) form, i.e., one stores

$$(s, h) = (salt, \mathsf{Hash}(pwd, salt))$$

for a randomly chosen value *salt*. Such a hashed password can easily be checked by recomputing the hash and comparing it to the stored value h. While a secure hash function cannot be inverted, i.e., directly computing the password *pwd* from (s, h) is infeasible in general, the mere fact that the server can verify the password gives rise to a so-called *offline guessing attack*. Here, an attacker produces a large number of password candidates $pwd_1, pwd_2, pwd_3, \ldots$, and verifies each candidate as described before. User-chosen passwords are well-known to be predictable on average [19, 37], so such an attack is likely to reveal a large fraction of the stored passwords, unless special precautions are taken.

A widely used method to defend against offline guessing attacks is using hash functions that are slow to evaluate. While cryptographic hash functions are designed to be fast to compute, password hashes are deliberately slow, often using iterated constructions to slow down an attacker. This, of course, also slows down the legitimate server, but the attacker is typically more substantially affected by the slow-down as he needs to evaluate the hash functions millions or billions times. Some well-known examples for password hashes are the classical descrypt [24], which dates back to the 1970s, md5crypt, sha256crypt/sha512crypt, PBKDF2 [18], bcrypt [31], and scrypt [30]. There is ongoing effort to design stronger password hashes, e.g., the password hashing competition [29].

Currently lacking is a thorough understanding of the resistance of those password hashes against attacks using non-standard computing devices, in particular FPGAs and GPUs. Understanding these issues is, however, crucial to decide which password hash should be used, and at what hardness settings.

In this work, we make several contributions towards this goal: First, we provide an implementation of scrypt on GPUs that supports arbitrary parameters, which is substantially faster than existing implementations; second, we determine "equivalent" parameter sets for password hashes to allow for a fair comparison; third, based on the equivalent parameter sets, existing implementations, and our implementation of scrypt, we draw a fair comparison between bcrypt and scrypt. In summary, we find that for fast parameters both bcrypt and scrypt offer about the same level of security, while for slow parameters scrypt offers more security, at the cost of increased memory consumption.

1.1 Related Work

Password security. Guessing attacks against passwords have a long history [2, 39, 22]. More recently, probabilistic context-free grammars [37] as well as Markov models [25, 5] have been used with great success for password guessing. Most

password cracking tools implement some form of mangling rules, some also support some form of Markov models, e.g., JtR and hashcat. An empirical study on the effectiveness of different attacks including those based on Markov-models can be found in [7]. If no salt is used in the password hash, *rainbow-tables* can be used to speed up the guessing step [15, 28] using precomputation. An implementation of rainbow-tables in hardware is studied in [23].

Closely related to the problem of password guessing is that of *estimating the strength of a password*. In early systems, password cracking was used to find weak passwords [24]. Since then, so-called pro-active password checkers are used to exclude weak passwords [2, 4]. However, most pro-active password checkers use relatively simple rule-sets to determine password strength, which have been shown to be a rather bad indicator of real-world password strength [36, 20, 6]. More recently, Schechter et al. [32] classified password strength by counting the number of times a certain password is present in the password database, and Markov models have been shown to be a very good predictor of password strength and can be implemented in a secure way [6].

Processing platforms for password cracking. Password cracking on general-purpose CPUs is widely used, and cleverly optimized implementations can achieve substantial speed-up compared to straight-forward implementations. Well-known examples for such "general purpose tools" are *John the Ripper* [17], as well as specialized tools such as TrueCrack [35] for TrueCrypt encrypted volumes. However, due to the versatility of their architecture, CPUs usually do not achieve an optimal *cost-performance ratio* for a *specific* application.

Modern graphics cards (GPUs) have evolved into computation platforms for universal computations. GPUs combine a large number of parallel processor cores which allow highly parallel applications using programming models such as OpenCL or CUDA. GPUs have proven very effective for password cracking, demonstrated by tools such as the Lightning Hash Cracker by ElcomSoft [9] or hashcat [33].

Special-purpose hardware usually provides significant savings in terms of costs and power consumption and at the same time provides a boost in performance time. This makes special-purpose hardware very attractive for cryptanalysis [13, 14, 10, 40]. With the goal of benchmarking a power-efficient password cracking approach, Malvoni et al. [21] provide several implementations of bcrypt on low-power devices, including an FPGA implementation. Similarly, Wiemer et al. [38] provide an FPGA implementation of bcrypt. In [8], the authors provided implementations of PBKDF2 using GPUs and an FPGA cluster, targeting TrueCrypt.

1.2 Outline

We describe the scrypt algorithm and our GPU implementation in Section 2, and briefly review the bcrypt algorithm and recent work on implementing bcrypt on FPGAs in Section 3. In Section 4 we present a framework for comparing password

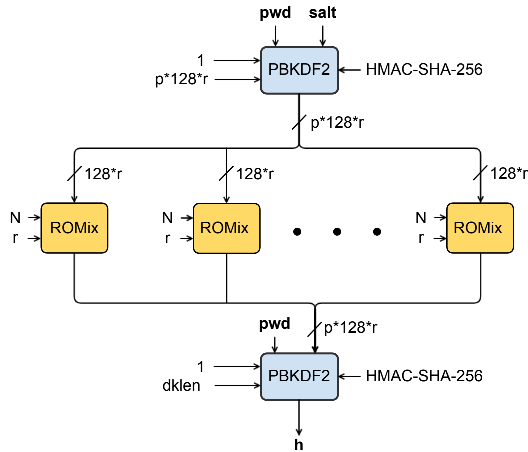


Fig. 1. Overview of scrypt. The data widths are given in bytes.

hashing functions and dedicated attacker platforms. We present the final results and a discussion in Section 5.

2 scrypt

In this section we describe the scrypt password hash and present a GPU implementation of scrypt for guessing passwords in parallel.

2.1 The scrypt password hash

The scrypt password hash [30] is a construction for a password hash which specifically counters attacks using custom hardware (the cost estimations specifically target ASIC designs, but the results hold, in principle, against FPGAs as well). The basic idea of the scrypt design is to force an attacker to use a large amount of memory, which results in large area for the memory cells and thus high cost of the ASICs.

Parameters. Scrypt takes as input a password *pwd* and a salt *salt*, and is parameterized with the desired output length *dklen* and three cost parameters: memory usage N, a block-size r, and a parallelism factor p. If $p > 1$ then basically p copies of the ROMix algorithm, which is described below, are executed independently (sequentially) of each other; the overall memory usage for ROMix is $128 \cdot r \cdot N$ bytes. Scrypt returns a hash value h of size *dklen* bytes.

Overall structure. The overall structure or scrypt consists of three main steps (see Figure 1).

Algorithm 2.1: ROMix

Data: B, r, N
Result: B'

```
1  X ← B;
2  for i ← 0 to N − 1 do
3  |  V[i] ← X;
4  |  X ← BlockMix(X);
5  end
6  for i ← 0 to N − 1 do
7  |  j ← Integerify(X) mod N;
8  |  X ← BlockMix(X ⊕ V[j]);
9  end
10 B' ← X;
```

(i) Initially, scrypt applies the PBKDF2 password hash [18] to the password and the salt, with an iteration count of 1, using HMAC-SHA-256 as MAC function, and producing $128 \cdot p \cdot r$ bytes of output. PBKDF2 is used to distribute the entropy from the password and salt and expand the input length, and presumably as a fail-safe mechanism to ensure the onewayness of the overall construction.

(ii) The output of this initial step is split into p chunks of $128 \cdot r$ bytes, and each chunk is fed into one of p parallel copies of the ROMix algorithm, which is the core part of the scrypt construction and described below.

(iii) Each invocation of the ROMix algorithm yields $128 \cdot r$ bytes of data, which are concatenated and fed into another instances of PBKDF2, together with the password, an iteration count of 1, and using HMAC-SHA-256, which finally produces the desired output of length *dklen*.

ROMix. The ROMix algorithm is the core of the construction. It operates on blocks of size $128 \cdot r$ bytes, and allocates an array V of N blocks as the main data structure. ROMix first fills the array V with pseudo-random data, and then pseudo-randomly accesses the data in the array to ensure the attacker is actually storing the data.

(i) First, ROMix fills the array V by repeatedly calling BlockMix, abbreviated H, which is basically a random permutation derived from the Salsa20/8 hash function (see below). The current state X is initialized with the input bytes (derived from the output of PBKDF2). Then, successively, H is applied to the state and the result written to successive array locations. The pseudo-code is shown in Algorithm 2.1 from line 2 to 5.

(ii) Second, the stored memory is accessed in a pseudo-random fashion in an attempt to ensure that all memory cells are stored. The initial state X is the final state of the previous step. The current state is interpreted as an index pointing to an element in the array V, that target value is XORed to the current state, and H is applied to form the next state. The pseudo-code is shown in Algorithm 2.1 from line 6 to 9

BlockMix. The BlockMix construction H operates on $2 \cdot r$ blocks of size 64 bytes each. It resembles the CBC mode of operation, with a final permutation of the block order. It's main use is apparently to widen the block size from the fixed 64 bytes of salsa20/8 to arbitrary width as required by the ROMix.

Recommended parameter values. Two sets of parameter choices are given [30] for typical use cases. For storage encryption on a local machine Percival proposes $N = 2^{20}, r = 8, p = 1$, which uses 1024 MB of memory. For remote server login he proposes $N = 2^{14}, r = 8, p = 1$, which uses 16 MB. Android since version 4.4 uses scrypt for storage encryption [11], with parameters $(N, r, p) = (2^{15}, 3, 1)$ [34].

2.2 GPU Programming

Over the years, GPUs have changed from being mere graphic processors to general purpose processing units, offering programming interfaces such as CUDA [27] for cards manufactured by NVIDIA.

GPUs execute code in so called kernels, which are functions that are executed by many threads in parallel. Each thread is member of a block of threads. All threads within a block have access to the same shared memory, which allows communication and synchronization between threads. During execution, blocks are assigned to Streaming Multiprocessors (SMs). An SM then schedules its pending blocks in chunks of 32 threads, called a warp, to its hardware, where each thread within a warp executes the same instruction. When threads in the same warp execute different instructions they are scheduled one after another (thread divergence). When threads are scheduled for high-latency memory instructions, the scheduler will execute additional warps while waiting for the memory access to finish, thus to a certain extend hiding the slow memory access.

Each thread has private *registers* and *local memory* which is, for example, used for *register spilling*. Threads from the same block can access the fast per-block *shared memory*, which can be used for inter-thread communication. All threads can access *global memory*, which is by far the largest memory, but also the slowest. There are some specialized memory regions, *constant memory* and *texture memory*, which are fast for specific access patterns.

NVIDIA's GTX 480 GPU [26] is a consumer-grade GPU which offers reasonable performance at an affordable price. It entered market in 2010 at the price of 499 dollars. A GTX 480 consists of 15 SMs with 32 computing cores each, i. e., the architecture provides 480 cores within a single GPU. Memory bandwidth is 177.4 GB/s. The cores are running at 1401 MHz and can reach a single-precision floating point performance (Peak) of up to 1345 GFLOPS. (For comparison: Intel's recent Core i7 980 CPUs running at 3.6 GHz are listed at 86 GFLOPS [16].) The GTX 480 offers 1536MB of global memory.

2.3 Implementing scrypt on CUDA

Our implementation performs a brute-force password search over a configurable character set. The implementation is fully on the GPU, the CPU is only responsible for enumerating the passwords, calling the GPU kernels, and comparing

the final results.(Parts of the implementation are inspired by the cudaMiner [3], a miner for the litecoin cryptocurrency, which uses scrypt with very low cost parameters $(N, r, p) = (1024, 1, 1)$ as proof-of-work.)

The CPU keeps track of the current progress and calls a new kernel with a starting point in the space of all passwords. It starts as many threads in parallel as are allowed by the available global memory, but always requires the number of threads to be a multiple of 32, as we are running 32 threads per warp. If the parameter p is greater than one, then those blocks will be executed one after another, which does not increase memory usage. In the remainder of the section we give some details about the GPU implementation.

PBKDF2. The implementation of PBKDF2 is rather straightforward. The iteration count of $c = 1$ is hard-coded. Overall, the operation is not time-critical.

BlockMix. The BlockMix operation operates on a state of $2 \cdot r$ words of size 64 bytes each, thus $128 \cdot r$ bytes in total, which are kept in the registers. For an efficient implementation of the mixing layer, in addition to the array holding the data, we implement an array with pointers that serve as index for the data; this way the mixing layer can be implemented by copying pointers (4 bytes) instead of blocks of data (64 bytes). The Salsa20/8 implementation follows the original proposal [1] including the optimization to eliminate the transpositions by alternatingly processing rows and columns.

ROMix. The implementation of ROMix has to take special care of the memory hierarchy in order to utilize the GPUs potential. The main concern is maximizing *memory throughput.* Global memory can be accessed in chunks of 32, 64, or 128 bytes, which must be aligned to a multiple of their size (naturally aligned). However, one thread can access a word of at most 16 bytes, so memory throughput is maximized when several threads access contiguous and aligned words; then memory access is called *coalesced.* Therefore, reading one block (64 bytes) is distributed across four threads reading words of 16 bytes, and, as each of the four threads needs to access a full block after all, they will cooperate four times to load all four blocks. Data is first read to shared memory by the cooperating threads, then copied to the registers by each thread individually.

Writing data to global memory follows the same rules. The data is first copied by the individual threads from registers to shared memory and then written to global memory by cooperating threads in an aligned and coalesced fashion.

Time-memory trade-off. Our implementation also provides the possibility to use a time-memory trade-off. By just storing every t-th data segment generated by the initial BlockMix iterations, only $1/t$ of the original amount of memory is needed. In return, every time a segment that was not stored is needed, it must be recomputed from the nearest previous segment. If t is increased, the probability of such a recomputation rises. So does the time needed for a recomputation since the there are on average more iterations to recompute.

Algorithm 3.1: bcrypt

```
  Input: cost, salt, pwd
  Output: hash
1 state ← EksBlowfishSetup(cost, salt, pwd);
2 ctext ← "OrpheanBeholderScryDoubt";
3 Repeat (64) begin
4 |  ctext ← EncryptECB(state, ctext);
5 end
6 return Concatenate(cost, salt, ctext);
```

Algorithm 3.2: EksBlowfishSetup

```
  Input: cost, salt, pwd
  Output: state
1 state ← InitState();
2 state ← ExpandKey(state, salt, pwd);
3 Repeat (2^cost) begin
4 |  state ← ExpandKey(state, 0, salt);
5 |  state ← ExpandKey(state, 0, pwd);
6 end
7 return state;
```

3 bcrypt

The second password hash we consider is the bcrypt hash function.

3.1 The bcrypt password hash

Provos and Mazières published the bcrypt hash function [31] in 1999, which, at its core, is a cost-parameterized, modified version of the blowfish algorithm. The key concepts are a tunable cost parameter and a constantly modified moderately large (4KByte) block of memory. Bcrypt is used as the default password hash in OpenBSD since version 2.1 [31]. Additionally, it is the default password hash in current versions of Ruby on Rails and PHP.

Parameters. Bcrypt uses the input parameters *cost*, *salt*, and *key*. The number of executed loop iterations is exponential in the *cost* parameter, cf. Algorithm 3.2. The algorithm uses a 128-bit *salt* to derive a 192-bit password hash from a *key* of up to 56 bytes.

Design. Bcrypt is structured in two phases. First, `EksBlowfishSetup` initializes the internal bcrypt-state. Afterwards, Algorithm 3.1 repeatedly encrypts a magic value using this state. The resulting cipher text is then concatenated with the cost and salt and returned as the hash. While the encryption itself is as efficient as the original Blowfish encryption, most of the time is spent in the `EksBlowfishSetup` algorithm.

The `EncryptECB` encryption is effectively a blowfish encryption. Within its standard 16-round Feistel network, the SBoxes and subkeys are determined by the current *state* and the plaintext is encrypted in 64-bit blocks.

The `EksBlowfishSetup` algorithm is a modified version of the blowfish key schedule. It computes a state, which consists of 18 32 bit subkeys and four SBoxes – each 256×32 bit in size – which are later used in the encryption process. The state is initially filled with the digits of π and a modified version of the blowfish keyschedule is performed. After xoring the key to the subkeys, it successively uses the current state as SBoxes and subkeys to encrypt blocks of the current state and update the state. In this process, the function `ExpandKey` computes 521 blowfish encryptions. If the salt is fixed to zero, one call to `ExpandKey` resembles the standard blowfish key schedule.

Recommended parameter values. Provos and Mazières originally proposed to use a cost parameter of six for normal user passwords, while using eight for administrator passwords.

3.2 Implementations of bcrypt on FPGAs

While general-purpose hardware, i. e., CPUs, offers a wide variety of instructions for all kinds of programs and algorithms, usually, only a few are important for a specific task. More importantly, the generic structure and design might impose restrictions and become cumbersome, i. e., when registers are too small or memory access times becomes a bottleneck. Reconfigurable hardware like Field-Programmable Gate Arrays (FPGAs) and special-purpose hardware like Application-Specific Integrated Circuits (ASICs) are more specialized and dedicated to a single task. An FPGA consists of a large area of programmable logic resources (the fabric), e. g., lookup tables, shift registers, multiplexers and storage elements, and a fixed amount of dedicated hardware modules, e. g., memory cores (BRAM), digital signal processing units or even PowerPCs, and can be specialized for a given task.

Recently, two groups presented implementations of bcrypt on FPGAs. The latest work is by Wiemer et al. [38], who present an implementation of bcrypt on Xilinx FPGAs from the low power consumption and low cost segment. Their platform is the zedboard, more precisely the Zynq-7000 XC7Z020 FPGA. The Zynq-7000 persists mainly of a dual-core ARM Cortex A9 CPU and an Aritx-7. The zedboard allows easy access to the logic inside the FPGA fabric via direct memory access and provides several interfaces, e. g., AXI4, AXI4-Stream, AXI4-Lite or Xillybus. These cores come with drivers for embedded Linux kernels and thus offer an easy way of accessing custom logic from a higher abstraction layer. Their design has a LUT consumption of $2,777$ LUTs per (quad-)core and uses 13 BRAMs. Including a simple logic for generating password candidates for a brute-force guessing attack, they were able to fit 10 quad-core designs on a single FPGA, which runs at a maximum clock frequency of 100MHz.

The other work by Malvoni et al. [21] reported a rate of 4571 passwords per second for a cost parameter of 5 on the zedboard. Due to unstable behavior,

they could not fully implement there design idea of 56 bcrypt instance and had to reduce this number to 28. Therefore, the simulated their design on the larger Zynq-7045 and reported 7044 passwords per second as the expected result for a stable behavior. Additionally, they reported a theoretical rate of 8112 passwords per second which they derived from the performance for a cost parameter of 12.

4 Methodology

Next, we present the methodology that we use to compare different algorithms on different platforms.

4.1 Basic idea

In this work we consider offline guessing attacks, and consequently the hashrate, i. e., how fast an attacker can verify its password guesses, is the critical factor. The effect of a password hash is to slow down the attacker's verification of a password guess. Slower password hashes will usually slow down both the (honest) verification of a password, as well as the attacker. However, an attacker is not bound to use the same implementation as the (honest) verification server, and so he may utilize optimizations that the legitimate verifier is not able to implement; in particular can the adversary use different hardware platforms much more easily than the verification server.

Thus, it is important to consider the ratio between the following two runtimes: first, the runtime of the normal (optimized for server use) implementation on typical server CPUs, and second, the runtime for a password on an attacker's implementation on comparable hardware of his choice. Here, the defender chooses the algorithm and parameters to be used, while the attacker can choose a hardware platform and has certain optimization techniques that the defender cannot use. As we want to compare different password hashing algorithms attacked on different platforms, we need to derive *reasonably equivalent parameters* for the different password hashes. Thus, we start by measuring the runtime of the algorithms on different PCs – which differ in the amount of processors as well as architecture and available memory – and derive comparable algorithm-parameter pairs.

4.2 Derivation of equivalent parameters

To determine the "equivalent" parameter sets for the different schemes, we run a series of tests on different CPUs and compare runtimes. We use implementations that target password checking by legitimate servers (i. e., that check one password at a time). Thus, we call two parameter sets of two algorithms "equivalent" if the legitimate server that checks the passwords needs the same runtime to do so in both cases.

We used the following implementations: For *PBKDF2*, we used the implementation in the OpenSSL library calling `PKCS5_PBKDF2_HMAC()` with `SHA512`.

For *bcrypt*, we used a version available from the Openwall website (`http://www.openwall.com/crypt/`), which was compiled on the target system gcc and compiler flags `-O3 -fomit-frame-pointer -funroll-loops`. For *scrypt*, we use our own implementation in C, as the original implementation is packaged into a larger project. The runtimes were comparable to those published by Percival [30].

Table 8 in Appendix B lists the platforms we used for the parameter derivation. We utilized different CPUs, with an emphasis on server CPUs, and measured runtimes for each of them. Appendix B gives the full measurement results. As there is no single system we can optimize for but are interested in general statements, we take the average runtime over all CPUs we tested. Note that the runtimes were, despite the wide variance of CPUs, grouped together relatively closely, the worst-case being a factor of two between the fastest and the slowest CPU, and in general much lower.

To investigate reasonable parameters and their resulting runtimes, one must ask for the actual size of parameters used in real-world applications. First, we need to note the this strongly depends on the application scenario. In an interactive login scenario the server must be able to quickly response to the user who tries to authenticate with a password. The situation is different if we consider key derivation for storage encryption, where longer delays are acceptable. (But note that the delay time is not the only bound for a practical implementation. Also extensive memory usage may hinder a server from choosing according parameters.) In light of these differences in the security requirements for password hashing, we make a comparison across a wide range of parameters and desired runtimes.

We give four classes of parameters, for targeted runtimes of (approximately) 1ms, 10ms, 100ms, 1000ms. Percival [30] states 100 ms as an upper bound on the delay for interactive login. For storage encryption, the acceptable runtime is higher and may extend slightly higher than 1000ms. But note that the parameters used for scrypt in Android since version 4.4 [11] for storage encryption (namely $(N, r, p) = (2^{15}, 3, 1)$ [34]) yields moderate running times (around 100ms on server CPUs, but higher on typical mobile devices).

Both bcrypt and scrypt offer a relatively coarse control over the runtime (incrementing the hardness parameter by one approximately doubles the execution time), thus no parameter will match exactly the target time. Therefore, we interpolate the parameters from the measured values, to more accurately model the desired runtimes and making the comparison fair. This means we have to interpolate the runtimes for the attacking implementations in the same way.

The (interpolated) equivalent parameters are listed in Table 1, the detailed measurements are listen in Table 9, 10, and 11 in Appendix B.

4.3 Comparing Different Platforms

For comparing the ration between the runtime on the legitimate server and the attacker, we also need a method to compare attacks using different hardware platforms.

Algorithm (Parameter)		Target (CPU) runtime			
		1ms	10ms	100ms	1000ms
PBKDF2	(Iteration count)	313	3 138	31 347	313 925
bcrypt	(Cost)	3.69	7.03	10.3	13.6
scrypt	(log N (r=8, p=1))	6.93	10.3	13.7	16.9

Table 1. "Equivalent" parameters for several target runtimes for PBKDF2, bcrypt, and scrypt.

An attack scales linearly with invested resources, mainly cost of the equipment and energy consumption, and thus we have to take their influence into account. (In addition, one can consider development cost, but here we will assume that implementations are already available. While GPU programming is quite similar to CPU programming and thus generally quicker, code development for FPGAs is substantially different and usually requires more time, and thus cost.) This leads to a time-cost trade-off, which is affected by the amount of devices the attacker uses in parallel to increase the hashrate, the costs per device and the power costs.

Generally speaking, equipment cost is in favor of the graphic cards, as GPUs are a consumer product that is sold in large quantities. Also, older versions usually receive a discount, making them more cost-effective. Interestingly, FPGA vendors use a different strategy: with the release of a new product line, the price of the old family stays roughly unchanged, while the new version is offered with a small discount to make the consumers switch away from the abandoned hardware platform. In terms of power consumption, reconfigurable hardware is by far more effective than GPUs. We will consider equipment cost and energy consumption for the different devices when comparing those implementations.

5 Results

Finally, we present and compare the hash rates of different implementations.

5.1 Comparing with oclHashcat

Before giving a more detailed comparison of different platforms, we start with an evaluation of our implementation of scrypt (given in Section 2.3) with existing implementations of scrypt, in particular with the oclHashcat [33] implementation. The scrypt algorithm is supported by oclHashcat starting with version 1.30, released in August 2014. We used the latest version at the time of writing, oclHashcat v1.31. To the best of our knowledge, oclHashcat is the only implementation of scrypt on GPUs allowing for a full variable choice of parameters. The litecoin miner cudaMiner [3], as well as other mining software we are aware of, only implement fixed parameter values (in particular $(N, r, p) = (1024, 1, 1)$

	$(2^{12}, 8, 1)$	$(2^{12}, 4, 1)$	$(2^{12}, 1, 1)$	$(2^{17}, 8, 1)$	$(2^{17}, 4, 1)$	$(2^{17}, 1, 1)$
oclHashcat v1.31	19.86	72.58	280.38	0.10	0.26	1.31
Our implementation	287.61	1171.82	27748.27	0.38	2.15	83.08

Table 2. Selected scrypt hashrates from oclHashcat and our GPU implementation.

which are the standard litecoin parameters), or partially fixed parameter values (in particular $(N, r, p) = (N, 1, 1)$ which are the parameters for some other scrypt-based altcoins). These constraints allow for deeper optimization techniques.

Both our implementation as well as oclHashcat run on a single NVIDIA Geforce GTX 480 card. Selected hashrates are listed in Table 2, more comprehensive measurements can be found in Appendix A. We can see that our implementation outperforms oclHashcat by a factor of 10 to 100 for moderate values of $N < 2^{14}$, which drops to approximately 4 for higher $N \approx 2^{17}$ and $r = 8$. We cannot investigate why oclHashcat is slower, as it is closed source software.

5.2 Measuring Hashrates

We use the following platforms in the comparison:

- Our GPU-based scrypt implementation is run on an NVIDIA Geforce GTX 480. Prices for GPUs have a substantial variability over time, being influenced by competing products, customer demand and releases of newer cards. The GTX 480 was released at a price of $499 in the first quarter of 2010 and dropped to around $310 in the third quarter of 2011. For the subsequent discussion, we estimate a reasonable price at $350. We assume an attacker mounts three graphics cards on a single motherboard, a setup which was empirically found to offer a good balance [12]. A suitable motherboard is estimated at $200, and a suitable 1600W power supply we estimate at $300. Overall, a machine with 3 GPUs will cost approx. $1550, and the average price per GPU (including peripherals) is approx. $517. The GTX 480 has been benchmarked with up to 430W under full load, even though the TDP is given as 250W only.
- The same setup was used to measure the speed of oclHashcat's implementation of bcrypt on GPUs.
- The bcrypt implementation from [38] runs on the zedboard, cf. Section 3. The zedboard is currently available for approx. $319, and has a power consumption of 4.2W.

Table 3 shows the results of the implementations for the derived parameter sets. We can see a couple of interesting points already from this basic data. First, we see that bcrypt is faster on the zedboard than it is on a GTX 480, despite the latter being more expensive and more energy consuming. This is a common observation, that specialized hardware implementations are faster and more efficient. Second, our scrypt implementation scales reasonably well

	Target (CPU) runtime			
	1ms	10ms	100ms	1000ms
bcrypt				
– zedboard	9 230 H/s	916.25 H/s	98.77 H/s	9.93 H/s
– GTX 480	2 868 H/s	319.37 H/s	33.73 H/s	2.71 H/s
scrypt				
– GTX 480	42 650 H/s	2 333 H/s	49.06 H/s	0.37 H/s
	(t=1)	(t=2)	(t=8)	(t=4)

Table 3. Hashrates of attacking implementations.

		Target (CPU) runtime			
	HW cost	1ms	10ms	100ms	1000ms
bcrypt					
– zedboard	$319	28.93 H/$s	2.87 H/$s	0.31 H/$s	0.03 H/$s
– GTX 480	$517	5.55 H/$s	0.62 H/$s	0.07 H/$s	0.01 H/$s
scrypt					
– GTX 480	$517	82.50 H/$s	4.51 H/$s	0.09 H/$s	0.00 H/$s
		(t=1)	(t=2)	(t=8)	(t=4)

Table 4. Hashes per dollar-second for attacking implementations.

		Target (CPU) runtime			
	Energy	1ms	10ms	100ms	1000ms
bcrypt					
– zedboard	4.2 W	2 198 H/Ws	218.15 H/Ws	23.52 H/Ws	2.36 H/Ws
– GTX 480	430 W	6.67 H/Ws	0.74 H/Ws	0.08 H/Ws	0.01 H/Ws
scrypt					
– GTX 480	430 W	99.19 H/Ws	5.43 H/Ws	0.11 H/Ws	0.00 H/Ws
		(t=1)	(t=2)	(t=8)	(t=4)

Table 5. Hashes per watt-second for attacking implementations.

	Cost		Target (CPU) runtime			
	HW	Energy	1ms	10ms	100ms	1000ms
bcrypt						
– zedboard	$319	$7.41	28.3 H/$s	2.81 H/$s	0.303 H/$s	0.0304 H/$s
– GTX 480	$517	$759	2.25 H/$s	0.250 H/$s	0.0264 H/$s	0.00212 H/$s
scrypt						
– GTX 480	$517	$759	33.4 H/$s	1.83 H/$s	0.0384 H/$s	0.000287 H/$s
			(t=1)	(t=2)	(t=8)	(t=4)

Table 6. Hashes per dollar-second taking into account total cost for two years.

between the parameter classes for 1ms and 100ms, but then the pressure from memory consumption becomes too large and speed drops substantially. Also, as expected, with increasing runtime (and thus memory consumption) higher trade-off parameters become optimal, as they trade memory consumption for computational load (except for the highest class of 1000ms). Third, even though the hashrates for different platforms are not directly comparable, we can already see that, while the hashrates for bcrypt scale almost linearly with the CPU runtime, scrypt scales much worse, which is caused by the increased memory consumption.

5.3 Comparison taking cost into account

The comparison in the previous section has the advantage that, by using equivalent parameters for the different password hashes, we obtain a fair comparison between the different password hashes. However, results from different hardware platforms, most notably GPUs and FPGAs, are still hardly comparable, as both have fundamentally different characteristics.

There are two main parameters that are different for the two different platforms: first, hardware cost is different. On the one hand, most GPUs are consumer products and are sold in huge quantities, while FPGA boards that are easily usable by consumers are a niche product. On the other hand, GPUs are equipped with additional units that are irrelevant for our specific application, and FPGAs can fully be configured to the task at hand. Second, the energy cost is fundamentally different. FPGA designs only implement the logic required to compute the desired functionality, which means that most overhead can be avoided. This results in a decreased number of switching logic gates and thus reduced power consumption. There are other factors one could consider, e.g, development cost, but in this discussion we will concentrate on energy cost and hardware cost.

Table 4 shows the results taking into account the hardware cost of the individual devices, as estimated in the previous section. Data is given in hashes per second and dollar (H/$s). What we observe is that the influence of the price is smaller than we expected, as the prices for a GPU including (shared) host PC and an FPGA that sits on a development board are quite similar. Note here that the devices used are one example only, and by using other GPUs or FPGAs the prices are somewhat variable. Also, the price of a development board such as the zedboard is substantially higher than a single FPGA only, but note that an FPGA always will need some additional hardware to facilitate its operation, e.g., to ensure network connectivity. Overall, while all these prices come with some uncertainty, the overall picture of the comparison should be quite reliable.

Table 5 shows the results taking into account the energy costs of the different architectures. We listed the approximate power consumption of the GTX 480 and the zedboard as 430 Watt and 4.2 Watt, respectively, which already illustrates the fundamental difference between the two. Note that, again, these estimates are somewhat variable and depend on the specific FPGA and GPU used, as well as the exact load put on the device. What we see is that the zedboard is clearly

superior to the GTX 480 in this metric due to the significantly higher power consumption.

Finally, we aim to bring these different results together and determine a total cost, combining the energy and hardware cost for a duration of two years. We estimate energy cost at a price of 10.08 cents per kWh (average retail price of electricity in the United States in 2013, according to the U.S. Energy Information Administration[1]). The results are shown in Table 6. Basically what this table shows is that scrypt can be attacked rather efficiently for low parameters with the GTX 480. The attack is even stronger than the bcrypt attack with the zedboard. But for higher parameters the FPGA attack on bcrypt is more efficient than the GPU attack on scrypt.

Finally, we can say that we were surprised by the fast operation of scrypt on GPUs for moderate parameters. In scenarios where FPGA-based crackers are unavailable (e.g., for the casual password cracker), or for applications where power cost is not a critical factor, bcrypt is more resistant to password cracking for parameters up to the 100ms class. When we additionally consider FPGA-based attacks, the picture changes, as bcrypt can be computed quite efficiently on FPGAs, in particular in terms of energy cost. Except for low parameters from the 1ms class (where GPUs against scrypt are more efficient FPGAs against bcrypt in terms of hardware cost as well as total cost for two years), scrypt is harder to attack, based on the implementations we are considering. This advantage is almost exclusively caused by the energy cost (energy cost for a single GTX 480 is approximately a factor 100 higher than for a single zedboard).

6 Conclusion

In this work we have provided a methodology for comparing different password hashes on varying platforms. We have applied this methodology to bcrypt and scrypt implementations on GPUs and FPGAs, including our own implementation of scrypt on GPUs, which may be of independent interest. Taking into account all the attacking implementations we have considered, bcrypt and scrypt offer comparable strength for smaller parameters (that take about 1ms to 10ms on the defenders side), while scrypt is stronger for larger parameters.

References

1. D. J. Bernstein. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs*, volume 4986 of *LNCS*, pages 84–97. Springer, 2008.
2. M. Bishop and D. V. Klein. Improving system security via proactive password checking. *Computers & Security*, 14(3):233–249, 1995.
3. Bitcoin Forum. cudaMiner – a new litecoin mining application. `http://bitcointalk.org/index.php?topic=167229.0`.

[1] http://www.eia.gov/electricity/data/browser

4. W. E. Burr, D. F. Dodson, and W. T. Polk. Electronic Authentication Guideline: NIST Special Publication 800-63, 2006.
5. C. Castelluccia, A. Chaabane, M. Dürmuth, and D. Perito. Omen: An improved password cracker leveraging personal information. Available as arXiv:1304.6584, 2013.
6. C. Castelluccia, M. Dürmuth, and D. Perito. Adaptive Password-Strength Meters from Markov Models. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*. The Internet Society, 2012.
7. M. Dell'Amico, M. Pietro, and Y. Roudier. Password Strength: An Empirical Analysis. In *INFOCOM '10: Proceedings of 29th Conference on Computer Communications*. IEEE, 2010.
8. M. Dürmuth, T. Güneysu, M. Kasper, C. Paar, T. Yalçin, and R. Zimmermann. Evaluation of Standardized Password-Based Key Derivation against Parallel Processing Platforms. In *Proc. European Symposiumon Research in Computer Security (ESORICS)*, volume 7459 of *Lecture Notes in Computer Science*, pages 716–733. Springer, 2012.
9. ElcomSoft. Lightning Hash Cracker, Nov 2011. `http://www.elcomsoft.com/lhc.html`.
10. T. Gendrullis, M. Novotný, and A. Rupp. A real-world attack breaking a5/1 within hours. *IACR Cryptology ePrint Archive*, 2008:147, 2008.
11. Google Inc. – Open Handset Alliance. Android KitKat. `https://developer.android.com/about/versions/kitkat.html`.
12. J. Gosney. Password cracking hpc. Presentation at Passwords12 Conference. Online at `http://passwords12.at.ifi.uio.no/Jeremi_Gosney_Password_Cracking_HPC_Passwords12.pdf`, 2012.
13. T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, November 2008.
14. T. Güneysu, C. Paar, G. Pfeiffer, and M. Schimmler. Enhancing COPACOBANA for advanced applications in cryptography and cryptanalysis. In *Proceedings of the Conference on Field Programmable Logic and Applications (FPL 2008)*, pages 675–678, 2008.
15. M. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.
16. Intel. Intel Core i7-900 Desktop Processor Series, 2011. `http://download.intel.com/support/processors/corei7/sb/core_i7-900_d.pdf`.
17. John the Ripper. `http://www.openwall.com/john/`.
18. B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, Sept. 2000. `http://tools.ietf.org/html/rfc2898`.
19. D. V. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *Proc. USENIX UNIX Security Workshop*, 1990.
20. S. Komanduri, R. Shay, P. G. Kelley, M. L. Mazurek, L. Bauer, N. Christin, L. F. Cranor, and S. Egelman. Of Passwords and People: Measuring the Effect of Password-Composition Policies. In *CHI 2011: Conference on Human Factors in Computing Systems*, 2011.
21. K. Malvoni, S. Designer, and J. Knezovic. Are your passwords safe: Energy-efficient bcrypt cracking with low-cost parallel hardware. In *Proc. 8th USENIX Conference on Offensive Technologies*, WOOT'14. USENIX Association, 2014.
22. S. Marechal. Advances in password cracking. *Journal in Computer Virology*, 4(1):73–81, 2008.

23. N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. Time-Memory Trade-Off Attack on FPGA Platforms: UNIX Password Cracking. In *Reconfigurable Computing: Architectures and Applications*, volume 3985 of *Lecture Notes in Computer Science*, pages 323–334. Springer Berlin / Heidelberg, 2006.
24. R. Morris and K. Thompson. Password Security: A Case History. *Commun. ACM*, 22(11):594–597, Nov. 1979.
25. A. Narayanan and V. Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *Proc. 12th ACM conference on Computer and communications security*, pages 364–372, New York, NY, USA, 2005. ACM.
26. Nvidia. NVIDIA GeForce GTX 400 GPU Datasheet, 2010. `http://www.nvidia.com/docs/IO/90025/GTX_480_470_Web_Datasheet_Final.pdf`.
27. Nvidia. CUDA Developer Zone (Website), 2011. `http://developer.nvidia.com/category/zone/cuda-zone`.
28. P. Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Proc. of Advances in Cryptology (CRYPTO 2003)*, volume 2729 of *LNCS*, pages 617–630. Springer, 2003.
29. Password hashing competition. `https://password-hashing.net/`.
30. C. Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. Presentation at BSDCan'09. Available online at `http://www.tarsnap.com/scrypt/scrypt.pdf`, 2009.
31. N. Provos and D. Mazières. A Future-Adaptable Password Scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91. USENIX, 1999.
32. S. Schechter, C. Herley, and M. Mitzenmacher. Popularity is everything: a new approach to protecting passwords from statistical-guessing attacks. In *Proceedings of the 5th USENIX conference on Hot topics in security*, pages 1–8. USENIX Association, 2010.
33. J. Steube. oclhashcat, 2014. `http://hashcat.net/oclhashcat/`.
34. P. Teufl, A. G. Fitzek, D. Hein, A. Marsalek, A. Oprisnik, and T. Zefferer. Android encryption systems. In *International Conference on Privacy & Security in Mobile Systems*, 2014. in press.
35. TrueCrack. Online at `http://code.google.com/p/truecrack/`.
36. M. Weir, S. Aggarwal, M. Collins, and H. Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS 2010)*, pages 162–175. ACM, 2010.
37. M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek. Password Cracking Using Probabilistic Context-Free Grammars. In *IEEE Symposium on Security and Privacy*, pages 391–405. IEEE Computer Society, 2009.
38. F. Wiemer and R. Zimmermann. High-speed implementation of bcrypt password search using special-purpose hardware. In *Proc. International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2014.
39. T. Wu. A real-world analysis of kerberos password security. In *Network and Distributed System Security Symposium*, 1999.
40. R. Zimmermann, T. Güneysu, and C. Paar. High-Performance Integer Factoring with Reconfigurable Devices. In *FPL*, pages 83–88. IEEE, 2010.

A Full runtime listings for hashcat

	$(2^{11}, 8, 1)$	$(2^{12}, 8, 1)$	$(2^{13}, 8, 1)$	$(2^{14}, 8, 1)$	$(2^{15}, 8, 1)$	$(2^{16}, 8, 1)$	$(2^{17}, 8, 1)$
oclHashcat v1.31	71.80	19.86	5.34	1.35	0.62	0.26	0.10
Our implementation	909.75	287.61	85.12	26.30	4.92	0.93	0.38
	$(2^{11}, 4, 1)$	$(2^{12}, 4, 1)$	$(2^{13}, 4, 1)$	$(2^{14}, 4, 1)$	$(2^{15}, 4, 1)$	$(2^{16}, 4, 1)$	$(2^{17}, 4, 1)$
oclHashcat v1.31	146.29	72.58	20.41	5.05	1.33	0.62	0.26
Our implementation	3253.26	1171.82	365.22	108.52	31.86	7.94	2.15
	$(2^{11}, 2, 1)$	$(2^{12}, 2, 1)$	$(2^{13}, 2, 1)$	$(2^{14}, 2, 1)$	$(2^{15}, 2, 1)$	$(2^{16}, 2, 1)$	$(2^{17}, 2, 1)$
oclHashcat v1.31	285.87	143.72	71.34	20.32	5.26	1.35	0.61
Our implementation	17887.84	5311.56	1967.22	660.43	191.21	54.62	15.64
	$(2^{11}, 1, 1)$	$(2^{12}, 1, 1)$	$(2^{13}, 1, 1)$	$(2^{14}, 1, 1)$	$(2^{15}, 1, 1)$	$(2^{16}, 1, 1)$	$(2^{17}, 1, 1)$
oclHashcat v1.31	567.47	280.38	140.80	70.47	19.53	5.25	1.31
Our implementation	55694.39	27748.27	9179.61	3380.60	1075.55	313.39	83.08

Table 7. Comparison of hashrates for our implementation and oclHashcat v1.31.

B Full runtime listings for the benchmark CPUs

	CPU	CPU Launch	OS	Type
CPU1	Intel Core i5-2400, 3.1 GHz	Q1'11	Win7/CygWin	Desktop
CPU2	AMD Opteron 6276, 2.3GHz	Q1'13	CentOS 6.2	Cluster
CPU3	Intel Core i5-2520M CPU, 2.50GHz	Q1'11	Win/CygWin	Laptop
CPU4	Intel Xeon E5540, 2.53GHz	Q1'09	Ubuntu 12.04	Server
CPU5	Intel Xeon E3-1220 V2, 3.10GHz	Q2'11	Fedora 19	Server

Table 8. Hardware used to measure CPU runtimes.

Iteration count	250	500	1k	2k	4k	8k	16k	32k	64k	128k	256k	512k	1024k
CPU1	0.63	1.27	2.53	5.05	10.06	20.22	40.44	80.92	162.43	323.12	646.22	1 287.80	2 603.60
CPU2	1.03	2.05	4.11	8.22	16.51	33.00	66.16	131.69	262.81	524.50	1 050.75	2 119.00	4 217.01
CPU3	1.04	2.09	4.18	8.36	16.62	33.38	66.74	133.53	266.86	532.74	1 063.15	2 135.65	4 263.50
CPU4	0.79	1.57	3.15	6.29	12.59	25.19	50.38	100.78	201.44	402.87	805.25	1 612.00	3 222.00
CPU5	0.50	0.99	1.98	3.97	7.93	15.87	31.72	63.47	127.00	254.00	507.75	1 015.00	2 035.00
Average	0.80	1.59	3.19	6.38	12.74	25.53	51.09	102.08	204.11	407.45	814.62	1 633.89	3 268.22

Table 9. Running times of **PBKDF2** with HMAC and SHA-512 on CPUs (in *ms*).

N (r=8, p=1)	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}
CPU1	1.13	2.29	4.53	8.95	17.81	35.49	71.57	143.47	287.04	578.58	1 159.10	2 324.40	4 664.31
CPU2	1.02	1.90	3.62	7.09	14.07	28.24	56.75	115.53	237.31	474.25	959.25	1 937.50	3 911.00
CPU3	1.19	2.42	4.74	9.45	18.88	37.98	76.00	152.93	307.32	616.79	1 235.92	2 478.85	4 957.59
CPU4	1.16	2.21	4.32	8.53	16.97	33.91	68.34	137.13	287.44	577.37	1 157.25	2 317.00	4 643.00
CPU5	0.72	1.38	2.70	5.33	10.59	21.12	42.56	86.00	173.50	346.87	694.50	1 388.50	2 780.00
Average	1.04	2.04	3.98	7.87	15.66	31.35	63.04	127.01	258.52	518.77	1 041.20	2 089.25	4 191.18

Table 10. Running times of **scrypt** on CPUs (in *ms*).

Cost	4	5	6	7	8	9	10	11	12	13	14	15	16
CPU1	1.28	2.40	4.63	9.12	18.04	35.93	72.15	143.96	288.21	576.61	1 148.18	2 307.25	4 612.91
CPU2	1.67	3.11	6.00	11.79	23.38	46.63	92.81	185.31	370.38	740.50	1 480.25	2 960.00	5 918.00
CPU3	1.35	2.52	4.87	9.55	18.92	37.76	75.20	150.39	301.18	600.41	1 203.52	2 410.25	4 842.19
CPU4	1.48	2.76	5.34	10.47	20.75	41.32	82.44	164.75	329.13	658.25	1 316.50	2 631.00	5 263.99
CPU5	1.04	1.95	3.77	7.41	14.68	29.23	58.34	116.50	232.94	465.50	931.75	1 862.50	3 728.00
Average	1.36	2.55	4.92	9.67	19.15	38.17	76.19	152.18	304.37	608.25	1 216.04	2 434.20	4 873.02

Table 11. Running times of **bcrypt** on CPUs (in *ms*).

C Full runtime listings for different trade-off parameters for scrypt

log(N) (r=8,p=1)	6.93	10.35	13.66	16.94
No Tradeoff	42,650.62	2,053.98	30.97	-
Tradeoff = 2	22,153.09	2,333.11	39.84	-
Tradeoff = 4	15,870.75	1,552.01	45.02	0.37
Tradeoff = 8	9,548.42	949.73	49.06	0.23

Table 12. Obtained hashrates for scrypt. Computed as interpolation of the nearest measurements.

SlidePIN:
Slide-Based PIN Entry Mechanism on a Smartphone

Huiping Sun, Shuaiying Guo, Ke Wang, Nan Qin, and Zhong Chen

School of Software & Microelectronics, Peking University
No.5 Yiheyuan Road, Haidian District, Beijing, P.R.China
sunhp@ieee.org,soft87@126.com,{k3wang,qinn,chz}@pku.edu.cn

Abstract. SlidePIN is a PIN entry mechanism based on slide input method combined with a random numeric keypad. As slide input method ensures higher usability and security, a random numeric keypad introduced, at a slight cost of usability, conspicuously enhances the security of SlidePIN. As an indirect entry mechanism, SlidePIN keeps users away from additional computation or memory burden. Theoretic analysis and experiments show that SlidePIN performs effectively against one-time shoulder surfing attack and better than 4-digit PIN mechanism against multi-time shoulder surfing attack.

Keywords: Smartphone; PIN; Shoulder Surfing; SlidePIN

1 Introduction

Smartphones have globally become the most popular communication tool nowadays which gradually change our life and work. Great amounts of private as well as business information are stored in our smartphones. As the foundation of smartphone security against unwanted access, unlocking mechanisms get more indispensable. 4-digit PIN (Personal Identification Numbers) mechanism that is used most widely asks users to input the PIN directly, which makes it vulnerable to shoulder surfing attacks [1, 2].

A wide range of research tried to resist shoulder surfing attacks to minimize relevant security threats. Two main mechanisms, invisible entry mechanism [5–10] and indirect entry mechanism [11–18], are proposed and developed. However, these mechanisms need either additional hardware or extra computation to ensure security and usability, which motivates us to design a better unlocking mechanism.

The Word-Gesture Keyboard [3, 4] concept was proposed by Montgomery in 1982. The idea suggests using slide gesture to input English words on a touch screen with a soft keyboard. Some implementations of this technology have been developed into products put into market by some companies (ShapeWriter, Swype, TouchPal and etc).

Inspired by Word-Gesture Keyboard, we propose SlidePIN as an indirect entry mechanism with a random numeric keypad and slide input method. SlidePIN

inserts users' 4-digit PIN into a slide sequence. Even if attackers captured the slide sequence, extracting the 4-digit PIN from the sequence would still be a conundrum.

To the best of our knowledge, SlidePIN is the first 4-digit PIN entry mechanism that combines slide input method with a random numeric keypad. Generating PIN indirectly based on user's gestures, SlidePIN can effectively prevent one-time shoulder surfing attack and withstand multi-time shoulder surfing attack with better performance, compared with 4-digit PIN.

In following sections of this paper, the concept of SlidePIN will be introduced first. Afterward, theoretical and experimental analysis will be provided to illustrate the improvement on security of SlidePIN and slight compromise of usability, compared with 4-digit PIN.

2 Related Works

In order to withstand shoulder surfing attacks, current entry methods mainly adopt invisible entry mechanism and indirect entry mechanism.

Invisible Entry Mechanism. An invisible entry mechanism [5–10] is to utilize special human-computer interaction methods to implement the input process of the PIN or password. Typical examples are eye tracking [5], tactile sensor [6], pressure sensor [10], vibration sensors [7], back-of-device interaction [8] and physical block [9], etc. It is difficult for the attacker to visually capture the interactions between computers and humans, thus this mechanism has great capability to resist shoulder surfing attack. However, an invisible entry mechanism is not suitable on smartphones because of its dependency on additional hardware followed by extra deployment costs.

Indirect Entry Mechanism. An indirect entry mechanism [11–18] is built with a human-computable challenge-response mechanism. Colors [11, 15], symbols [12, 14] or directions [13, 16–18] are added as additional authentication factors based on a traditional keypad or even the layout of a keypad, as a challenge, which is rebuilt with these factors. Generally in these methods, users use PINs kept in mind to compute the response against the corresponding challenge. The computation always involves collection attribution, color or symbol matching, orientation comparison, table looking up, etc. Users input the response instead of the PIN and it is hard to extract the PIN from the response, which leads to an improvement on security of the mechanism. Additionally, less dependency on auxiliary devices or hardware makes it compatible on smartphones, but additional computation or memory burden for users inevitably reduces its usability.

As an indirect entry mechanism, SlidePIN depends on no additional hardware or devices, which hence avoids extra deployment costs. What is more, with slide input method and a random numeric keypad introduced, SlidePIN accordingly

generates the input response when a user slides to unlock. No additional computation or memory burden is imposed to users, which makes security potentially improve as users of SlidePIN slide to unlock their smartphone habitually and conveniently.

3 SlidePIN Concept

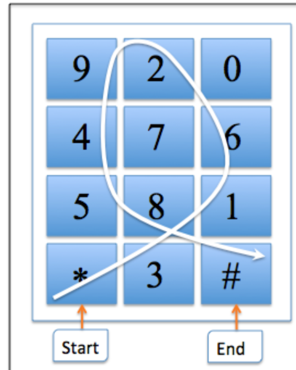


Fig. 1. SlidePIN Concept

Design. SlidePIN (Slide-based PIN entry) is one of sliding PIN entry methods which is inspired by the concept of slide input English words introduced in Word-Gesture Keyboard [4]. Two key mechanisms are introduced into our design compared with 4-digit PIN: *1)* Click input is replaced by slide input. *2)* The fixed layout keypad is substituted by the randomly distributed keypad.

We adopt these two mechanisms mainly because:

1. **Slide input is faster.** It has been proved [20, 21] that click input method is more difficult to use, which suggests that slide input will help improve input efficiency.
2. **Slide input is more secure.** As an indirect entry mechanism, a user's PIN is concealed in a slide sequence, which makes SlidePIN more secure than traditional click input method against shoulder surfing attack.
3. **Input with a random numeric keypad is more secure.** A random numeric keypad helps SlidePIN perform better against replay attack.

Implementation. There are two implementation phases of SlidePIN as follows:

- Setup Phase: Like 4-digit PIN, the user chooses 4 ordered digit numbers as the master secret, usually referred to a PIN, between him/her and the smartphone.

- Unlocking Phase: The user touches and slides over a keypad passing all digits of the PIN in order. As Fig.1 shows, the keypad is a random numeric keypad. In addition, the sliding process should be started from '*' and ended up with '#'. If the slide sequence contains the PIN as its subsequence, the authentication will be passed.

For instance, '1245' as a user's PIN, the user needs to slide and generate a trace starting from '*' and ending with '#' and subsequence '1245' has to be contained in an exact order in the slide sequence. As Fig.1 shows, '*381629458#' is one of the valid slide sequences which can unlock the smartphone.

4 Theoretical Analysis

4.1 Model Definition.

Firstly, we will introduce some concepts.

- **PIN(P):** A PIN is an ordered sequence consisting of integers from 0 to 9. $\{p_1, p_2...p_n\}$, $p_i \in [0, 9]$, $i \in [0, n]$. In this paper, we set n=4.
- **Layout(L):** L is the distribution of keys in a keypad as Fig.1. A new layout is generated randomly every time before a user inputs.
- **Trajectory(T):** A trajectory is the trace that is formed when the user slides.
- **Sequence(S):** S represents the keys a user slide over by fingers, which forms an ordered sequence $\{s_1, s_2...s_m\}$. S starts from '*', containing PIN (p_1, p_2, p_3, p_4) and inserted numbers $(i_1, i_2, i_3, i_4, i_5\ )$ and then end with '#', as Fig.2 shows.

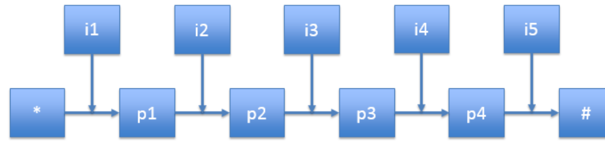


Fig. 2. Model of Input Sequence

- **Slide Map Function** f**:** f defines the process that user creates the sequence by sliding on the layout according to PIN:

$$f(L \times P) \rightarrow S \tag{1}$$

- **Attack Function** f^{-1}**:** f^{-1} defines the reverse process or the attack process based on keypad layouts and trajectories captured by an attacker who aims at obtaining the PIN. Since a specific sequence is determined by a specific random keypad layout and a specific trajectory, the relationship can be described as follows:

$$L \times T \rightarrow S \tag{2}$$

Accordingly, the attack function is defined as:

$$f^{-1}(L \times T) \rightarrow f^{-1}(S) \rightarrow P \tag{3}$$

4.2 Sequence Length Analysis.

The sequence length has a direct influence on security and usability of SlidePIN. Longer slide sequences make users' PIN more secure but the usability is impaired. Shorter ones lead to higher usability yet lower security. Therefore, sequence length analysis needs to be conducted to searching for a good tradeoff between security and usability. First, we estimate the distance between keys of keypad. Afterward, we estimate the distribution of PIN and finally the sequence length.

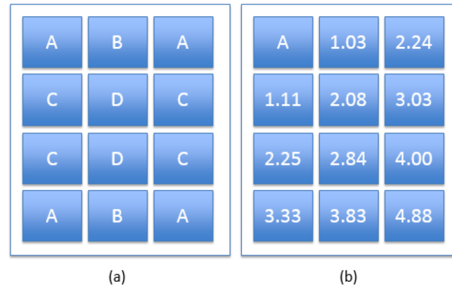


Fig. 3. Category of Keypad and Distances Between Category A and Other Keys

Estimate of Distance between Keys. Keys on a SlidePIN keypad can be divided into four categories (A, B, C, D) and distances between keys in category A and others can be calculated as showed in the Fig.3 [1]. Then the average distance between key A and others is $D(A) = (1.03+2.24+1.11+2.08+3.03+2.25+2.84+4.00+3.33+3.83+4.88)/11 \approx 2.78$

Distances between keys in category B, C, D and others, as Fig.4 shows, are D(B)=2.39, D(C)=2.25, D(D)=1.87, which can be calculated using the same method.

As the keypad showed in Fig.1, the sequence in category A is started at * and ended with #. As defined in the model, the average distance is D(A) at position i1 and i5. Other than that, the average distance among 10 numbers is $D_{avg} = (D(A) \times 2 + D(B) \times 2 + D(C) \times 4 + D(D) \times 2)/10 \approx 2.31$, which is the distances at the position i2, i3, i4.

[1] The distances are calculated based on data from sequence length experiments.

(a)

1.03	B	1.03
2.08	1.11	2.08
2.84	2.25	2.84
3.83	3.33	3.83

(b)

1.11	2.08	3.03
C	1.03	2.24
1.11	2.08	3.03
2.25	2.84	4.00

(c)

2.08	1.11	2.08
1.03	D	1.03
2.08	1.11	2.08
2.84	2.25	2.84

Fig. 4. Distances Between Category B,C,D and Other Keys

Estimate of PIN Distribution. the Layout of a SlidePIN keypad can be divided into area Z1, Z2, Z3. According to the method in the previous section, we can separately calculate the average distance when all of the digits in a PIN are in the area of Z1, Z2, Z3 , which is D(Z1)=8.08, D(Z2)=10.82, D(Z3)=11.55.[2]

We could calculate probability of P(Z3)=1, if PIN in the area of Z3. When PIN in the area of Z2, $P(Z2) = \frac{7}{10} \times \frac{6}{9} \times \frac{5}{8} \times \frac{4}{7} = \frac{1}{6}$ and $P(Z1) = \frac{4}{10} \times \frac{3}{9} \times \frac{2}{8} \times \frac{1}{7} = \frac{1}{210}$ when PIN in the third and fourth row except '*' and '#'.

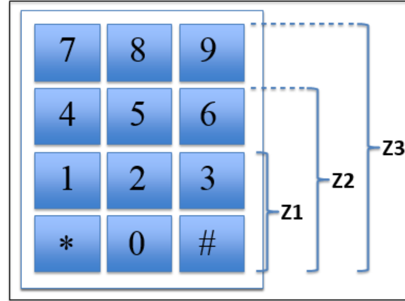


Fig. 5. Distribution of PIN

$Z3'$ refers to the case in which at least one digit of PIN is distributed in the first row. Then the probability of $Z3', P(Z3') = P(Z3) - P(Z2) = 1 - \frac{1}{6} = \frac{5}{6} \approx 0.8333$. Similarly, $Z2'$ represents one or more digits of PIN are distributed in the second row and $P(Z2') = P(Z2) - P(Z1) = \frac{17}{105} \approx 0.1619$. $P(Z1') = P(Z1) = \frac{1}{210} \approx 0.0048$. Therefore, the probability of $Z3', Z2', Z1'$ is approximate to 0.8333, 0.1619, 0.0048.

In case of Z3', the average length of sequence, $D(Z3') = D(Z2) \times P(Z2) + D(Z3') \times P(Z3') = 11.39$. Similarly, D(Z2')=10.52, D(Z1')=8.08.

[2]The calculating process is exemplified in following section, Estimate of Sequence Length.

Estimate of Sequence Length.

Mean Value of Sequence Length. As we have demonstrated above, the average of sequence length when a PIN can be distributed randomly on the entire keypad is $(D(A) \times 2 - 1) + D_{avg} \times 3 = 11.55$.[3]

Lower Threshold of Sequence Length. If the lower threshold is set to 8, approximate to the average length of Z1, there are only $\binom{8}{4} = 70$ possible PINs in an 8-bit-length sequence, making SlidePIN subject to guessing attack. However, longer sequences lead to lower usability and SlidePIN cannot stand the impairment on usability by 10 or more. Thus we set the lower threshold to 9 as a ideal tradeoff with 126 possibilities to resist guessing attack.

Upper Threshold of Sequence Length. Excessively long sequences are vulnerable to replay attack and with lower usability. As to replay attack, the attacker has averagely at least 8 (D(Z1) = 8.08) target characters to input. According to the previous section, a target character generally leads to about 2 input characters (D(A)=2.78, D(B)=2.39, D(C)=2.25, D(D)=1.87). To further improve capability of resisting replay attack, we conservatively use 1.87 as the factor to calculate: $8.08 \times 1.87 \approx 15.11$. Therefore, we set the upper threshold to 15, which is a reasonable upper threshold ensuring both security and usability. An illustration will be provided later in the experiment section.

4.3 Security Analysis

Threat Model. Shoulder surfing attack refers to using direct observation techniques, such as looking over someone's shoulder, to get information. It can also be done by camera (known as video attack) or other vision-enhancing devices. Shoulder surfing attack can be catalyzed as *one-time shoulder surfing attack* and *multi-time shoulder surfing attack*. Excluding threats like phishing attack or malware attack, we only focus on attacks targeting weakness of human-computer interface in this paper.

Besides, we also take guessing attack and replay attack into consideration in this paper. A attacker conducts guessing attack to obtain users' PIN by brute-force attack or dictionary attack based on partial knowledge on users or not. In this paper, replay attack refers to obtaining the a user's complete slide sequence when he successfully unlocks his smartphone and reusing this sequence or one of its subsequences to illegally get access to the user's smartphone.

Guessing Attack. Similar to traditional 4-digit PIN mechanism, a 4-digit PIN is adopted in SlidePIN. Thus both traditional 4-digit PIN mechanism and SlidePIN have the same PIN space, 10000, and the capability of withstanding guessing attack is equivalent.

[3] We excluded the start and the end point when estimating distance between keys.

One-time Shoulder Surfing Attack. Given that the attacker has obtained one valid unlock sequence. Because several invalid numbers are concealed with the user's PIN in a longer slide sequence, the attacker have to take major efforts to extract the PIN. In the worst case, the attacker has successfully conducted one-time shoulder surfing attack and obtained the exact slide sequence and the length of sequence is 9, the lower threshold value, the probability of getting the PIN will be $1/\binom{9}{4} \approx 0.79\%$. However when confronting shoulder surfing attack, input process of traditional 4-digit PIN mechanism or even the PIN is directly exposed to the attacker. Therefore SlidePIN is better at resisting one-time shoulder surfing attack.

Multi-time Shoulder Surfing Attack. Assuming in the worst-case situation, an attacker may capture several valid slide sequences when a user slides to unlock, and PIN can be calculated by using statistical methods. In this paper, we calculate the longest common sequence (LCS) of slide sequences in a simulated attack experiment to validate the security of SlidePIN against multi-time shoulder surfing attack. Longer sequences are more secure against multi-time shoulder surfing attacks. In SlidePIN, the lower threshold of sequence length is set to 9 and due to that, SlidePIN owns capability of resisting such attacks, which will be detailed in the attack experiment.

Replay Attack. In order to resist replay attack, a random numeric keypad is introduced in SlidePIN. Every time before users input PIN, it generates a new layout. So the layout and the sequence are indispensable when replay attack is conducted. Moreover, merely reentering the entire slide sequence with another randomly distributed layout leads a prominent prolonging on slide sequence that will exceed the length limitation discussed in the sequence length analysis and experiment.

Additionally, shoulder surfing attack has negative relevance with replay attack. Longer sequences have stronger capability of resisting shoulder surfing attack but weaker capability against replay attack. Therefore, it is an effective solution and a reasonable tradeoff to set thresholds of sequence length discussed above for SlidePIN.

4.4 Usability Analysis

In this section, usability is evaluated from perspectives of cost of learning, unlock time, orientation time and error rate.

Cost of Learning. SlidePIN is easy to learn and use.

SlidePIN is built based on 4-digit PIN. A SlidePIN keypad is similar to the keypad used in 4-digit PIN. However, clicking and sliding can both be processed in SlidePIN and no mode-switch is demanded. Therefore SlidePIN is easy to learn and provides a smooth transition for users from click input to slide input.

SlidePIN is easy to use. As mobile devices (e.g. smartphones, tablet computers), particularly the ones with touch-screen, rapidly pervade the world, sliding has become one of most common gestures for users. With slide input method, SlidePIN provides an easy and comfortable way for users to unlock their devices.

SlidePIN is interesting to use. Doodling or scrawling being human's inborn preference, it is more interesting to "doodle around" with slide input method than to click fixed buttons mechanically.

Orientation Time. Users need to observe the keypad layout every time before an unlocking movement and the duration of this process is defined as *orientation time*. Since a random numeric keypad is adopted in SlidePIN, orientation time in SlidePIN will be longer than that in traditional 4-digit PIN mechanism.

Unlock Time. Sliding is faster than clicking, however, as a random numeric keypad is introduced and input sequences become longer, unlock time is increased.

Sliding is faster. Clicking input can be considered as a task sequence consisting of multiple single tasks. Each of these tasks can be described by Fitts' Law [20, 21] as entering a single character by clicking. Similarly, slide input in SlidePIN can be regarded as a task sequence including tasks that require sliding over a character. Accot and Zhai have already proved that Fitts' Law is applicable in the case of slide input and sliding across target characters is faster than clicking.[22]

Input Sequences Become Longer. An input sequence contains 6 characters rather than 4, including not only 4-digit PIN but a start point '*' and a end point '#', which partly counteracts the advantage in input speed brought by slide input method.

Random Numeric Keypad Increases Unlock Time. When using a random keypad, some users tend to skip this observation process and input immediately as they get the keypad layout. In this case, it will take them longer to search for next target character after one has already been input.

A random numeric keypad apparently cannot help users form fixed gesture and relevant memory to faster unlock after use experience is accumulated. However, movement that is easy to be transformed into stable memory is subject to attack.

Error Rate. SlidePIN system is more complicated than 4-digit PIN system, which leads to a slight increment on the error rate of unlocking by using SlidePIN. Error rate increases mainly because: 1) Certain range of input sequence length has been set up in SlidePIN. 2) Start point '*' and end point '#' is required during each slide. 3) Participants in our experiments are not familiar enough with this novel mechanism.

5 Experimental Analysis

To evaluate the usability and security of SlidePIN, we recruited 20 students as volunteers to conducted relevant experiments including sequence length experiment, unlock experiment and attack experiment. Based on data obtained from these experiments, we analyzed the reasonable range of slide sequence, distance between two digits of a PIN, orientation time, unlock time, error rate and the capability of SlidePIN against multi-time shoulder surfing attacks.

5.1 Experiment Design

Settings. We designed and implemented an app as experiment software to conduct sequence length experiment and unlock experiment. The app is deployed on a ZTE U930 smartphone (4.2 inches screen, 960 * 540 resolution, Android 4.0).

Volunteers. we recruited 20 students from different majors as volunteers, ranging in age from 18 to 26 (average: 24). 8 were female and 12 were male and all of them have 2 more years experience of using a smartphone and a unlock system.

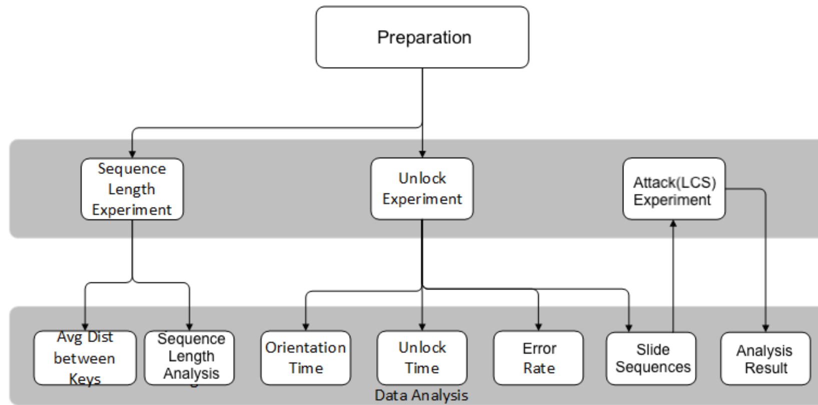


Fig. 6. Experiment Flow Chart

Description.

Preparation.

1. A quick training on how to correctly use SlidePIN is provided to volunteers. They could practise for 5 minutes before experiments formally started.

2. Some essential operation protocols, for example, sliding from '*' and ending with '#', were additionally emphasized.
3. PINs used in the experiments are randomly generated to avoid the negative effects brought by the preference of users.

Sequence Length Experiment Process. As mentioned above, the sequence length has effects on security and usability. So we carried out an experiment to analyze sequence length and distribution.

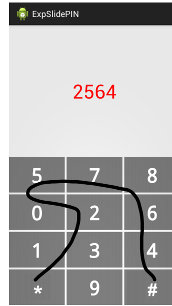


Fig. 7. Sequence Length Experiment UI

In this experiment, according to a 4-digit PIN randomly generated on the screen, each volunteer slid to input the PIN six times without any limitation on slide sequence length, as showed in Fig.7. Due to different keypad layouts generated randomly, one PIN maps to 6 different slide sequences with different sequence length. The slide sequences, sequence length, PINs and layouts are recorded by experiment software in real time for further analysis.

Table 1. Group Setting of Unlock Experiment

Method	Traditional Keypad	Random Keypad
Click	Group 1 (Traditional 4-digit PIN)	Group 2
Slide	Group 3	Group 4 (SlidePIN)

Unlock Experiment Process. In unlock experiment, each volunteer needs to finish inputing each PIN in 4 different groups and for each PIN, 6 more times are conducted. Limitation of sequence was set up ranging from 9 to 15. The slide sequences beyond this limitation were identified as a invalid slide sequence leading to a failure unlocking.

A 2-second countdown is set before keypad appears. As soon as the countdown ends, a timer will start to record *orientation time* till volunteers touch the keypad and begin to input.

The duration from the screen being touched to unlocking being successfully finished would be recorded as *unlock time*. Otherwise, times of failure would be recorded. Application calculates the *error rate* and exits after 6 successful unlocking.

Attack Experiment Process. Based on slide sequences obtained from unlock experiment, attack experiment is conducted to evaluate SlidePIN's security against multi-time shoulder surfing attack. The minimum quantity of slide sequences that can help one get the PIN statistically is evaluated by comparing LCS (Longest Common Subsequence) of them with the PIN.

5.2 Experiment Analysis.

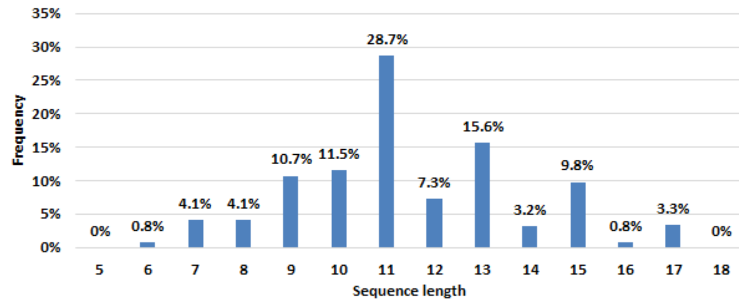


Fig. 8. Distribution of Sequence Length

Sequence Length Experiment. Based on statistical calculation, the average of sequence length is 11.46, approximate to the theoretical value 11.55. Meanwhile, according to the previous section, the average distance between keys is 2.49 and the standard deviation is 1.16.

As Fig.8 shows, 9.0% of sequences are shorter than 9 (lower threshold), which means that setting lower threshold to 9 leads to a loss on usability.

Additionally, all sequences are equal or shorter than 17. Only 4.1% are longer than 15 (upper threshold), while sequences with 15 has reached 9.8%. So it's reasonable to set the upper threshold 15 to improve usability.

Based on theoretical and experimental analysis, it is a good tradeoff to set 9 as the lower threshold and 15 as the upper threshold. Nearly 13.1% of valid sequences are excluded in this case, which compromised usability a little, however, the security is greatly improved.

Unlock Experiment.

Orientation Time. We collect 4 groups of data about orientation time from unlock experiment and put corresponding points on both sides of a time axis. Then a diagram (Fig.9) describing distribution of orientation time is generated. On each axis, ignoring the points with large error by setting up a threshold value marked on the axis leads to a result of higher accuracy.

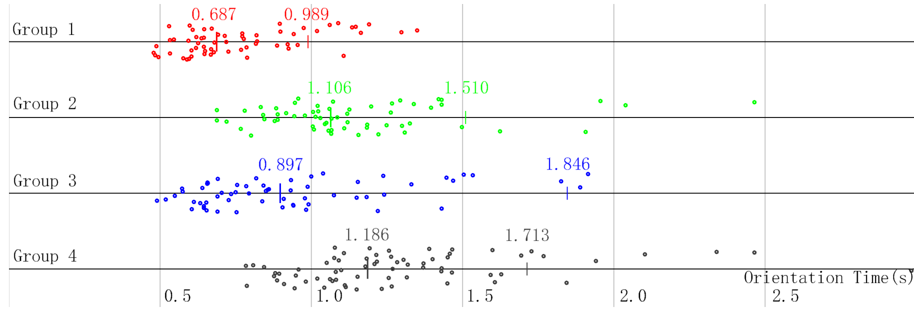


Fig. 9. Orientation Time

According to Fig.9 and Table 2, average orientation time in Group 2 and 4 is longer than in Group 1 and 3, which demonstrates that random keypad increases orientation time. However, orientation time in click input is approximate to that in slide input, which means slide input method has almost no effect on orientation time. Finally, SlidePIN has limited impact on orientation time, compared with 4-digit PIN mechanism.

Table 2. Orientation Time (s)

Groups	Average	Standard Deviation	Threshold Value
1	0.687	0.133	0.989
2	1.064	0.199	1.510
3	0.798	0.293	1.846
4	1.186	0.225	1.713

Unlock Time. According to Table 2, we can conclude that "random keypad" leads to additional unlock time. Volunteers are more familiar with traditional 4-digit PIN mechanism. Besides, more digits, start point '*' and end point '#' included, need to be input in SlidePIN. Consequently, the slide input method is not the main reason that increases the unlock time.

The average distance of two numbers to input is 2.49, we can find that average elapsed time between two input in Group 4 (SlidePIN) is longer than that in Group 1 (4-digit PIN) by only about 0.14 second. Even if target characters have reached 6 after bring in start point and end point, the average unlock time of

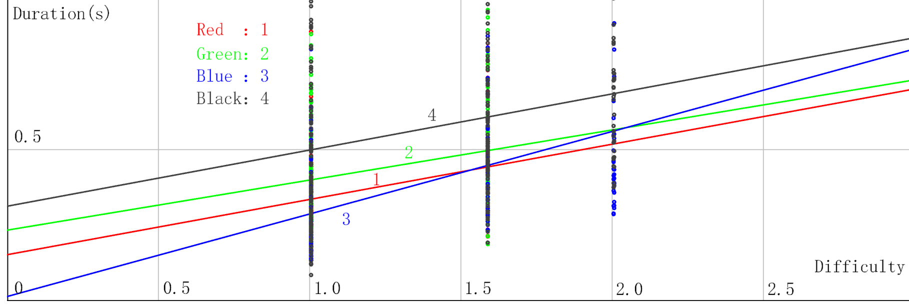


Fig. 10. Experiment Result of Unlocking Time with Fitts' Law

SlidePIN, 3.552 seconds[4], is longer than that of traditional 4-digit PIN, 1.597 seconds, by less than 2 seconds according to calculation based on data collected in this experiment.

Table 3. Regression Equation on Sliding

Groups	Regression Equation
1	T = 0.182*ID + 0.153, a = 0.153, b=0.182
2	T = 0.165*ID + 0.234, a = 0.234, b=0.165
3	T = 0.272*ID + 0.015, a = 0.015, b=0.272
4	T = 0.185*ID + 0.314, a = 0.314, b=0.185

Error Rate. Calculating the error rate, we can conclude that SlidePIN has higher error rate than traditional 4-digit PIN and that both "random keypad" and "slide input" separately have negative effects on error rate.

Table 4. Error Rate

Groups	Group 1	Group 2	Group 3	Group 4
Error Rate	1.67%	3.33%	7.69%	13.04%

In Group 4, 69 input samples are valid for evaluating and 9 of them failed, which makes the error rate reach 13.04%. And in all of these 9 samples, the

[4]For consecutive input tasks, input time can be calculated based on
$T_n = N \times a + b \times k = \sum_{k=1}^{n-1} \log_2(\frac{D_{k,k+1}}{W} + 1)$,
where W represents the offset on the movement direction, N represents the number of characters and $D_{k,k+1}$ represents the distance between the k-th and the (k+1)-th keys [20]

sequence length was less than 9, the lower threshold of sequence length. Based on the sequence experiment described above, length of 13.1% input sequences is beyond the sequence length limitation, which is of high resemblance with error rate. According to this, we believe that it is closely related to length limits of slide sequence and that as the experience of using SlidePIN is accumulated, the error rate will simultaneously increase.

Attack Experiment. We evaluate the SlidePIN's capability of withstanding multi-time shoulder surfing attacks or statistical attacks by calculating the LCS of slide sequences and when the length of LCS is reduced to 4, we believe that the PIN is exposed. In this experiment, we chose the first 10 valid slide sequences from the previous experiment and calculated the LCS of their slide sequences with incremental quantity till the PIN was exposed, as is shown in Table 5:

Table 5. LCS Analysis ("4" means PIN has been exposed)

Volunteer ID	a	b	c	d	e	f	g	h	i	j
LCS length of 2 slide sequences	6	6	6	6	7	6	6	7	6	4
LCS length of 3 slide sequences	5	5	4	4	4	4	4	5	4	
LCS length of 4 slide sequences	4	4						4		

As is showed in Table 5, a PIN was successfully extracted from 2 sequences involved in just one sample and in many cases, 3 or 4 sequences were necessary for extracting the 4-digit PIN. Thus SlidePIN performs better at resisting multi-time shoulder surfing attacks (statistic attacks) than tradition 4-digit PIN mechanism.

6 Discussion

PIN Storage and Input Sequence Verification. Hash encryption methods like MD5 are not feasible in SlidePIN for PIN storage and verification. Considering both the security of users' credentials and the practicability on smartphones, we provide a solution as follows:

PIN Storage: The device ID of a smartphone is chosen as the identity and a secret key is calculated based is as well as the user's PIN, and then the key is stored after encryption.

Verification: The plaintext of the PIN is obtained and then it will be compared with the input sequence. If the PIN is one of subsequences contained in the input sequence, then authentication is passed.

Fixed Start Point and End Point. SlidePIN demands users to slide from '*' and end with '#', which makes users input 6 target characters. Although slide input costs less time than clicking, additional digits make negative effects on duration of a complete process of unlocking. However, if terminal points '*' and

'#' are not demanded, part of users prefer to slide from the first digit of their PIN and to the last one, which leads to an impairment on security of SlidePIN because merely two digits are concealed during a sliding with the first one and last one exposed.

Same Adjacent Digits. PINs with same adjacent digits are supported in SlidePIN. Given that PIN is '1158', the user needs to slide away from the first '1' to others and then slide back to '1' making the input sequence '11'. However, we highly recommend users to use PIN with no same adjacent digits in SlidePIN to improve both security and usability.

Smudge Attack. A smudge attack [19] is a method to discern the PIN or password pattern of a touchscreen device such as a smartphone or tablet computer. SlidePIN adopting a random numeric keypad, different slide tracks and keypad layouts will be generated each time users slide, which makes SlidePIN effectively resist smudge attack.

Attacks Based on Features. Besides unlock sequences captured by shoulder surfing attack, users' precise features (e.g. angles and dwell time of single digit of slide sequences) can be captured and analyzed to attack SlidePIN. For instance, if a user tends to take a sudden turn on a specific character when sliding, it is of high possibility that this character is one digit of the PIN.

Attacks targeting weakness of human-computer interface discussed in this paper can hardly accurately capture those features. We plan to evaluate the how the attacks based on precise features will affect SlidePIN in our future work.

7 Conclusion and Future Work

SlidePIN performs better than 4-digit PIN against shoulder surfing attacks. At the same time, it has acceptable usability.

In the future work, we plan to conduct more research on numeric keypad layouts and using experiences to find out their impacts on usability and security. In addition, we will try to design SlideText based on English letters.

References

1. Goucher W.: Look behind you: the Dangers of Shoulder Surfing. Computer Fraud & Security. 17–20 (2011.11)
2. Schaub, F., Deyhle, R., and Weber, M. Password entry usability and shoulder surfing susceptibility on different smartphone platforms. In Proc. Mobile and Ubiquitous Multimedia (MUM '12), ACM (2012).
3. Montgomery, E.B.: Bringing Manual Input Into the 20th Century: New Keyboard Concepts, Computer (1982), 15(3) 11–18.

4. Shumin, Z. Per Ola, K,. The Word-Gesture Keyboard: Reimagining Keyboard Interaction. Communications of the ACM, 2012(9), 91–101.
5. M. Kumar, T. Garfinkel, D. Boneh, and T. Winograd. Reducing shoulder-surfing by using gaze-based password entry. In Proc. SOUPS07, ACM (2007), 13–19.
6. Sasamoto, H., Christin, N., and Hayashi, E. Undercover: authentication usable in front of prying eyes. In Proc. CHI08 (2008), 183–192.
7. De Luca, A., von Zezschwitz, E., and Hussmann, H. Vibrapass: secure authentication based on shared lies. In Proc CHI09 (2009), 913–916.
8. De Luca, A., von Zezschwitz, E., Dieu Huong Nguyen,N.,Maurer,M., Rubegni,E., Paolo Scipioni, M.,Langheinrich,M.,Back-of-Device Authentication on Smartphones. In Proc CHI13 (2013), 2389–2398.
9. Azenkot, S., Rector, K., Ladner, R., and Wobbrock, J. Passchords: secure multi-touch authentication for blind people. In Proc. ASSETS 12, ACM (2012), 159–166.
10. Andrea B., Ian O., Dong-Soo K., Bianchi Open Sesame Design Guidelines For Invisible Passwords. Computer (2012), 58–65.
11. V. Roth, K. Richter, and R. Freidinger. A pin-entry method resilient against shoulder surfing. In Proc. CHI 04 (2004), 236–245.
12. Tan, D. S., Keyani, P., and Czerwinski, M. Spy- resistant keyboard: more secure password entry on public touch screen displays. In Proc. OzCHI05 2005, 1–10.
13. Wiedenbeck, S., Waters, J., Sobrado, L., and Birget, J.-C. Design and evaluation of a shoulder-surfing resistant graphical password scheme. In Proc. AVI 06 (2006), 177–184.
14. M. Lei, Y. Xiao, S. Vrbsky, C.-C. Li, and L. Liu. A virtual password scheme to protect passwords. In Proc. ICC08 (2008), 1536–1540.
15. De Luca A, Hertzschuch K, Hussmann H. ColorPIN: securing PIN entry through indirect input. In Proc CHI10, ACM(2010), 1103–1106.
16. Takada, T. FakePointer: An Authentication Scheme for Improving Security against Peeping Attacks Using Video Cameras. In Proc. UBICOMM '08, 2008, 395–400.
17. Weiss, R., and De Luca, A. Passshapes - utilizing stroke based authentication to increase password memorability. In Proc. NordiCHI08, ACM (2008), 383–392.
18. Kim, D., Dunphy, P., Briggs, P., Hook, J., Nicholson, J.,Nicholson, J., and Olivier, P. Multi-touch authentication on tabletops. In Proc. CHI10, ACM (2010), 1093–1102.
19. Aviv, A., Gibson, K., Mossop, E., Blaze, M., and Smith, J.: Smudge attacks on smartphone touch screens. In Proc. USENIX 2010, USENIX Association (2010), 1–7.
20. Bi X, Li Y, Zhai S. Fitts law: Modeling finger touch with Fitts' law; proceedings of the Proceedings of the 2013 ACM annual conference on Human factors in computing systems, F, 2013 [C]. ACM.
21. Fitts P M. The information capacity of the human motor system in controlling the amplitude of movement [J]. Journal of experimental psychology, 1954, 47(6): 381–391.
22. Johnny Accot, Shumin Zhai : Performance evaluation of input devices in trajectory-based tasks: An application of the steering law. In Proceedings of ACM CHI 1999 Conference on Human Factors in Computing Systems(1999), 466–472.

Private Password Auditing
Short Paper

Amrit Kumar and Cédric Lauradoux

INRIA, France
{firstname.lastname}@inria.fr

Abstract. Password is the foremost mean to achieve data and computer security. Hence, choosing a strong password which may withstand dictionary attacks is crucial in establishing the security of the underlying system. In order to ensure that strong passwords are chosen and that they are periodically updated, system administrators often rely on password auditors to filter weak password digests. Several tools aimed at preventing digest misuse have been designed to aid auditors in their task. We however show that the objective remains a far cry as these tools essentially reveal the digests corresponding to weak passwords. As a case study, we discuss the issues with *Blackhash*, and develop the notion of *Private Password Auditing* — a mechanism that does not require a system administrator to reveal password digests to an external auditor and symmetrically the dictionaries remain private to the auditor. We further present constructions based on *Private Set Intersection* and its variant, and evaluate a proof-of-concept implementation against real-world dictionaries.

Keywords: Password, Auditing, Password cracking, Private Set Intersection, Private Set Intersection Cardinality.

1 Introduction

Passwords are pervasive to data and computer security. However, despite their utmost reliance on passwords, users often deliberately choose one which is common and easy to remember. This has been confirmed by the data leakages over the recent past[1]. In addition to revealing the password habits of users, these leakages have further increased the number of dictionaries containing common and weak passwords. These dictionaries form the basis of password cracking tools such as John the Ripper[2] and Hashcat[3]. Furthermore, Narayanan et al. [5] show that as long as passwords remain human-memorable, they are vulnerable to "smart dictionary attacks" even when the space of potential passwords is large.

The advances made in password crackers and the ever evolving dictionaries have forced system administrators to take drastic measures to protect their users. Password auditing is one of them. System administrators periodically audit system passwords to inform users to change their passwords in case they are found to be weak (with respect to the available dictionaries and the cracking tools). Typically, they extract password digests from systems and then they themselves perform an internal audit. Another alternative is to outsource this task to an expert third-party security auditor or to an in-house security team. Since a system administrator has

[1] http://bit.ly/19xscQO
[2] http://openwall.com/john/
[3] http://hashcat.net/oclhashcat/

privileged access to several sensitive user information, revealing the weakness of a user password to him may lead to massive security breach. Furthermore, considering the expertise of external auditors and to ensure transparency of the process, the latter approach to auditing is often preferred.

Several proprietary tools such as `l0phtcrack.com` as well as free softwares have been developed to aid password auditors. Most of these auditing tools go beyond determining whether a password is weak. For instance, they also allow the auditor to verify whether the passwords are periodically changed by the users. Some free softwares, a notable example being *Blackhash* [1], are essentially restricted to knowing whether system passwords are weak. However, these tools can be easily adapted to perform a full scale auditing.

While tools capable of performing full scale auditing require the password digests of all the users, some specialized tools such as Blackhash claim to filter weak passwords without having access to the full digests. Contrary to the claims, we highlight that these password auditing tools, in particular Blackhash require the system administrator to reveal the password digests corresponding to *easy-to-crack* passwords. Eventually, these tools require the administrator to reveal weak passwords. A malicious auditor may use these passwords for his own benefit before reporting its potential weakness to the administrator.

To this end, we present *Private Password Auditing*: a mechanism that allows a user or a system administrator to filter weak passwords from the password digests without revealing the digests to the auditor. Furthermore, the dictionaries used for auditing remain private to the auditor. The presented tool relies on *Private Set Intersection* [4] and *Private Set Intersection Cardinality* [3]. We finally evaluate the performance of a proof-of-concept implementation of the tool. This leads us to the conclusion that in the general auditing scenario, private password auditing tools are practical.

2 Password Auditing

Password auditing may be considered as a preventive mechanism to resist password cracking tools. In its restricted form, password auditing consists of determining whether any of the system passwords are weak and hence susceptible to cracking tools. This is essentially performed with the help of an auditor who uses dictionary based tools to filter weak digests. In the following we present existing approaches to password auditing of this kind and analyze their weaknesses.

2.1 Naive Approach

A naive approach to password audit would typically involve extracting password digests from systems and then sending them to a third-party security auditor or an in-house security team. The auditor relying on tools such as John the Ripper or Hashcat may easily uncover potentially weak passwords. However, such an approach ensues serious risks. The password digests may be lost or stolen from the security team. Furthermore, a rogue security team member may secretly make copies of the password digests and may mount *pass-the-hash attacks*. Worse, some of these digests may correspond to easy-to-crack passwords. The auditor may recover in clear the weak passwords and use it for malicious purposes before reporting it to the system administrator.

Consequently, it is hard to guarantee that the password digests are handled and disposed of securely and that access to the digests is not abused. Indeed, only the system administrator and his team should have access to password digests. Extracting the digests and giving them to someone else fundamentally compromises the security of the system.

2.2 Auditing Without Full Hashes

This kind of auditing checks system digests for weak passwords without actually having access to the full digests. A notable example is Blackhash [1], which is based on Bloom filters [2]. In the following we briefly describe Bloom filters and in the sequel we present Blackhash.

Bloom Filter. Bloom filter [2] is a space and time efficient probabilistic data structure that provides an algorithmic solution to the *set membership query problem,* which consists in determining whether an item belongs to a predefined set.

Classical Bloom filter as presented in [2] essentially consists of k independent hash functions $\{h_1, \ldots, h_k\}$, where $\{h_i : \{0,1\}^* \to [0, m-1]\}_k$ and a bit vector $\boldsymbol{z} = (z_0, \ldots, z_{m-1})$ of size m initialized to $\mathbf{0}$. Each hash function uniformly returns an index in the vector $\boldsymbol{z}$. The filter $\boldsymbol{z}$ is incrementally built by inserting items of a predefined set $\mathcal{S}$. Each item $x \in \mathcal{S}$ is inserted into a Bloom filter by first feeding it to the hash functions to retrieve k indices of $\boldsymbol{z}$. Finally, insertion of x in the filter is achieved by setting the bits of $\boldsymbol{z}$ at these positions to 1.

In order to query if an item $y \in \{0,1\}^*$ belongs to $\mathcal{S}$, we check if y has been inserted into the Bloom filter $\boldsymbol{z}$. Achieving this requires y to be processed (as in insertion) by the same hash functions to obtain k indexes of the filter. If any of the bits at these indexes is 0, the item is not in the filter, otherwise the item is present (with a small *false positive probability*).

The space and time efficiency of Bloom filter comes at the cost of false positives. If $|\mathcal{S}| = n$, i.e. n items are to be inserted into the filter and the space available to store the filter is m bits, then the optimal number of hash functions to use and the ensuing optimal false positive probability p satisfy:

$$k = \frac{m}{n} \ln 2 \qquad \text{and} \qquad \ln p = -\frac{m}{n} (\ln 2)^2 .$$

Blackhash

Blackhash [1] is a tool for restricted auditing of passwords, i.e. check for weak password digests in the system file without having access to the full digests. It works by building a Bloom filter from the system password digests. The system manager extracts the password digests and then uses Blackhash to build the filter. The filter is saved to a file, then compressed and given to the audit team. The audit team maintains a set of dictionaries of weak passwords against which the password digests are to be tested. Upon reception of the filter, the auditor simply checks for each entry of the dictionary, whether or not it is present in the filter. If weak passwords are found to be present in the filter, the security team creates a weak filter of these passwords and sends it back to the system manager. Finally, the system manager tests the weak filter against the system digests to identify individual users with weak passwords.

Bloom filter parameters. The filter size m to store the system digests is 2^{26} bits, and can accommodate up to 1 million digests. The number of hash functions $k = 2$, and the hash functions employed are either MD4 or MD5. Developers claim to achieve a false positive probability of 0.0008. Clearly, these parameters are not optimal. To achieve a false positive probability of 0.0008 for 1 million digests, a filter of size $14,842,031 \approx 2^{24}$ bits is required.

Issues with Blackhash. Developers claim that Blackhash does not reveal password digests to the auditor. Hence, it constitutes a better and secure tool compared to the naive approach. Contrary to the claim, using a Bloom filter of password digests instead of full digests does not improve user's privacy. The most serious issue with Blackhash is that the auditor while finding

the weak passwords with the help of dictionaries actually retrieves the weak passwords in clear. To paraphrase, Blackhash requires the system administrator to reveal the weak passwords. Furthermore, due to the false positive probability of Bloom filters, strong passwords might get detected as being weak. Keeping the false positive probability extremely low however comes at the cost of space/time required to store/query the filter.

3 Private Password Auditing (PPA)

In the previous section, we highlighted the issues with Blackhash. The most serious one being that, it requires the administrator to reveal weak passwords. To this end, we propose *Private Password Auditing* (PPA), a mechanism which does not require a user or the administrator to reveal password digests while auditing. Two scenarios may be considered where PPA may play important role:

Multi-user scenario: There is a system administrator with a list of system password digests and wishes to know the ones which correspond to easy-to-crack passwords. Once these passwords are identified, the respective owners are contacted and asked to change their passwords.

Single user scenario: There is a user who wishes to know whether his password digest is easy-to-crack.

We suppose that auditing in both the scenarios is performed with the help of an external auditor who may be malicious and that auditing is restricted to verifying whether provided password digests contain weak ones. We also suppose that the auditor performs a dictionary based password cracking, i.e. the auditor checks whether a password digest corresponds to the digest of a word in the given dictionary (or a set of dictionaries).

Privacy guarantees. In addition to the fact that the user or system digests are not revealed to the auditor, the external auditor himself may not wish to reveal the dictionaries he uses for password auditing. This is usually the case for proprietary tools. Hence, PPA simultaneously ensures privacy for both the system administrator/user and the auditor. The digest(s) hence remain private to the user/administrator and symmetrically, the dictionaries used for auditing remain private to the auditor.

In the following, we present construction of a PPA tool that relies on a primitive called *Private Set Intersection* and its variant. The construction can be seen as an application of private set intersection in password auditing. We succinctly present private set intersection protocols and in the sequel we present its variant called *Private Set Intersection Cardinality*. For each primitive, we discuss its applicability to private password auditing.

3.1 PPA based on Private Set Intersection

Private Set Intersection (PSI) considers the problem of computing the intersection of private datasets of two parties. The scenario consists of two sets $\mathcal{U} = \{u_1, \ldots, u_m\}$, where $u_i \in \{0,1\}^\ell$ and $\mathcal{DB} = \{v_1, \ldots, v_n\}$, where $v_i \in \{0,1\}^\ell$ held by a user and the database-owner respectively. The goal of the user is to privately retrieve the set $\mathcal{U} \cap \mathcal{DB}$. The privacy requirement of the scheme consists in keeping $\mathcal{U}$ and $\mathcal{DB}$ private to their respective owner. There is an abounding literature on novel and computationally efficient PSI protocols. The general conclusion being that for security of 80 bits, protocol by De Cristofaro et al. [4] performs better than all other protocols, while for higher security levels, other protocols supersede the protocol by De Cristofaro et al.

PSI provides a primitive to design a PPA tool in the multi-user scenario where a system administrator has a list of system digests and wishes to know the digests which correspond to weak passwords. We suppose that the auditor has a dictionary of weak digests $\mathcal{DB} = \{w_1, \ldots, w_n\}$ and the administrator owns the digest set $\mathcal{U} = \{d_1, \ldots, d_m\}$. Then by invoking a PSI protocol on the sets, the administrator may know the digests which are easy-to-crack. The security of PSI ensures that the sets remain private to their respective owner.

3.2 PPA based on Private Set Intersection Cardinality

Private Set Intersection Cardinality (PSI-CA) is a variant of PSI where the goal of the client is to privately retrieve the cardinality of the intersection rather than the contents. While generic PSI immediately provide a solution to PSI-CA, they however yield too much information. While several PSI-CA protocols have been proposed, we concentrate on PSI-CA protocol of De Cristofaro et al. [3], as it is the most efficient.

PSI-CA builds a PPA primitive in the single user scenario, where a user wishes to know if his password is weak with respect to the existing dictionaries. As earlier, the auditor has a dictionary of digests $\mathcal{DB} = \{w_1, \ldots, w_n\}$ and the user owns a digest d. Clearly, invoking an instance of PSI-CA protocol on the sets, the user may privately know if his password digest is easy-to-crack: if the intersection set is of cardinality 1, then the password digest is weak. The security of PSI-CA again ensures that data remain private to their respective owner.

4 Practicality of PPA Tool

We implemented the PSI protocol by DeCristofaro et al. [4] and the PSI-CA protocol of [3], since they are the most efficient. Recommended parameters of $|p| = 1024$ and $|q| = 160$ bits have been used for PSI-CA, while an RSA modulus of 1024 bits has been considered for PSI. For both the primitives, SHA-1 hash function has used for signatures. These parameters ensure a security of 80 bits in the *semi-honest* adversary model.

We evaluated PPA tools based on these protocols and compared their performance with Blackhash. The tests were performed on a 64-bit processor desktop computer powered by an `Intel Xeon E5410 3520M` processor at `2.33 GHz` with `6 MB cache`, `8 GB RAM` and running `3.2.0-58-generic-pae Linux`. We have used `GCC 4.6.3` with `-O3` optimization flag. The implementation uses GMP library[4] v4.2.1.

For Blackhash and PSI based tool, we fix the number of system digests to be $59,169$. This corresponds to a representative data provided with the Blackhash source code. In order to evaluate the performance of the techniques, we tested these implementations against real-world dictionaries of varied sizes, from 100 entries up to 14 million entries. The dictionaries are presented in Table 1a.

Table 1b presents the results obtained for unsalted SHA-1 password digests. We observe that while Blackhash is not privacy-friendly, it is the most efficient. This is due to the time efficiency of the underlying Bloom filters. PPA tool based on PSI-CA is faster than Blackhash for smaller dictionaries since PSI-CA considers only one digest. Moreover, even for moderately large dictionaries (2M), the audit time remains very practical, i.e. 3 mins. PSI based tool incurs considerable cost for large dictionaries. In fact, both PSI and PSI-CA based private auditing against large dictionaries are suitable in the settings where password auditing is not supposed to be instantaneous, which usually is the case. Indeed a security audit may last for days.

[4] `https://gmplib.org/`

Table 1: Dictionaries used and the results obtained for $59,169$ unsalted system digests.

(a) Dictionaries used.

Dictionary	# entries
Top-100	100
John the Ripper (JtR)	3107
Xato Top-10k (Xato)	10,000
Cain & Abel (C&A)	306,706
Dazzlepod	2,151,220
RockYou	14,344,391

(b) Cost incurred by different auditing tools.

		Time					
	Tool	Top-100	JtR	Xato	C&A	Dazzle.	RockYou
	Blackhash [1]	6s	6s	6s	15s	1m	2m
PPA	PSI-CA [3]	47ms	359ms	1s	28s	3m	23m
PPA	PSI [4]	1m	1m	2m	6m	37m	4h

We highlight that auditing of salted digests is very similar to the unsalted ones. To this end, we assume that the salts are public and hence known to the auditor. In case of single user digest, the auditing time remains largely unaffected. While in the multi-user scenario, the size of the set on the auditor's side gets increased by a factor of m to incorporate the salt of each user, where m is the number of users. The auditing time hence is increased by a factor $\approx m$.

5 Conclusion

In this work, we discussed the issues faced by system administrators in face with malicious auditors. The existing password auditing tools essentially require the system administrator or the user to reveal weak passwords. While password auditing tools like Blackhash may prevent pass-the-hash attacks, they are yet susceptible to revealing weak passwords to the auditor. Considering the extreme sensitivity of passwords, more secure means must be deployed to ensure the privacy of passwords. To this end, we provide a private password auditing tool which does not require the user to reveal the password digests to the external auditor. Symmetrically, the auditor keeps his dictionaries private. The tool is based on private set intersection and its variants. An evaluation reveals that privacy friendly tools are practical in scenarios where password auditing is not instantaneous (which usually is the case). We highlight that the primitives used in PPA require heavy public key operations, a future work consists in designing efficient and dedicated PPA protocols relying only on symmetric cryptographic primitives.

Acknowledgements: This research was conducted with the partial support of the Labex PERSYVAL-LAB(ANR–11-LABX-0025) and the project-team SCCyPhy. We also thank anonymous reviewers for their suggestions and remarks.

References

1. Richard B. Tilley. Blackhash software. `http://16s.us/software/Blackhash/`.
2. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13, July 1970.
3. Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In *Cryptology and Network Security*, volume 7712 of *Lecture Notes in Computer Science*, pages 218–231. Springer Berlin Heidelberg, 2012.
4. Emiliano De Cristofaro and Gene Tsudik. Experimenting with Fast Private Set Intersection. In *Trust and Trustworthy Computing*. Springer Berlin Heidelberg, 2012.
5. Arvind Narayanan and Vitaly Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 364–372, New York, NY, USA, 2005. ACM.

SAVVIcode: Preventing Mafia Attacks on Visual Code Authentication Schemes (Short Paper)

Jonathan Millican and Frank Stajano
jrm209@cantab.net, frank.stajano@cl.cam.ac.uk

University of Cambridge Computer Laboratory

Abstract. Most visual code authentication schemes in the literature have been shown to be vulnerable to relay attacks: the attacker logs into the victim's "account A" thanks to credentials that the victim provides with the intent of logging into "account B". Visual codes are not human readable and therefore the victim cannot distinguish between the codes for A and B; on the other hand, codes must be machine-readable in order to automate the login process. We introduce a new type of visual code, the SAVVIcode, that contains an integrity-validated human-readable bitmap. With SAVVIcode, attackers have a harder time swapping visual codes surreptitiously because the integrity check prevents them from modifying or hiding the human-readable distinguisher.

1 Introduction

A current area of research in the field of authentication is *visual code authentication schemes* (VCASs). In such schemes, the user logs into a remote system by acquiring a visual code with a scanning device rather than (or, sometimes, in addition to) typing a password. Examples of such systems include Snap2Pass [5], our own Pico [9], tiQR [10], QRAuth [1] and SQRL [6], as well as patents filed by GMV solutions [2] and Google [4]. The "visual code" is a kind of two-dimensional barcode — often the QR-code that modern smartphones are already equipped to decode. Indeed, in many of the authentication schemes above, the device that scans the visual code is the user's smartphone.

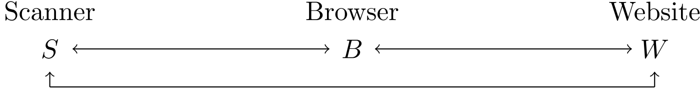
Jenkinson et al [7] have shown that, without an additional out-of-band secure communication channel from the device to the terminal, these schemes are inherently vulnerable to "Mafia fraud relay attacks" when they use existing types of visual code.

We propose an extension to visual codes that prevents an attacker from undetectably substituting a visual code for another. By making any attacks obvious to the user, we hope this mechanism will greatly reduce the likelihood that such attacks can succeed.

Pre-proceedings version. Do not circulate.
Final version to appear in *Proceedings of Passwords 2014*, Springer LNCS.

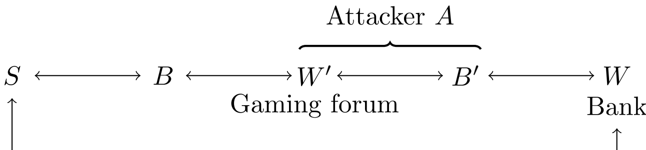
2 The problem: Mafia Fraud Relay Attacks

Jenkinson et al [7] outline the typical operation of a VCAS, which we summarize as the following exchange of messages between Website (W), Browser (B) and Scanning device (S) owned by user (U), using session identifying nonce $n_{recipient}$:



1. $W \longrightarrow B : W, n_B$
2. $B \longrightarrow S : W, n_B$ (visual channel)
3. $S \longleftrightarrow W$: authentication
4. $S \longrightarrow W : n_B, U$
5. $B \longrightarrow W : n_B$
6. $W \longrightarrow B$: authenticates as U

A mafia attack, first described as such by Desmedt et al [3], introduces a two-faced man-in-the-middle attacker A. With one face, A masquerades as a website W' to browser B and its user U, while with the other it masquerades as a browser B' to the original website W. In the example of Jenkinson et al [7], website W is a bank where user U has an account, whereas the attacker-controlled website W' is an online gaming forum where U has another account. The attacker A thus tricks the victim U into believing they're logging into forum W', whereas in fact they are giving A the credentials to log into bank W.



1. $W \longrightarrow A : W, n_A$
2. $A \longrightarrow B : W, n_A$
3. $B \longrightarrow S : W, n_A$ (visual channel)
4. $S \longleftrightarrow W$: authentication
5. $S \longrightarrow W : n_A, U$
6. $A \longrightarrow W : n_A$
7. $W \longrightarrow A$: authenticates as U

Note in step 3 that the browser B need not be aware that it is transmitting information to the scanning device S, as it may simply show a pre-rendered image of a visual code, so cannot detect that it is allowing authentication to a different service.

The practical implication of this attack is that, if a scanning authenticating device S holds credentials for multiple services, one of which an attacker can

successfully impersonate or modify, then the attacker could cause the S to authenticate the attacker's browser B' to a high-value service W, while the user imagines themself to be authenticating to a lower-value service W'. This is similar to a phishing attack but differs in that the user can't distinguish between being asked to authenticate to the bank or to the gaming forum, because visual codes are *not* human readable and all look alike to users.

In theory, this attack only applies to visual codes: users would not type their banking password into their gaming forum login page because the contexts don't match. But in practice the attack would also work, without visual codes, against victims who reuse passwords across high and low value web sites.

Our proposal is to bind some context to the visual code, thus alerting users when an authentication visual code is being presented in the wrong context.

3 Our solution: the SAVVIcode

The mafia attack is possible when a visual code does not have to match its context. If the scanner could recognise when a code is displayed in the wrong context, it could stop this attack. In the general case, this is a hard problem involving computer vision and contextual knowledge. We simplify it by observing that:

1. A scanner that can read a visual code must inherently be able to read "pixels" of a given size, as it must read the matrix of the visual code itself.
2. As in a visual code, a pixel matrix can be serialised into a string of bits.
3. Even if text is pixellated and blocky, humans can very easily read it – and will do so effortlessly.
4. It is difficult for a person to scan a visual code without looking at it, therefore any obvious text alongside the code will most likely be read by the user.

These observations suggest a solution initially outlined by the first author [8] whereby an image is placed above the visual code at the same block-resolution. It should appear as text or an image identifying the service requiring authentication.

At this point we assume that the remote service has an "identity keypair" which it uses to prove its identity—either through a Public Key Infrastructure, or without one as in Pico. This key can be used to sign the serialised image and the payload of the visual code. This signature should be included in the visual code, allowing a scanner to determine if the scanned image matches the one intended for display alongside the visual code. If we assume a hacker cannot, in reasonable time, forge a valid signature, then, under this scheme, *a valid visual code cannot be displayed without revealing the image that its creator wanted the user to see*. This ensures that a mafia attack is easily evident to the user before it occurs. We name the image described a *Serialised And Visually Verified Image Code* or SAVVIcode, and suggest that it should be hyphenated with the visual code type used, e.g. QR-SAVVIcode when used with a QR code.

It is worth noting that multiple systems could very easily choose to display the same image at the point of logging in. This is, however, no risk to the scheme as their authentication credentials would be different, and thus the authentication device would not authenticate to one as if it were the other. In other words, the malicious online forum operator may well say "I'm your bank" in the picture of his SAVVIcode (but that's not going to entice the user to log in to the forum); what he cannot do is to modify *the bank*'s SAVVIcode to say "I'm the online forum", because then the bank's signature would not verify.

The picture below exemplifies what a QR-SAVVIcode might look like. It consists of a QR code containing a payload, a public key and a signature, and an image above containing text and a number of patterns that may be used to detect the bounds. As the QR code itself contains finding patterns which will align the reader, only rudimentary additions should be required on the SAVVIcode.

4 Challenges and future work

Many visual codes, including at least QR-code, Aztec, Data Matrix and Maxicode, use Reed-Solomon error correction to compensate for inaccuracies in the decoded bit-stream,[1] thus allowing a significantly less than perfect scan to recover the original contents. For a SAVVIcode to be effective, however, we don't want error correction to allow an attacker to pass a modified code for a genuine one, so we must be careful.

To clarify, the attack would consist of taking the bank's SAVVIcode, whose human-readable text says "I'm the bank", then modifying that text to say "I'm the online forum" but with sufficiently few pixel changes that the error correction would still accept it as a slightly noisy version of "I'm the bank", thus allowing

[1] Another mechanism used by visual codes to improve scanning accuracy is to use encodings that break up large contiguous blocks of the same colour. The SAVVIcode does not use this method for resynchronisation because we want the bitmap to be immediately readable, even without meaning to, by the person scanning the code. We thus want a high-contrast bitmap in which the black ink stands out against a background of white space.

the signature to verify. The attacker would have then succeeded in persuading the user that they are logging in to the online forum (because the text says so and the digital signature verifies OK) while instead they'd be logging the attacker into the bank. We must absolutely prevent this.

We propose two approaches to implement a SAVVIcode. Their trade-offs should be evaluated and compared after writing prototype implementations. In both cases a free-form bitmap is drawn in an area above the main visual code, within a border with reference markers, as in the previous picture. The recognition software that turns the main visual code image into a boolean matrix is modified to recognize and digitize this extended region as well.

In the first approach, the free-form bitmap is also compressed and appended to the original payload; the whole payload is then digitally signed before being encoded in the main visual code. The SAVVIcode thus contains two versions of the bitmap: one that is also human-readable but whose integrity is not digitally protected, and one embedded in the code and digitally signed. The recognition software must compare the two versions and the challenge is to ensure that all fraudulent modifications of the human-readable version are detected by the recognizer, while allowing for some amount of non-malicious bit errors.

In the second approach, a detached error detection and correction (EDC) code for the free-form bitmap is generated. The concatenation of the serialized bitmap, the detached EDC code and the original payload is digitally signed. The concatenation of the detached EDC code, the original payload and the digital signature (but not the serialized bitmap) is then encoded in the main visual code. The recognition software thus extracts the bitmap, the EDC, the original payload and the signature; then applies the EDC to the bitmap to correct errors; then verifies the signature. This approach seems at first more robust, and carries only one copy of the bitmap, but it is still vulnerable to the same attack. A malicious adversary could subtly change the bitmap in a way that tricked the human viewer; but then the EDC would "undo" those changes and allow the verification to pass, yielding a false accept and thus a successful attack.

Fine-tuning the false accept (fraud) rate against the false reject (failure to admit honest customers) rate is, as ever, going to be the crucial security trade-off. The two approaches must be evaluated experimentally against malicious alterations to determine which one offers the best characteristics.

Although the SAVVIcode makes the visual code partially human-readable, a structural limitation of our strategy is its reliance on the alertness of the user. A "rushing user" (the kind that wants to get on with their real work and presses the OK button no matter what the annoying dialog box says) may well scan the code without paying any attention to the bitmap. We recognize this limit. The SAVVIcode is not an absolute defence against the man-in-the-middle attack on visual code authentication, but we believe it is still an improvement on the status quo: even though users may not pay attention to the bitmap, it is cognitively difficult to scan the visual code without reading it (as in "Do not think of an elephant!"). The SAVVIcode thus makes it difficult to mount the MITM attack without someone noticing.

5 Conclusions

Most visual code authentication systems (Pico excepted [7]) are vulnerable to middleperson attacks. The SAVVIcode adds an integrity-protected human-readable bitmap to the visual code. It is hard for a person to scan a SAVVIcode without reading this bitmap and thus, while the scheme relies on the user's cooperation, such cooperation happens almost automatically.

While the protection offered by the SAVVIcode does not claim to be absolute, it might prove remarkably effective compared to its modest cost of deployment. We believe its adoption would greatly reduce the impact of the inherent vulnerability to Mafia relay attacks that affects most existing visual schemes.

Acknowledgements

We are grateful to the Pico team for their feedback and to Andy Rice for helpful discussions on visual code scanning technology.

The Pico team is also working on an alternative "augmented reality" approach in which the human-readable tag is displayed by the scanner rather than being shown alongside the visual tag.

References

1. L. Batyuk, S.A. Camtepe, and S. Albayrak. Multi-device key management using visual side channels in pervasive computing environments. In *Proc. BWCCA 2011*, pages 207–214, Oct 2011.
2. J.J.L. Cobos and P.C. De La Hoz. Method and system for authenticating a user by means of a mobile device, September 4 2012. US Patent 8,261,089.
3. Yvo Desmedt, Claude Goutier, and Samy Bengio. Special uses and abuses of the Fiat-Shamir passport protocol. In *Proc. CRYPTO '87*, LNCS. Springer, 1987.
4. D.B. DeSoto and M.A. Peskin. Login using QR code, August 22 2013. US Patent App. 13/768,336.
5. Ben Dodson, Debangsu Sengupta, Dan Boneh, and Monica S. Lam. Secure, consumer-friendly web authentication and payments with a phone. In *Proc. Conf. on Mobile Computing, Applications, and Services (MobiCASE10)*, 2010.
6. Steve Gibson. Secure quick reliable login. `https://www.grc.com/sqrl/sqrl.htm`.
7. Graeme Jenkinson, Max Spencer, Chris Warrington, and Frank Stajano. I bought a new security token and all I got was this lousy phish—Relay attacks on visual code authentication systems. In *Proceedings of Security Protocols Workshop 2014*, LNCS. Springer, 2014.
8. Jonathan Millican. Implementing Pico Authentication for Linux. Undergraduate final year dissertation, May 2014.
9. Frank Stajano. Pico: No more passwords! In *Proc. Security Protocols Workshop 2011*, LNCS, pages 49–81. Springer, 2011.
10. Roland M. Van Rijswijk and Joost Van Dijk. Tiqr: A novel take on two-factor authentication. In *Proc. LISA'11*, pages 7–7. USENIX Association, 2011.

PassCue: the Shared Cues System in Practice

Mats Sandvoll, Colin Boyd, and Bjørn B. Larsen

Norwegian Institute of Science and Technology, Trondheim, Norway
matssandvoll@gmail.com, colin.boyd@item.ntnu.no,
bjorn.b.larsen@iet.ntnu.no

Abstract. Shared Cues is a password management system proposed by Blocki, Blum and Datta at Asiacrypt 2013. Unlike the majority of password management systems Shared Cues passwords are *never stored*, even on the management device. The idea of the Shared Cues system is to help users choose and remember passwords in a manner proven to avoid brute force searching under reasonable assumptions.
Blocki *et al.* analysed Shared Cues theoretically but did not describe any practical tests. We report on the design and implementation of an iOS application based on Shared Cues, which we call PassCue. This enables us to consider the practicality of Shared Cues in the real world and address important issues of user interface, parameter choices and applicability on popular web sites. PassCue demonstrates that the Shared Cues password management system is useable and secure in practice as well as in theory.

1 Introduction

Passwords are used to secure valuable data for banking, voting, mail, social networks, commerce and enterprise resources. The typical user has multiple password protected accounts and needs to manage each password. As the number of accounts increases, the management of passwords gets complicated. As a consequence, many users tend to adapt weak password management schemes which can significantly reduce the security of the system.

This problem has become widely understood in recent years and several password management schemes developed, in addition to methods for selecting passwords in a manner which can help users to remember the correct password for a given account. However, until recently there has been no systematic method to compare password management schemes with regard to either their usability or their security. At Asiacrypt 2013 Blocki *et al.* [4] proposed a mathematical model to address this deficiency. This model, which we call the *BBD model* from now on, allows a quantitative comparison of both usability and security based on assumptions about human memory and adversary capability.

A fundamental assumption of the BBD model is that persistent memory on user devices (hard disks, tokens, mobile devices, cloud storage) is insecure. This assumption rules out many popular password management systems as fundamentally insecure and requires that users store passwords in their own human memory. Some may regard this assumption as too pessimistic and unnecessary,

but experience shows that it is realistic in general. Moreover, if a high level of security and usability is achievable with this assumption then it is surely preferable to assume that it holds. Given this assumption, the BBD model can be used to compare systems based on two criteria.

Usability is measured by the expected number of *extra rehearsals* required by a system. Extra rehearsals are reminders to the user memory which are additional to those that automatically occur when a user logs on to an account.

Security is measured by the adversary expected effort to crack a target password which includes three scenarios:
1. the attacker is offline and has access only to persistent memory;
2. the attacker has the ability to test passwords online up to a given limit;
3. the attacker has access to one or more passwords of the user (which in general can be related in some way).

Having defined and explored their model, Blocki *et al.* [4] proposed a password management scheme called *Share Cues*. Shared Cues provides a good level of security and usability as measured in the BBD model. However, despite a thorough theoretical treatment in the model, until now it has not been implemented. It has therefore been unclear whether it could successfully be used in practice or whether there are obstacles to solve.

The main object of this paper is to determine how Shared Cues can be designed and implemented in software in a manner that is both usable and secure. We report on an implementation of Shared Cues as an iOS application — we call our application PassCue. We solve the following specific issues which arise in making the Shared Cues scheme into a real-world system.

- We choose suitable parameter values for the number and combinations of private and public cues to achieve an acceptable balance of usability and security.
- We identify and solve problems with password composition policies which would otherwise prevent the Shared Cues system from being used with many popular websites.
- We identify suitable ways to generate public and private cues for Shared Cues on a typical mobile device.

Outline Section 2 presents the background including the security and usability challenges with password management. Section 3 presents the Shared Cues password management model. Section 4 covers the design of PassCue. Section 5 presents the analysis of the designed system.

2 Background

This section presents key concepts required to understand the design and analysis of password management schemes, in particular the Shared Cues scheme.

2.1 Password Management Schemes

Informally we can regard a *password management scheme* as any method for creating passwords intended to be stored in human memory. In the BBD model such a scheme includes both a *generator* and a *rehearsal schedule*, the latter being a necessary part of defining usability.

Strong passwords are hard for attackers to guess, but they are also hard for users to remember. This results in the inherent trade-off between usability and security [9]. Smith notes, "the password must be impossible to remember and never written down" [22]. Wilderhain *et al.* [11] describe 15 different schemes and compare them in terms of usability and security. Many password management schemes are vague and unclear [11] in their description which may lead the user to select a weak password.

Following Blocki *et al.* [4], we highlight five password management schemes; *Reuse Weak*, *Reuse Strong*, *Lifehacker*, *Strong and Independent* and *Randomly Generated* described in Table 1. We will use these for comparison later.

Table 1: Password Schemes

Scheme	**Password creation**	**Use**	**Example**
Reuse Weak	Select a word w randomly from a dictionary	w for all accounts	*horse*
Reuse Strong	Select four words $w_1w_2w_3w_4$ randomly from a dictionary	$w_1w_2w_3w_4$ for all accounts	*appledoghorseblue*
Lifehacker	Select three words $w_1w_2w_3$ randomly as base. Use a derivation rule $d()$ to derive a string from account name	$w_1w_2w_3d(A_{name})$ unique for each account	*appledoghorsefac* $d(A)$ is first three letters of account name
Strong Random and Independent	Select four words $w_1w_2w_3w_4$ randomly from a dictionary	$w_1w_2w_3w_4$ unique for each account	*appledoghorseblue*
Randomly Generated	Generate a random password using a password generator	*random* unique for all accounts	*bcxtabf2owale89n*

2.2 Person-Action-Object

The human memory is limited in remembering sequences of items [14] and has a short-term capacity of approximately seven items [20]. It has been shown that humans remember well when the sequence of items is familiar chunks in words or familiar symbols. The human memory is semantic [4], so in order to facilitate usability the number of chunks should be minimized. The human memory can be characterised as associative, which means that memories are associated with

other memories. We are more capable at remembering information we can encode in multiple, redundant ways [28].

Mnemonic techniques are methods to help retaining memories and many such techniques exist. The effectiveness of a technique is individual, and common devices include music, names, pictures, expressions, words and models. It has been shown that users can exploit mnemonic strategies in order to remember passwords [3]. Blocki notes, "Competitors in memory competitions routinely use mnemonic techniques which exploit associative memory" [4].

Models for the human memory [17,26,1] differ in some details, but all of them model an associative memory with cue-association pair. Cues are the context in which a memory is created. The cue can be a sound, surrounding environment, a person, an object or anything that is associated with a specific memory. In order to remember the password the user associates it with a cue. The human memory is lossy, so the cue strength is often reduced as time passes. To prevent the user from forgetting the cue-association pair, it is essential to create strong associations and maintain them over time through rehearsal. The rehearsal process should be as natural as possible, and a part of the users normal activities so the usability is not decreased.

A Person-Action-Object (PAO) story is a mnemonic technique that consists of a picture of a person, an action and an object. The person could, for example, be Elvis Presley with the action *shooting* and the object *banana*. The resulting PAO story would be; "Elvis Presley is shooting a banana". The PAO pattern is very simple and with randomly generated persons, actions and objects, the story can be quite surprising [4]. It has been shown that memorable sentences use strange combination of words in a common pattern [7]. A modified version of the PAO mnemonic technique is used in Shared Cues. The modified version also includes a picture of a place or a background type image.

2.3 Password Composition Policy

Password composition policies (PCPs) are used by system administrators, usually in order to prevent users from selecting weak passwords. While it is commonly understood that PCPs make passwords harder to guess, that is not always the case [16,18] because a PCP not only affects the password, but also the user behaviour. A policy that ensures a complex password could also lead users to write down the password, adapt insecure management schemes [18] or be adverse to password changes. Websites differ in their PCP, but common guidelines include use of: both lowercase and capital letters; a combination of letters, numbers and special characters; passwords with minimum length.

PCPs vary and in some cases it is not possible to follow common guidelines. The German bank Berliner Sparkasse and the financial services cooperation Fidelity are examples of websites that only allow alphanumerical characters. The Bank of Brazil only allows numbers, and in Virgin Atlantic the passwords are not case-sensitive [8]. Websites also put restrictions on which type of special characters can be used and restrictions on maximum length. The banking com-

pany Charles Schwab has a password length limit of eight characters [15] and Outlook.com has a maximum password length of 16.

PCPs can conflict with a password management scheme by disallowing passwords generated in the scheme. Problems can be that the password does not include symbol types demanded by the policy or that it is too long (or possibly too short). This can be a serious problem when implementing a password management scheme in the real world. Later we show how we solve this problem in PassCue and analyze its implications.

3 BBD model and the Shared Cues scheme

The BBD model is based on a usability assumption about the human memory. The assumption is that a user who follows a specific rehearsal schedule will maintain the corresponding memory. This assumption has been proven successful in different studies by using various rehearsal schedules [23,3,4].

The rehearsal schedule is necessary for keeping the cue-association pair $(\hat{c}, \hat{a})$ in associative memory. A rehearsal schedule is sufficient if the user maintains the cue-association $(\hat{c}, \hat{a})$ by following the rehearsal schedule [4]. A rehearsal schedule is simply a sequence of times for each cue; the association with that cue must be rehearsed within a window (for example, 1 day) around that time.

Definition 1 ([4]). *A password management scheme consists of:*

1. *a generator which is a randomised function of the user's knowledge (and can also depend on the rehearsal schedule and user's logon actions) and outputs a set of cues $c_1, \ldots, c_m$ and associated passwords $p_1, \ldots, p_m$; and*
2. *a rehearsal schedule which the user must follow for each cue.*

3.1 Usability Model

The usability depends on the rehearsal requirement for each cue, visitation schedule for each site and the number of cues the user needs to maintain. The usability of the password scheme is measured in numbers of extra rehearsals, $\mathrm{X}_{t,\hat{c}}$, the user needs to perform to maintain the associations in memory. Block *et al.*[4] consider three rehearsal assumptions;

Constant Rehearsal Assumption (CR): The rehearsal schedule is at a constant rate. For example, rehearsals can be scheduled one per day.

Expanding Rehearsal Assumption (ER): The rehearsal schedule follows the pattern 2^i. For example, rehearsals can be scheduled on day 1, day 2, day 4, day 8, and so on.

Squared Rehearsal Assumption (SQ): The rehearsal schedule follows the pattern i^2. For example, rehearsals can be scheduled on day 1, day 4, day 9, day 16, and so on.

The CR assumes that memories are not strengthened every time the user rehearses, hence the user must rehearse every σ days. The ER and SQ in contrast, assume that memories strengthen every time the user rehearses and therefore the number of rehearsals decreases over time. Studies have shown that the availability of a memory is dependent on recency [2] and the pattern of previous rehearsals. The difference between SQ and ER is not significant, but ER is consistent with known memory studies [23,27]. Based on these observations we follow Block *et al.* [4] and use ER as the basis of our PassCue implementation.

3.2 Security Model

Many breaches occur because users choose to put personal information in their password, such as hobbies and birth dates, assuming that this information is private. The BBD model assumes that the adversary has background information about the user, in addition to the public cues $c_1, \ldots, c_m$ stored in persistent memory and the details of the password management scheme. The secrecy of the passwords lies in the random string used to generate the set of passwords — in the Shared Cues scheme this corresponds to the randomly chosen association between public cues and secret cues.

Measuring the security and password strength can be performed in different ways. Min-entropy and password meters, which are often used to measure password strength, have in several studies been proven to be a weak measure of password security [4,6]. The BBD model measures the security of a password management scheme by estimating the cost of guessing, cracking, the password by a potential attacker. Three types of attacks are considered; online attack, offline attack and plaintext leak attack.

In the Shared Cues system, as its name implies, cues are shared across accounts in order to improve the usability because the number of cue-associations, α, is reduced and the rate of natural rehearsals increases. A (n, l, γ) sharing set family uses n separate cues (pictures) where each password consists of l public/private cue pairs and no more than γ cues are shared between any two passwords. Theorem 1 is the main security result proven by Blocki *et al.*[4], which shows that public cues can be securely shared across accounts if the public cues $\{c_1, \ldots c_m\}$ are a (n, l, γ) sharing set family.

Theorem 1 ([4]). *Let $\{c_1, \ldots, c_m\}$ be a (n, l, γ) sharing set of m public cues from the generator algorithm. If each association is chosen uniformly at random then the success probability of an adversary, δ, is limited by*

$$\delta \leq \frac{q}{\alpha^{l-\gamma r}} \text{ where}$$

- *q is the number of offline guesses performed by the adversary;*
- *l is the number of cues used per password;*
- *γ is the maximum number of shared cues between passwords:*
- *α is the number of number of cue-associations.*

4 Design

This section presents the design choices, required to implement the Shared Cues systems into the real implementation PassCue. We highlight the factors which affect the security and usability of PassCue.

4.1 Public Cues

The PassCue system uses n public cues supplied by the user. Each of the public cues consists of a picture of a known person and a picture of a known place/location/background. One action and one object are randomly selected from a large set and assigned to the public cue. The example in Figure 1 illustrates how a public cue is assigned an association. In the example, the user selected a public cue that consists of a trampoline and his grandmother. The randomly selected action is *surfing* and the object is *banana*. The action and object represents the association of the cue. The user should imagine his grandmother surfing a banana on the trampoline in his garden. The association is private and only displayed the first time a public cue is used. After initialization, the association is deleted and non-retrievable. The user must keep the association in associative memory. The next time this particular cue is used, only the public cue with a picture of a trampoline and his grandmother will be visible. The user must remember what the grandmother did on the trampoline.

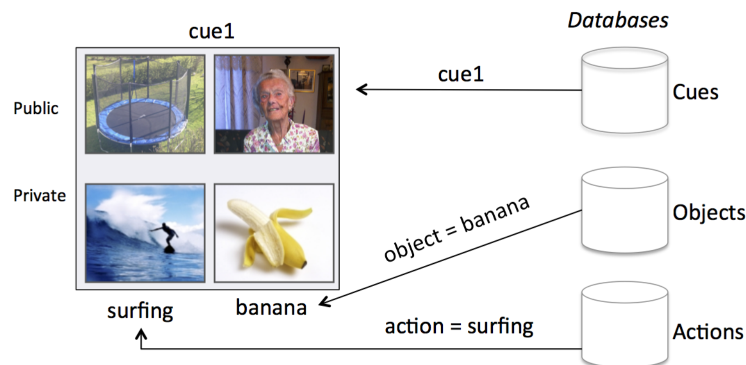


Fig. 1: Creating a PAO-story

Since the association is only stored in the associative memory of the user, it is important that the associations are strong. Studies have shown that people tend to create stronger associations when they see pictures of people they know and a place where they can imagine an action taking place [12]. In the PassCue application pictures come from from the user phone when initializing the public cues. In order to create strong association the user is forced to use private pictures from the photo library.

4.2 Sharing Set

The distribution of the public cues to each account is performed using (n, l, γ)-sharing sets where n is the total number of public cues, l is the number of cues used for each account, and γ is the maximum number of cues that can be shared between two accounts. A sharing set is selected for each new account.

PassCue strives to provide a usable and secure way to manage passwords. There is a clear usability-security trade-off when defining l and γ. A large l requires the user to invest a lot of effort to retrieve a password, but as the usability decreases, the security increases, and a large l provides higher security. PassCue is designed with $l = 4$ because it provides a reasonable compromise between security and usability.

The sharing of the public cues between the accounts is essential in order to maintain the required associations without forcing the user to invest additional time. γ should be close to l in order to utilize the sharing property and increase usability. In order to minimize the number of extra rehearsals, the PassCue application is designed with $\gamma = 3$. In order to preserve the security, the number of cues must be larger than the maximum number of cues two accounts can share to avoid two accounts having identical passwords.

The number of public cues, n, does not affect the security, but it determines the number of accounts/passwords that can be generated with PassCue. When $l = 4$ and $\gamma = 3$, the number of unique passwords that PassCue can generate is 35 if $n = 7$. For $n = 8$ it could accommodate 70 different passwords. This may be sufficient for the some users, but as the technology develops the need for new accounts increases. Increasing the number of cues to $n = 9$ enables PassCue to generate 126 unique passwords, which should be enough for the active users. Therefore PassCue is designed with a $(9, 4, 3)$-sharing set.

It seems reasonable to assume that the user has nine pictures of a person and nine pictures of a background on his phone, or that it is obtainable using the phone's camera. The (9,4,3)-sharing set is incrementally designed. This means that the first six accounts all share cues 1, 2 and 3. After creating six accounts in PassCue, the user already knows all the cue-association pairs. The next 120 passwords generated use cues the user already knows. This makes it easier for the user to start using PassCue and it significantly increases the usability.

4.3 Rehearsal Schedule

When a cue is used for the first time, a rehearsal schedule is created. The objective of the rehearsal schedule is to assure that the association assigned to the public cue is kept in the associative memory of the user and not forgotten. The user is notified according to the rehearsal schedule when he needs to rehearse a specific cue-association in order to maintain the memory of the association.

Designing PassCue with the CR rehearsal schedule would force the user to practice all the cue-association pairs every day, and significantly reduce the usability of the system. If PassCue creates strong cue-association pairs, it is reasonable to assume that daily rehearsal is not required in order to maintain the

association. The difference between SQ and ER is not significant, but ER is consistent with known memory studies [23,27] and was therefore chosen as the rehearsal schedule for PassCue.

4.4 Password Composition

It is important to understand that, because the passwords are never stored, the method for choosing passwords can only be suggested by the PassCue application and cannot be enforced. The design of PassCue features such as the rehearsal schedule is based on the assumption that users will apply the recommended method. Moreover, the security and usability analysis is valid only when users follow the method described in this subsection.

The password is created using the first three letters of the action and object for each of the associations. The first character is capitalized in order to cope with common PCPs. The whole action/object words could have been used but this is not recommended because passwords can be rejected due to restrictions on either maximum password length or the use of dictionary words. In any case, this results in no loss of entropy in most cases.

Figure 2 shows how private associations can be used to derive a password. The associations for each of the public cues are only stored in associative memory and are not available for a potential attacker. Account *Gmail* uses cue 1, 2, 3 and 4. All the cues consist of two public pictures and a private association of two pictures. In cue 1, the user selected a trampoline and his grandmother as public pictures. For cue 2, he selected a picture of a toilet and his mother. In cue 3, a picture of his garden was selected as background picture and his father as person picture. Cue 4 uses a picture of a hallway and his sister. The first three letters of each action and object are used to create a password for account *Gmail*.

Due to remote system PCPs, embedding numbers and special characters in the PassCue application would in some cases make the generated passwords unusable. As a consequence, PassCue is designed to have an account note field where the user can specify numbers, special characters and other information in order to fulfill the PCP of a specific site. This information is displayed in plaintext and is therefore assumed accessible by a potential attacker in the analysis.

4.5 Association Set

The associations are only displayed when a cue is initialized and the user must use the public pictures to retrieve the associations from memory. For the first part of the *Gmail* password the user must imagine his grandmother on the trampoline, and retrieve the memory *surfing* and *banana*. The second part is his mother on the toilet *presenting* a *dog*. The third cue is his father in the garden *drawing* a *bunny*. For the last part of the *Gmail* password, cue 4 displays a picture of the user's sister and a hallway. The user must ask himself; "What did my sister do in the hallway?". The answer from associative memory is *inspecting* a *gift*. By combining the three first letters in the action and object for each of the four cues, the *Gmail* password is derived as "Surbanpredogdrabuninsgif". Since

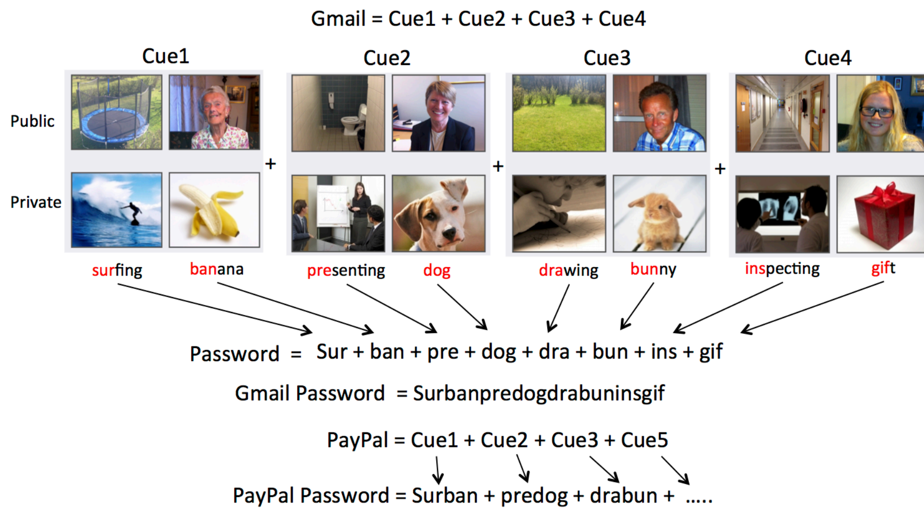


Fig. 2: Deriving password from actions and objects

account *Gmail* and *PayPal* share cue 1, 2 and 3, the 18 first letters are the same and the last six differ.

Section 3.2 and Theorem 1 show that the security in the BBD model is dependent on the association set size. Larger set size means larger password space and higher security. The association size is also bound to the password creation strategy. The security bound on PassCue assumes that each of the actions and objects is uniquely encoded and that each of the actions and objects, given a password creation strategy, produce a unique string.

All the actions and objects must be supplied with the application and each represented by an image. In order to provide reasonable security while not demanding too much storage and system resources, the action and object size in PassCue is 200. How the association set size affects the security is presented in Section 5.2.The association set size can easily be extended in cases where higher security is required, or lowered to save system resources.

4.6 Specification

The PassCue specification is based on the above design choices. The design and implementation of PassCue fulfils the specification given in Table 2.

5 Analysis

This section covers the analysis of the PassCue implementation. The first part examines the usability of PassCue by calculating the additional time the user

Table 2: PassCue Specification

System
iOS application
Application data must be stored in persistent memory
A set of action and object with image must be included
Stand alone application with no external connection
Usability
(9,4,3)-sharing set
9 person images and 9 background images must be supplied by the user
4 cues are used for each account
Maximum 3 cues can be shared between two accounts
Support up to $m = 126$ accounts
Possible to remove accounts
The user must be notified when to rehearse according to the rehearsal schedule
Account notes to cope with password composition policies
Possible to reset a cue-association pair if association is forgotten
Natural rehearsing effects the rehearsal schedule
Use the ER rehearsal schedule
Security
Random numbers must be cryptographically secure
No cue-associations pairs are possible to retrieve after initialization
Action-Object set size must be 200

must invest in rehearsing the cues. The second part covers the security analysis: password entropy is calculated and the resistance against plaintext leak attacks, online attacks and offline attacks is evaluated.

5.1 Usability

The most important part of the usability of PassCue is the additional time the user must invest in order to maintain all the cue-association pairs. A rehearsal schedule is created for each cue-association pair and updated every time the cue is used for log in. Given the ER rehearsal schedule a cue-association pair must be rehearsed at least eight times per year in order maintain the cue-association pair in the user's associative memory. PassCue uses a (9,4,3)-sharing set, where maximum three cues can be shared between two accounts. This property significantly improves the usability, and reduces the extra amount of time the user must invest in rehearsing the cue-association pairs.

Table 3 shows the cue distribution for the first ten accounts in PassCue (9,4,3). The sharing set has been incrementally designed in order to make it easy for the user to start using PassCue. As given in Table 3, the first six accounts use cue 1, cue 2 and cue 3. After the first account is created, the user only needs to remember one new cue each time a new account is created. After creating

the sixth account, the user knows all the nine cue-association pairs. The next $126 - 6 = 120$ accounts use cues which the user already knows.

Table 3: Sharing set for the first 10 accounts

	Part 1	Part 2	Part 3	Part 4
Account	Cue	Cue	Cue	Cue
1	1	2	3	4
2	1	2	3	5
3	1	2	3	6
4	1	2	3	7
5	1	2	3	8
6	1	2	3	9
7	1	2	4	5
8	1	2	4	6
9	1	2	4	7
10	1	2	4	8

Creating a new account forces the user to rehearse some of the cues used in other accounts. Creating passwords for 25 accounts in PassCue would be sufficient yearly rehearsal for all the nine cues. Table 4 shows how many times each of the cue-association pairs are rehearsed when creating 25 accounts in PassCue. The average number of extra rehearsals, additional time invested in PassCue, for a normal user is close to zero.

Table 4: Number of cue rehearsals performed when creating 25 accounts

Cue	1	2	3	4	5	6	7	8	9
Times Rehearsed	25	21	10	10	7	7	7	7	6

5.2 Security

Online Attack. Most online services have a k-strike policy where the user is locked out after trying k wrong passwords. The number of guesses the attacker can perform online is limited to km, where m is the total number of passwords. Applying Theorem 1, the probability that the attacker can successfully retrieve the password (n, l, γ) for a sharing set of m public cues produced by PassCue is

$$P_r = \frac{km}{(|Actions| \times |Objects|)^{l-\gamma r}}, \tag{1}$$

where r is the number of plaintext password leaks and k is the number of guesses permitted.

PassCue resistance against online attacks is highly dependent on previous plaintext password leaks because cues are shared between accounts. Assuming $k = 3, r = 0$ and $m = 126$ gives $\Pr(\text{Attacker retrieves password}) = \frac{3\times126}{(200\times200)^{4-3\times0}} = 1.48 \times 10^{-16}$. The probability that the attacker is able to guess the *Gmail* or *PayPal* password from Figure 2, assuming no previous password leaks $r = 0$, is 1.48×10^{-16}. For the other password management schemes, P(Attacker retrieves password) is calculated using Equation 2, where H is the password entropy, $k = 3$ and $m = 126$:

$$P_r = \frac{km}{2^H} \tag{2}$$

Table 5 presents the probability that an account is compromised as a result of an online attack for various password management schemes. The online security of PassCue (9,4,3) is calculated using equation (1). PassCue (9,4,3) provides high resistance against online attacks even if one password is leaked, but online security breaks down if two plaintext passwords are leaked.

Table 5: Online Security Results

Scheme	m	r=0	r=1	r=2
PassCue (9,4,3)	126	1.47656×10^{-16}	0.00945	1
Reuse Weak	126	0.019	1	1
Reuse Strong	126	2.36×10^{-15}	1	1
Lifehacker	126	2.69×10^{-15}	1	1
SRI	126	2.36×10^{-15}	2.36×10^{-15}	2.36×10^{-15}
Random Generated	126	4.75×10^{-23}	4.75×10^{-23}	4.75×10^{-23}

Offline Attack The offline attacker has access to the hash of the user's password. There are many examples of online services that have been hacked and the cryptographic hash released [25,24,19]. The attacker can guess the password, compute the hash and compare it to the leaked password hash. Hashcat [10] is state of the art cracking software optimized for speed using the GPU. MD5 is considered the fastest of the common hash algorithms. The estimates below assume that MD5 is used in order to estimate a worst case security scenario. The estimates are based on renting computing capacity on the Amazon Elastic Compute Cloud (EC2) and assume that Hashcat can run 2100 million MD5 guesses per second on EC2 [5,10].

Given that there are no leaks ($r = 0$), each password generated by PassCue is equally likely, so there is no better searching strategy than brute-force search.

The number of possible passwords is $(|actions| \times |objects|)^l$ and the time to find search through all passwords in seconds is

$$T_{crack} = \frac{(|actions| \times |objects|)^{l-\gamma r}}{guesses/sec} \tag{3}$$

The guaranteed cracking time in seconds for a PassCue password hashed with MD5 using Hashcat with 2100 million guesses per second, assuming (9,4,3)-sharing set, association set of 200 and $r = 0$ is:

$$T_{crack} = \frac{(200^2)^4}{2100 \times 10^6} = 1219047619s$$

Renting the *cg1.4xlarge* GPU instance on EC2 costs $2.1 per hour. Thus the estimated cracking cost in USD for a PassCue password if $r = 0$ is:

$$C_{crack} = 1219047619 \times \frac{2.1}{3600} = 711111$$

In order to crack a PassCue (9,4,3) password, given no previous password leaks $r = 0$, the attacker must invest $\$711,111.1$ and it would take 14109 days, over 38 years, on a single GPU on Amazon EC2. The full analysis of the PassCue configurations is given in Table 6.

Table 6: Offline Security Results

	r=0		r=1	
Scheme	Time (*days*)	Cost (*$*)	Time (*days*)	Cost (*$*)
PassCue (9,4,3)	14109	711111.1	2×10^{-5}s	≈ 0
Reuse Weak	$< 10^{-5}$s	≈ 0	≈ 0	≈ 0
Reuse Strong	881.82	44444.15	≈ 0	≈ 0
Lifehacker	774.95	39057.6	≈ 0	≈ 0
SRI	881.82	44444.15	881.82	44444.15
Rand. Gen.	4.39×10^{10}	2.21×10^{10}	4.39×10^{10}	2.21×10^{10}

PassCue (9,4,3) provides significantly higher offline security compared to the *reuse weak* scheme when $r = 0$. PassCue (9,4,3) provides almost twice the offline security compared to the *reuse strong*, *lifehacker* and *strong random and independent* scheme for $r = 0$. After one password leak occurs, the offline security for PassCue (9,4,3), *reuse strong* and *lifehacker* breaks down.

In the full version of this paper [21] we provide also analysis for different versions of PassCue with parameters (43,4,1) and (60,5,1). This shows that PassCue (43,4,1) maintains a certain level of protection for $r = 1$ and PassCue (60,5,1) remain very secure. The *strong random and independent* scheme and the *randomly generated* scheme provide high security regardless of password leaks. Even if PassCue does not provide as high security as the *strong random and independent* scheme, it is much more usable. For $r = 0$ the security is also comparable.

Effect of Password Composition Policies. If the online service enforces a PCP with maximum length, and the password must contain numbers and symbols, it can reduce the password entropy. This is because the extra numbers and symbols included in the password are displayed in plaintext in the account notes in PassCue.

We recommend, where possible, that the user chooses the three first letters of each action and object unless there is a restriction on maximum password length. If the number of cues per account is l and we use an equal number of characters for actions and objects, then the upper limit for the number of characters from each action and object is:

$$|\text{chars per act/obj}| = \frac{(|\text{max password length}| - |\text{chars PCP}|)}{2l} \tag{4}$$

Assume, for example, that the PCP that requires a mix of letters, numbers and symbols, and has a maximum password length of 20. One character will be used for the number and one for the symbol. As the number and the symbol will be displayed in plaintext in the account notes, only 18 characters are available for the rest of the password. If the user selects three characters from each action and object, only the first three cue-association pairs will fit. This will result in an entropy reduction of $\log_2(200^8) - \log_2(200^6) = 15.28$ bits. If the user selects two characters from each action object, there will be no entropy reduction. As a result, the user should use two letters from each action and object if the maximum password length is less than 26.

6 Conclusion

PassCue has been implemented as an iOS application, which can be used to log on to a system. PassCue uses a (9,4,3)-sharing set, ER as rehearsal schedule and account notes to cope with PCPs. PassCue has an association set size of 200, and the public cues are created using pictures supplied by the user. The PassCue application utilizes less then 1% of the CPU and only 5.9 MB of memory in idle state of an iPhone 5.

PassCue (9,4,3) proved to provide higher online and offline security compared to the popular used password management schemes *reuse weak*, *reuse strong* and *lifehacker*. Unlike the other methods assessed, PassCue (9,4,3) provides a strong level of online security when $r = 1$, but the offline security of PassCue (9,4,3) breaks down when one password plaintext leak occur. PassCue (9,4,3) requires no extra rehearsals in order to maintain the cue-associations in the associative memory of the user.

It would be useful to test the PassCue application on different types of users in order to improve the design and application experience. User feedback can provide valuable insight into how the application is used and could reveal unknown needs or user difficulties. It would be possible to add support for multiple rehearsal schedules and allow the user to select and specify a rehearsal schedule according to personal needs. It may also be possible to adjust the rehearsal schedule based on the actual visitation schedule for a particular user.

References

1. John R. Anderson, Michael Matessa, and Christian Lebiere. Act-r: A theory of higher level cognition and its relation to visual attention. *Human-Computer Interaction*, 12(4):439–462, 1997.
2. John R. Anderson and Lael J. Schooler. Reflections of the environment in memory. *Psychological Science*, 2(6):396–408, 1991.
3. Alan David Baddeley. *Human Memory: Theory and Practice*. Lawrence Erlbaum Associates, Hove, UK, 1990.
4. Jeremiah Blocki, Manuel Blum, and Anupam Datta. Naturally rehearsing passwords. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT (2)*, volume 8270 of *Lecture Notes in Computer Science*, pages 361–380. Springer, 2013.
5. Matthew Bryant. Amazon ec2 gpu hvm spot instance password cracking – hashcat setup tutorial. `http://thehackerblog.com/amazon-ec2-gpu-hvm-spot-instance-cracking-setup-tutorial/#more-576`, 2013. Retrieved 26.04.2014.
6. Claude Castelluccia, Markus Dürmuth, and Daniele Perito. Adaptive password-strength meters from markov models. In *NDSS*. The Internet Society, 2012.
7. Cristian Danescu-Niculescu-Mizil, Justin Cheng, Jon M. Kleinberg, and Lillian Lee. You had me at hello: How phrasing affects memorability. *CoRR*, abs/1203.6360, 2012.
8. Defuse. Password policy hall of shame. `https://defuse.ca/password-policy-hall-of-shame.htm`. Retrieved 10.03.2014.
9. Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier. Password strength: An empirical analysis. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 983–991, Piscataway, NJ, USA, 2010. IEEE Press.
10. Andrew Dunham. Password cracking on amazon ec2. `http://du.nham.ca/blog/posts/2013/03/08/password-cracking-on-amazon-ec2/`, 2013. Retrieved 26.04.2014.
11. Anne Wildenhain et al. Comparison of usability and security of password creation schemes. `https://www.cs.cmu.edu/~jblocki/Anne_Wildenhain_2012.htm`, 2012. Retrieved 07.02.2014.
12. J. Foer. *Moonwalking with Einstein: The Art and Science of Remembering Everything*. Penguin Books Limited, 2011.
13. Google. Creating a strong password. `https://support.google.com/accounts/answer/32040?hl=en`, 2013. Retrieved 26.04.2014.
14. G. J. Johnson. A distinctiveness model of serial learning. *Psychological Review*, 98(2):204–217, 1999.
15. Casey Johnston. Why your password can't have symbols—or be longer than 16 characters. `http://arstechnica.com/security/2013/04/why-your-password-cant-have-symbols-or-be-longer-than-16-characters/`, 2013. Retrieved 11.03.2014.
16. P.G. Kelley, S. Komanduri, M.L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L.F. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 523–537, May 2012.
17. Teuvo Kohonen. *Associative Memory: A System-Theoretical Approach*. Springer, 1977.
18. Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. Of passwords

and people: Measuring the effect of password-composition policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2595–2604, New York, NY, USA, 2011. ACM.
19. LinkedIn. An update on linkedin member passwords compromised. http://blog.linkedin.com/2012/06/06/linkedin-member-passwords-compromised/, 2012. Retrieved 16.02.2014.
20. George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97, March 1956.
21. Mats Sandvoll. Design and analysis of a password management system. Masters Thesis, NTNU, 2014.
22. Richard E. Smith. The strong password dilemma, 2002. Ch. 6.
23. L. R. Squire. On the course of forgetting in very Long-Term-Memory. *J Exp Psychol Learn J Exp Psychol Learn*, 15(2):241–245.
24. USA Today. Zappos customer accounts breached. http://usatoday30.usatoday.com/tech/news/story/2012-01-16/mark-smith-zappos-breach-tips/52593484/1, 2012. Retrieved 16.02.2014.
25. The Verge. Evernote resets all passwords after user information is stolen in security breach. http://www.theverge.com/2013/3/2/4056704/evernote-password-reset, 2013. Retrieved 16.02.2014.
26. D J Willshaw and J T Buckingham. An assessment of Marrs theory of the hippocampus as a temporary memory store. *Philos Trans R Soc Lond B Biol Sci*, 329(1253):205–15, 1990.
27. Woźniak and E. J. Gorzelańczyk. Optimization of repetition spacing in the practice of learning. *Acta neurobiologiae experimentalis*, 54(1):59–62, January 1994.
28. Jeff Yan, Alan Blackwell, Ross Anderson, and Alasdair Grant. Password memorability and security: Empirical results. *IEEE Security and Privacy*, 2(5):25–31, September 2004.

A PassCue

The public pictures in the following example figures are taken by the author and used with persons' permission. The private action and objects pictures are used with permission from morguefile.com.

Figure 3 shows the application screens for the initialization process. The initialization process is only required the first time the application is launched. The user is told to select a background image and select a person image for the nine required cues. The user is able to select pictures from the photo library or downloaded images. When the user push the *Select Background Image* button, an image picker screen is displayed and the user can select the appropriate picture. The user can quit the application in order to obtain the images or take the images with the camera on the phone.

The cue pictures are saved within the document directory in the application, and the path is saved in the database. If the images where to be deleted from the photo library, it will not affect the application. In this example the user selects a picture of the trampoline in his garden and a picture of his grandmother as the first cue. Once pressed the *Next*-button, the user can select images for cue number two. The user must continue the process until cue nine is initialized.

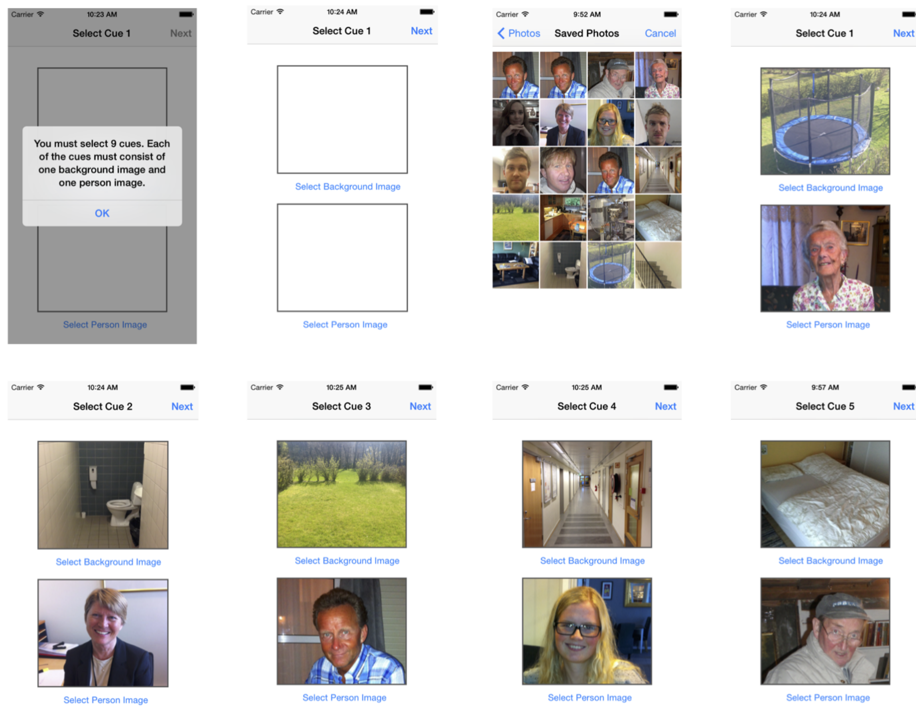


Fig. 3: Select Cue Images

When the user has selected images for all nine cues the top left screen in Figure 4 is displayed. The user can add an account by pushing the + button. The user must select an account name and write account notes if desired. As we saw in Section 2.3, many sites puts restrictions on the password selection in order to force the user to select a strong password. In this case, for the *Gmail* account, Google recommends using a mix of letters, numbers and symbols in the password [13]. The user inputs "23&." in the account notes field, and will use this when deriving the password. The account notes are displayed in plaintext and are assumed to be accessible to an attacker. How this affects the security is detailed explained in the security analysis in Section 5.2.

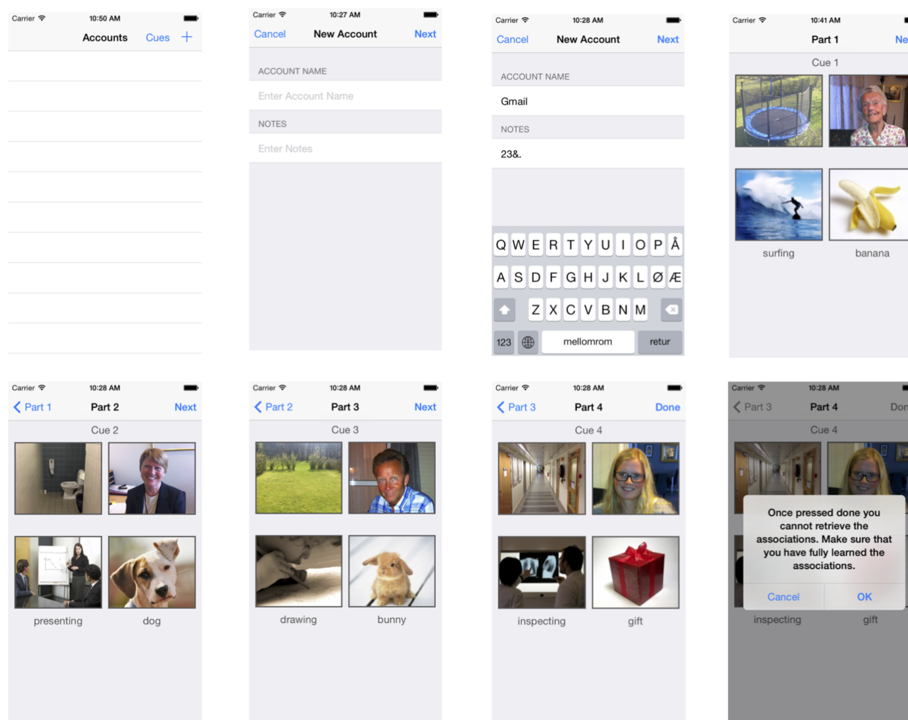


Fig. 4: Create New Account

When the *Next* button is pushed, the first cue and the randomly selected association is displayed. In the example in Figure 4 the user must imagine the following setting; "My grandmother is *surfing* a *banana* on the trampoline". *Surfing* and *banana* is the private part of the cue and will never be displayed after the cue initialization. *Surfing* and *banana* will be used to create the password. The public picture of the user's grandmother and his trampoline will later be used to trigger the association of *surfing* and *banana* from the users associative

memory. In cue two the user must reflect over the following story; "My mother is *presenting* a *dog* on the toilet". Cue three gives the following story; "My father is *drawing* a *bunny* in the garden". In cue four the user must imagine the following; "My sister is *inspecting* a *gift* in the hallway".

Once the user presses the *Done* button in part 4, a warning message alerts the user that the associations are non-retrievable after this step. A rehearsal schedule is created for cue 1, 2, 3 and 4. This is performed to ensure that the user does not forget the actions and objects associated with the cues.

Figure 5 shows how PassCue can be used to log in to a system. In this example, PassCue holds two accounts, *Gmail* and *PayPal*. If the user is to log in to the *Gmail* account, he selects the *Gmail* account and the account cues and notes are displayed. The user will use the cues in order to retrieve the associations from associative memory. The user must ask himself; "What did my grandmother on the trampoline?" and should remember that she was indeed "*surfing* a *banana*!". The next cue retrieves the association *presenting* and *dog*. Cue three reveals that "My father was *drawing* a *bunny* in the garden". The last cue was "My sister is *inspecting* a *gift* in the hallway".

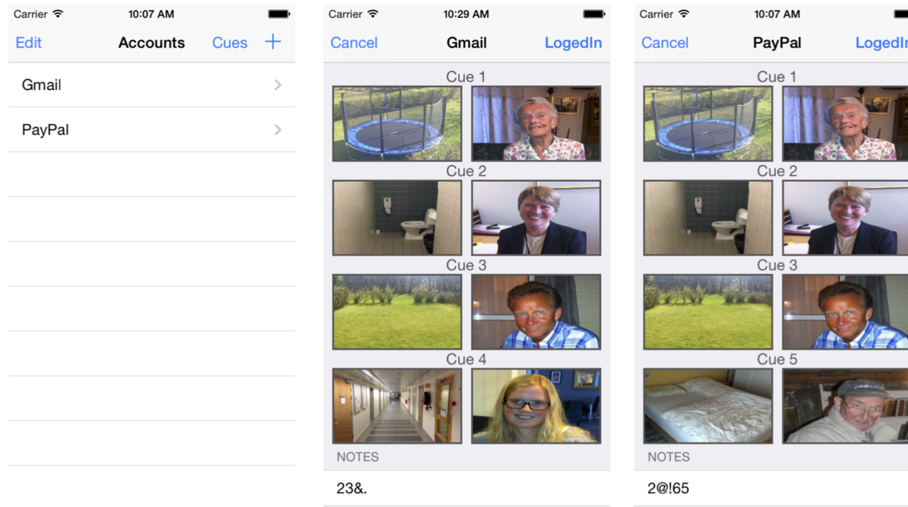


Fig. 5: Log in to *Gmail* and *PayPal*

In this example the user always uses the account notes as the first part of the password, and uses the three first letters from each action and object with capital first letter for all the action derived letters. The password for *Gmail* will be "23&.SurbanPredogDrabunInsgif". The user must press the *LogedIn* button for the rehearsal schedule to be updated. Once pressed *LogedIn* the application calculates a new rehearsal time for the involved cues according to the rehearsal

schedule. The *LogedIn* button is not connected to the *Gmail* online interface or any other online interface. The *Logedin* button is solely for use in the PassCue application in order to manage the rehearsal schedules. Pressing the *Logedin* button does not provide automatic login to the specified account. The user must derive the password using the cues, and press *LogedIn* for the application to update the rehearsal schedule. Cue five in Figure 5 gives the following setting; "My grandfather is *kicking* an *elephant* in the bed". Since *Gmail* and *PayPal* share the first three cues, parts of the password is identical. By using the same derive method as for the *Gmail* account, the password for *PayPal* is "2@!65SurbanPredogDrabunKicele".

If the user selects the *Cues* button from the main screen, he is able to see information about the cues. The first screen in Figure 6 shows the overview of the cues with pictures, and number or accounts. The user is able to choose one of the cues for more information. The cue information shows the next time for cue rehearsal, which accounts the cue is used in, and an option to reset the cue. The user can select one of the displayed accounts and the application will make a transition to the log in screen for the account, as shown in Figure 5. This is illustrated in Figure 5. In the event that the user forgets the action and object associated to a cue, the user can reset the cue. If the cue is reset, all accounts that use the particular cue is deleted, and a new association and rehearsal schedule is created for the cue.

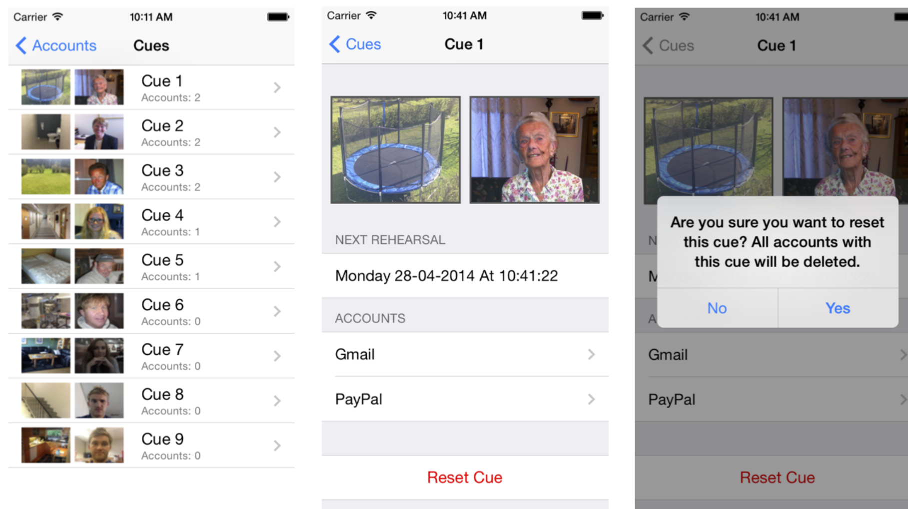


Fig. 6: Cue information

The main objective with the rehearsal schedule is to ensure that the user does not forget the action and object associated with each of the cues. This is done by

notifying the user to rehearse the cue-association according to specific intervals. Figure 7 shows how PassCue notifies the user when rehearsal is required. The first screen shows that the application icon reflects a notification, and the next screen shows the notification in the notification center. If the user has the application open, or opens the application after a notification has been fired, the message is displayed in the application. In this example, the user must practice the cue-association for cue 1 in order to not forget the association.

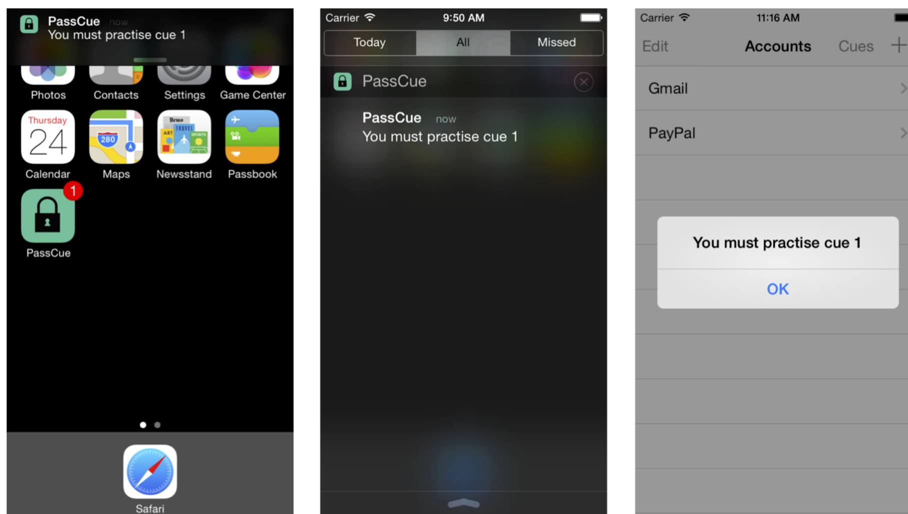


Fig. 7: Rehearsal notification

The user rehearses the cue-association by logging in to one of the accounts that use the cue. The cue overview, shown in Figure 6, can be used to help the user see which account he must log in to for rehearse. In this example, both *Gmail* and *PayPal* are using cue 1, so the user can choose which account to log in to. If the user logs in to *Gmail* the rehearse schedule for not only cue 1, but cue 2, 3 and 4 is updated.

Time Memory Tradeoff Analysis of Graphs in Password Hashing Constructions

Donghoon Chang, Arpan Jati, Sweta Mishra, Somitra Kumar Sanadhya

Indraprashtha Institute of Information Technology, Delhi (IIIT-Delhi), India
{donghoon,arpanj,swetam,somitra}@iiitd.ac.in

Abstract. One of the most important tradeoff for a password-hashing scheme is the Time-Memory Tradeoff (TMTO) and any such scheme needs to have a strong TMTO defense. A strong TMTO defense would mean that the attacker would not be able to reduce memory consumption at the cost of increased runtime. GPUs, FPGAs and custom ASICs provide tremendous amounts of computational power. A good password-hashing scheme must not allow the attacker to compute a password faster than possible on general purpose CPUs. In this work, we provide a technique to analyze algorithms which can be represented as a DAG (Directed Acyclic Graph) to estimate the expected run-times at varied levels of available memory. We provide a deterministic algorithm to traverse the DAG and obtain the real-world execution times. Even though our technique is applicable to other algorithms, we consider the two designs Catena and Rig for our analysis.

Keywords: Time-Memory tradeoff, password, hashing, graph traversal, bit-reversal graph, double butterfly graph

1 Introduction

'Password Hashing' is a technique for performing a one way transform on a password to convert it to a fixed length array of bytes. The password-hashing operation has the following major requirements:

- **One-wayness:** It should not be possible to obtain the password given the resulting hash.
- **Memory Hardness:** It should take a fixed amount of memory (RAM) to perform the hashing operation. If less memory is available than predetermined, the operation should take significantly longer time (preferably exponential).
- **Constant Time Operation:** The algorithm should take a fixed amount of time irrespective of the given input for a fixed set of parameters.

The *Memory Hardness* requirement is to make sure that the user has at-least a certain amount of memory in order to perform the hashing operation. This is to prevent attackers using GPU clusters, FPGAs and ASICs from brute-forcing a *password* from a given hash. A general purpose cryptographic hash function like SHA-2, BLAKE etc. are too fast in hardware or software implementations. An attacker can perform billions of tests per second for a given *password hash*, and, as a result recover the original *password* with ease.

The Password Hashing Competition [1] has several candidates which claim memory-hardness, but, there is no easy way to test memory hardness directly from an algorithmic description. In this paper, we provide a technique to analyze algorithms which can be represented as a DAG (Directed Acyclic Graph). We also provide an algorithm to traverse the DAG with low memory, and compute the re-computation penalties for different tradeoff options. The technique is applied to Catena [5] and Rig [3] to analyze them and obtain TMTO (see below) values for various combinations of options.

2 Preliminaries

2.1 Time-Memory Tradeoff

Hellman in [6] introduced the idea of Time-Memory Tradeoff (TMTO) wherein the attacker tries to optimize the product $T \cdot M$ for a task where T is the number of operations performed (time) and M is the number of words of memory. The relative cost of CPU cycles is much lesser than RAM space, as a result most attacks attempt to reduce memory at the cost of increased algorithmic runtime.

2.2 Directed Acyclic Graph (DAG)

A directed graph is an ordered pair $(\mathcal{V}, \mathcal{E})$ such that $\mathcal{V}$ is a set of nodes and $\mathcal{E}$ are a set of edges. Every edge $e = (X_i, X_j)$ in the set $\mathcal{E}$ is ordered and $\{X_i, X_j\} \subseteq \mathcal{V}$. A directed graph $\mathcal{G}$ is acyclic if it does not contain any directed cycles.

2.3 Bit-Reversal Permutation

A **bit reversal permutation** is a permutation of a sequence of m elements with $m = 2^k$. The elements are indexed from 0 to $m-1$ and the bits are reversed in the binary representation. Each element is then mapped to the new location as per the reversed value. Example, for $k = 3$, $m = 2^3$ and indices are $0, 1, \cdots, 7$:

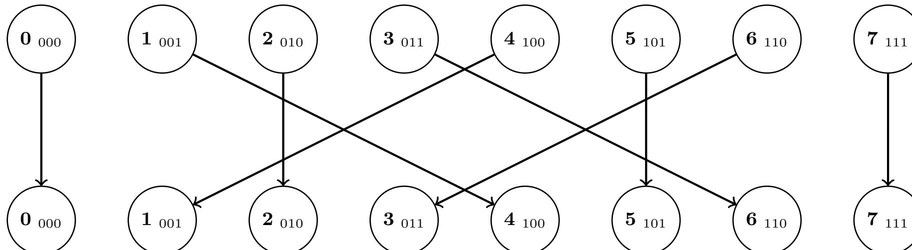


Fig. 1: Bit-Reversal Permutation

3 Graphs

In this paper, we analyze DAGs consisting of a two dimensional matrix of nodes. The connectivity (dependency) between the nodes determines the overall characteristics of the graph.

To define the various graphs we use the following 4 types of edges as shown in Figure 2. All the graphs analyzed will be the result of overlaying these graph types in various combinations.

The type *BitReversed* is obtained by applying bit-reversal permutation as shown in Figure 1 to all the layers in the graph, and the type *Butterfly* [4, 2] is obtained by placing two back-to-back Fast Fourier Transformation (FFT) graphs after omitting one row in the middle. The double-butterfly graph being described here was originally defined in [5].

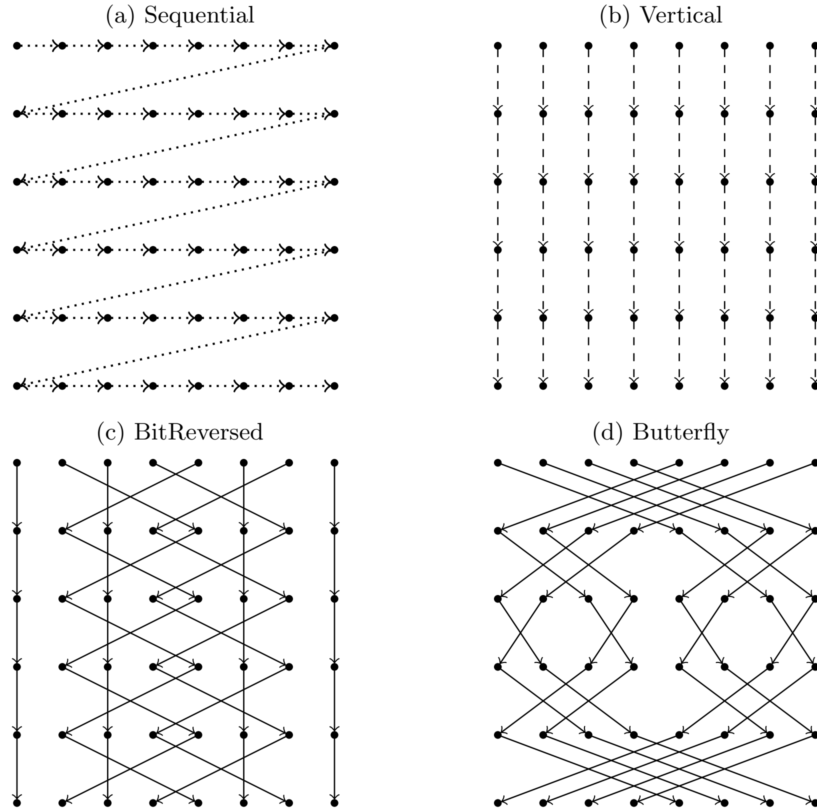


Fig. 2: Types of edges

3.1 $(\mathcal{N}, \lambda)$-Straight Graph

A $(\mathcal{N}, \lambda)$-Straight Graph with $\mathcal{V}$ vertices and $\mathcal{E}$ edges can be formed by overlaying the *Sequential* and *Vertical* edge types. $\mathcal{N} = 2^k$ where, $k \in \mathbb{N}$ and $k \geq 1$.

It is a simple and symmetric graph, without any permutations. We use it to show the working of the DAG traversal algorithm (defined below) with respect to the other designs Catena and Rig.

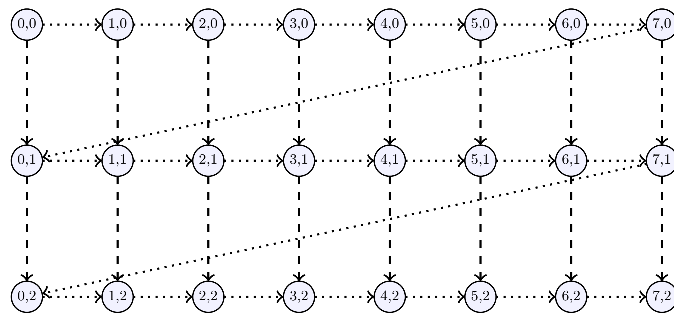


Fig. 3: (8,2)-Straight Graph

3.2 $(\mathcal{N}, \lambda)$-Bit-Reversal Graph

A $(\mathcal{N}, \lambda)$-Bit-Reversal Graph with $\mathcal{V}$ vertices and $\mathcal{E}$ edges can be formed by overlaying the *Sequential* and *BitReversed* edge types. $\mathcal{N} = 2^k$ where, $k \in \mathbb{N}$ and $k \geq 1$; definition adapted from [5].

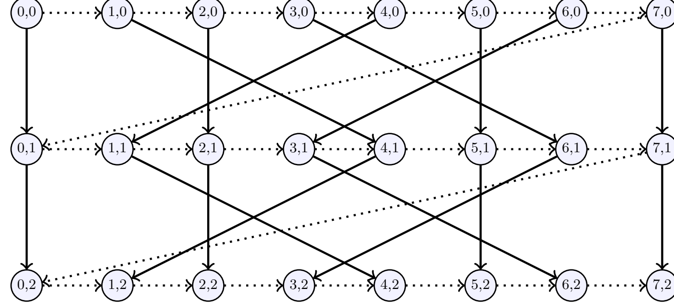


Fig. 4: (8,2)-Bit-Reversal Graph

As per the definition, it performs bit-reversal permutation at each level. Or, it is a stack of λ bit-reversal permutation operations.

3.3 $(\mathcal{N}, \lambda)$-Bit-Reversal-Straight Graph

A $(\mathcal{N}, \lambda)$-Bit-Reversal-Straight Graph with $\mathcal{V}$ vertices and $\mathcal{E}$ edges can be formed by overlaying the *Sequential*, *Vertical* and *BitReversed* edge types. $\mathcal{N} = 2^k$ where, $k \in \mathbb{N}$ and $k \geq 1$; definition adapted from [3].

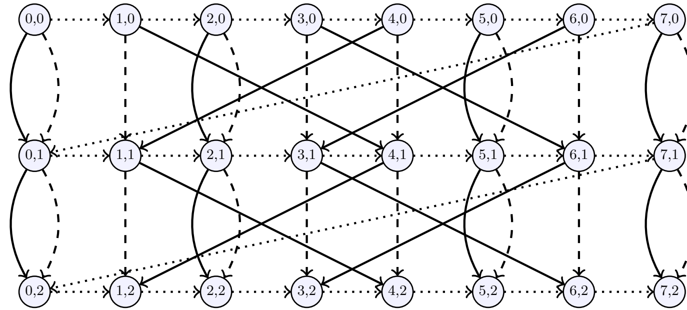


Fig. 5: (8,2)-Bit-Reversal-Straight Graph

This graph describes the Rig construction, considering, the attacker stores all the values at the ith location for both the memory arrays at the same time. A more efficient attack might use some other strategy where the two arrays are handled separately, but, for the sake of analysis we consider this graph.

3.4 $(\mathcal{N}, \lambda)$-Double-Butterfly Graph

A $(\mathcal{N}, \lambda)$-Double-Butterfly Graph with $\mathcal{V}$ vertices and $\mathcal{E}$ edges can be formed by overlaying the *Sequential*, *Vertical* and *Butterfly* edge types. $\mathcal{N} = 2^k$ where, $k \in \mathbb{N}$ and $k \geq 1$; definition adapted from [5].

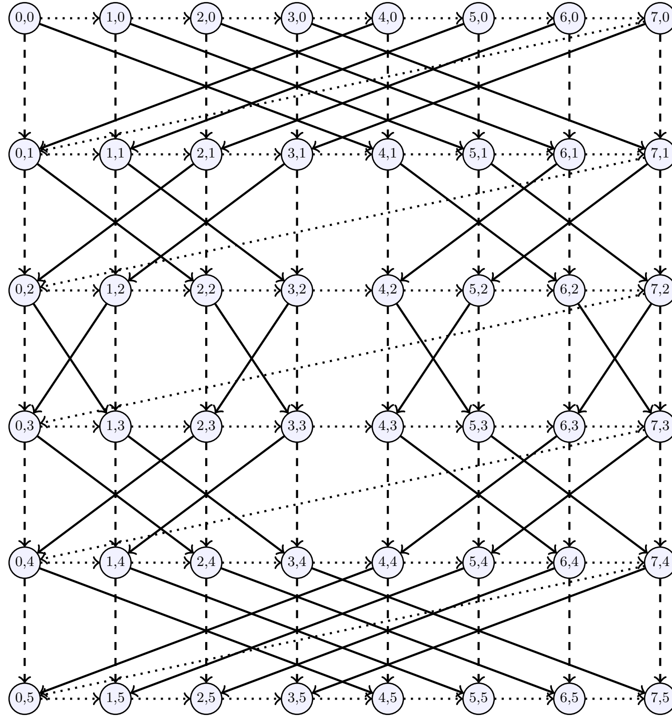


Fig. 6: $(8, 1)$-Double-Butterfly Graph

As per the description of Catena, this operation is stacked λ times. The original traversal pattern comes from the original FFT butterfly structure. Due to the significantly large number of operations and large number of layers this Graph traversal is significantly slower than the previous types, but, it has 3 input dependencies per node and the re-computation effort is exponential.

4 Traversing a Dependency Graph

A DAG representing the memory dependencies in a password hashing algorithm can be analyzed in many ways, for the purpose of Time-Memory Tradeoff analysis we have devised an algorithm (Algorithm 2) for traversing the nodes efficiently in order to obtain the tradeoff penalties for various combinations of memory configurations.

As the algorithm traverses nodes we define a data structure to keep the state of the nodes during traversal.

```
structure Node
{
    integer X = 0, Y = 0;
    boolean MemoryAllowed = false, MemoryValid = false, Traversed = false;
    array Node [ ] Dependencies;
}
```

The password hashing algorithms we are analyzing need memory equal to the number of nodes in one row. If the attacker has enough memory, then there is no need to do any TMTO, and the runtime will equal the time it takes to process all the nodes once. If the attacker cannot

store all the nodes, then he will need to perform significantly large number of operations to calculate the node values.

As defined above, the structure *Node* has the field *MemoryAllowed* to let the algorithm know which node has memory and allow it to store the value when it is available/calculated during traversal.

Algorithm 2: DAG Traverse

Input: **graph⟨Node⟩** {Dependency Graph to Traverse}, **integer** M {Columns}, **integer** N {Rows}
Variables: Node n, **stack⟨Node⟩** proc, dep, **boolean** depfound, **list⟨Node⟩** traverse
Output: **list⟨Node⟩** {A list of nodes traversed by the algorithm.}

```
1.  nd = graph[M-1, N-1];
2.  while(true) do
3.      if(n.Traversed == false)
4.          foreach dep in n.Dependencies do
5.              if dp.MemoryValid = false
6.                  dep.push(dp)
7.              end if
8.          end foreach
9.          n.Traversed ← true
10.     else
11.         if dep.count > 0
12.             n = dep.pop()
13.             proc.push(n)
14.             if n.MemoryValid = false
15.                 traverse.add(n)
16.             end if
17.             depfound ← true;
18.             foreach Node d in n.Dependencies do
19.                 if d.MemoryValid = false
20.                     depfound ← false
21.                 end if
22.             end foreach
23.             if depfound = true
24.                 while proc.count > 0 do
25.                 temp = proc.pop()
26.                     if temp.MemoryAllowed = true
27.                         temp.MemoryValid ← true
28.                     end if
29.                 end while
30.                 graph.clearAllTraversed()
31.             end if
32.         else break
33.         end if
34.     end if
35. return traverse
```

Algorithm 2 (as defined) is considering *pointer arithmetic*, so, when *nd* is pushed and then popped from the stack *dep*, any changes to *nd* will be reflected in the initial *graph* structure and all the associated lists are stacks.

The algorithm starts from the output node and runs iteratively, traversing node-to-node until all the dependencies are fulfilled. There can be a large number of possible combinations of allowed-memory locations and the overall effort (computations) will depend on the allowed memory and its allocation in the complete graph. The algorithm can be better understood with the help of an example. Figure 7 shows a $(4, 2)$-Bit-Reversal Graph.

Let us consider that all the nodes allow memory storage. The algorithm traverses the nodes in the following order for the graph in Figure 7. The procedure can be performed by traversing the nodes in the following order:

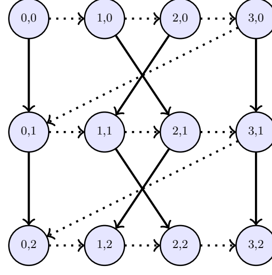


Fig. 7: $(4, 2)$-Bit-Reversal Graph

(3,2) → (3,1) → (3,0) → (2,0) → (1,0) → (0,0) →
(2,1) → (1,1) → (0,1) → (2,2) → (1,2) → (0,2)

To explain the traversal we will follow the following notation:

$$(N_x^i, N_y^i) \rightarrow \{\ (D_x^j, D_y^j), (D_x^{j+1}, D_y^{j+2}), \ldots \} \Rightarrow (N_x^{i+1}, N_y^{i+1}) \rightarrow \ldots$$

Where, (N_x^i, N_y^i), $((N_x^{i+1}, N_y^{i+1}))$ ... are the nodes and $(D_x^j, D_y^j), (D_x^{j+1}, D_y^{j+2})$, ... are the dependencies discovered during the traversal. So, according to this notation the traversal for a $(4, 2)$-Bit-Reversal Graph having 12 steps is described by the following path:

(3,2) → {(2,2),(3,1)} ⇒ (3,1) → {(2,1),(3,0)} ⇒ (3,0)→ {(2,0)}⇒
(2,0) → {(1,0)},(1,0) → {(0,0)}⇒(0,0) → { }⇒(2,1) → {(1,1)} ⇒
(1,1) → {(0,1)}⇒(0,1) → { }⇒(2,2) → {(1,2)}⇒(1,2) → {(0,2)} ⇒ (0,2) → { }

The aim is to find out the value of node (M-1,N-1) where M is the number of columns and N is the number of rows, while only the input (0,0) is known. All the nodes in between need to be calculated along the way. The simplified description of the algorithm is as follows.

- Set the node to be found.
- Find all the dependencies of the provided node.
- Continue finding dependencies until a node is found with no dependencies while adding the intermediate nodes to an array (initial value at (0,0) or node with *MemoryValid* set to true).
- Process the array and update all values and memory; set the *MemoryValid* flag for the memories as valid if the *MemoryAllowed* flag is set.
- During the previous step all the newly found dependencies are added to an array.
- Visit all nodes in the array which are found to be dependencies and continue until no more nodes are in the stack with dependencies.

Considering the node (3,2) as the starting point in Figure 7. The dependencies (2,2) and (3,1) are found; for (3,1) the dependency (2,1) and (3,0) is found; for (3,0) the dependency (2,0) is found; for (2,0) the dependency (1,0) is found; for (1,0) the dependency (0,0) is found. Whenever a dependency is found it is put in the stack. As the dependency (0,0) is known, all the nodes can be processed in the current stack, as a result the value and memory in (1,0), (2,0) and (3,0) is updated and now those dependencies can be readily met. Now, we process (2,1) which is already in the stack and it is found that (1,1) was a dependency and we add it to the stack. The stack is then processed again and the process continues until all the nodes are processed and all the dependencies are met. This complete operation is shown in Table 1.

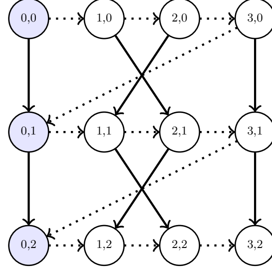


Fig. 8: $(4, 2)$-Bit-Reversal Graph with only memory in the first column.

Lets consider Figure 8, here, only the first column is allowed memory storage. The rest of the nodes have no memory and during traversal (*MemoryValid* set to **false**), they need to be re-calculated every time they are encountered.

The complete traversal for a $(4, 2)$-Bit-Reversal Graph as shown in Figure 8 has 35 steps, it is shown as follows:

$$\begin{gathered}
(3{,}2) \rightarrow \{(2{,}2),\ (3{,}1)\} \Rightarrow (3{,}1) \rightarrow \{(2{,}1),\ (3{,}0)\} \Rightarrow (3{,}0) \rightarrow \{(2{,}0)\} \Rightarrow \\
(2{,}0) \rightarrow \{(1{,}0)\} \Rightarrow (1{,}0) \rightarrow \{(0{,}0)\} \Rightarrow (0{,}0) \rightarrow \{\} \Rightarrow (2{,}1) \rightarrow \{(1{,}1),\ (1{,}0)\} \Rightarrow \\
(1{,}0) \rightarrow \{\} \Rightarrow (1{,}1) \rightarrow \{(0{,}1),\ (2{,}0)\} \Rightarrow (2{,}0) \rightarrow \{(1{,}0)\} \Rightarrow (1{,}0) \rightarrow \{\} \Rightarrow \\
(0{,}1) \rightarrow \{(3{,}0)\} \Rightarrow (3{,}0) \rightarrow \{(2{,}0)\} \Rightarrow (2{,}0) \rightarrow \{(1{,}0)\} \Rightarrow (1{,}0) \rightarrow \{\} \Rightarrow \\
(2{,}2) \rightarrow \{(1{,}2),\ (1{,}1)\} \Rightarrow (1{,}1) \rightarrow \{(2{,}0)\} \Rightarrow (2{,}0) \rightarrow \{(1{,}0)\} \Rightarrow (1{,}0) \rightarrow \{\} \Rightarrow \\
(1{,}2) \rightarrow \{(0{,}2),\ (2{,}1)\} \Rightarrow (2{,}1) \rightarrow \{(1{,}1),\ (1{,}0)\ \} \Rightarrow (1{,}0) \rightarrow \{\} \Rightarrow (1{,}1) \rightarrow \{(2{,}0)\} \Rightarrow \\
(2{,}0) \rightarrow \{(1{,}0)\ \} \Rightarrow (1{,}0) \rightarrow \{\} \Rightarrow (0{,}2) \rightarrow \{(3{,}1)\ \} \Rightarrow (3{,}1) \rightarrow \{(2{,}1),\ (3{,}0)\ \} \Rightarrow \\
(3{,}0) \rightarrow \{(2{,}0)\ \} \Rightarrow (2{,}0) \rightarrow \{(1{,}0),\ \} \Rightarrow (1{,}0) \rightarrow \{\} \Rightarrow (2{,}1) \rightarrow \{(1{,}1),\ (1{,}0)\} \Rightarrow \\
(1{,}0) \rightarrow \{\} \Rightarrow (1{,}1) \rightarrow \{(2{,}0)\} \Rightarrow (2{,}0) \rightarrow \{(1{,}0)\ \} \Rightarrow (1{,}0) \rightarrow \{\}
\end{gathered}$$

Table 1: Traversal for Figure 7

Position	Dependency
(3,2)	(2,2), (3,1)
(3,1)	(2,1), (3,0)
(3,0)	(2,0)
(2,0)	(1,0)
(1,0)	(0,0)
(0,0)	
(2,1)	(1,1)
(1,1)	(0,1)
(0,1)	
(2,2)	(1,2)
(1,2)	(0,2)
(0,2)	

5 Results

The DAG traversal algorithm given in the previous section can be used to calculate the traversal penalties for any graph. We apply the said algorithm to the following cases and come up with re-computation penalties in different scenarios.

As a large number of memory configurations are possible, in this work we allow a limited set of configurations, i.e. only columns can be enabled or disabled. As a result when a column is enabled all the nodes for that column will have memory storage abilities and vice versa. For analyzing reduced memory scenarios, we take the fraction of memory concerned and evenly distribute the memory along columns starting from the first column.

- $(\mathcal{N}, \lambda)$-Straight Graph (SG)
- $(\mathcal{N}, \lambda)$-Bit-Reversal Graph (Catena BRG)
- $(\mathcal{N}, \lambda)$-Double Butterfly Graph (Catena DBG)
- $(\mathcal{N}, \lambda)$-Bit-Reversal-Straight Graph (Rig Graph)

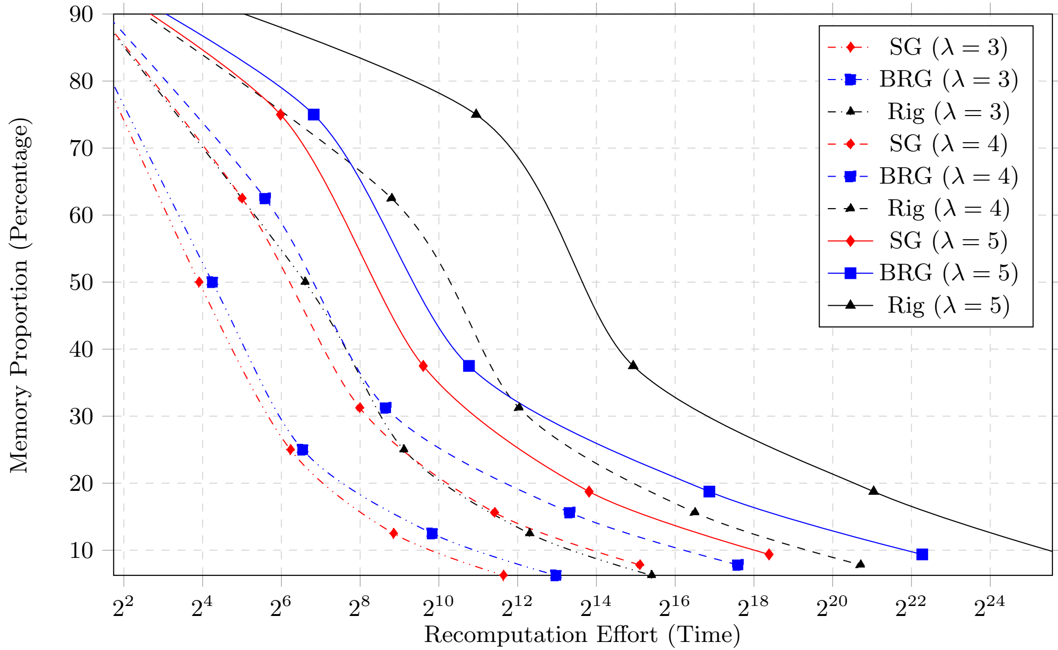


Fig. 9: Re-computation Penalties for Graphs

The cumulated results for the graphs are as shown in Figure 9. The results show the variation of the re-computation effort with change in allowed memory proportion. It is clearly visible that the re-computation effort needed increases drastically for reductions in memory size.

Table 2: Re-computation Penalties for Graphs with 4 rows ($\lambda = 3$)

Memory Proportion (%)	Straight Graph	Bit-Reversal Graph	Bit-Reversal-Straight Graph
50	15	19	97
25	75	93	551
12.5	460	909	5036
6.25	3181	8019	43143

We have taken M=64 (columns) for these experiments. We also tried with larger values like 256 and greater, but, the results are comparable as the characteristics for the DAGs in question do not depend significantly on the value of M, but, the runtime becomes large. From the graphs and tables we could conclude that Catena-DBG and Rig are exponential in nature, whereas Catena-BRG and Straight-Graph is not.

Table 3: Re-computation Penalties for Graphs with 5 rows ($\lambda = 4$)

Memory Proportion (%)	Straight Graph	Bit-Reversal Graph	Bit-Reversal-Straight Graph
62.5	32	48	445
31.25	254	401	4190
15.625	2724	10215	92483
7.8125	35120	197389	1707950

Table 4: Re-computation Penalties for Graphs with 6 rows ($\lambda = 5$)

Memory Proportion (%)	Straight Graph	Bit-Reversal Graph	Bit-Reversal-Straight Graph
75	63	113	1971
37.5	777	1736	31270
18.75	14378	119358	2152596
9.375	341447	5060331	-

Table 5: Relative Re-computation Penalties for Double-Butterfly Graph ($\lambda = 1$)

Memory Proportion (%)	Double-Butterfly Graph
50	18
25	922
12.50	60504
6.25	2043702

The algorithm uses stacks during calculations, but, the maximum size of the stack is bounded by the maximal length of a single path (plus sub paths). As a result, the algorithm does not consume large amount of memory even for huge re-computation dependency tree calculations.

6 Conclusion and Future Work

We have provided a technique for traversal of DAGs which can be used on various algorithms to analyze the Time-Memory Tradeoff. We then applied it on two designs from the Password Hashing Competition [1] and performed preliminary analysis with various parameters and TMTO options. The technique is flexible, and can be used in analysis of several cryptographic designs (some with minor simplification). It is possible to adapt the provided DAG traversal algorithm to suit various situations and combinations of operations to help us analyze complex cryptographic designs for which making a mathematical model is significantly difficult.

References

1. Password Hashing Competition (PHC), 2014. `https://password-hashing.net/index.html`.
2. William F Bradley. Superconcentration on a pair of butterflies. *arXiv preprint arXiv:1401.7263*, 2014.
3. Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Rig: A simple, secure and flexible design for password hashing. 2014.
4. James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.
5. Christian Forler, Stefan Lucks, and Jakob Wenzel. The Catena Password-Scrambling Framework. Submission to Password Hashing Competition (PHC), 2014. https://password-hashing.net/submissions/specs/Catena-v2.pdf, 2014.
6. Martin E Hellman. A cryptanalytic time-memory trade-off. *Information Theory, IEEE Transactions on*, 26(4):401–406, 1980.