# The Password Hashing Competition

Peter Gutmann

University of Auckland

## Prehistory of Password Protection

Before timesharing

- Whoever submitted the card deck owned it

## Prehistory of Password Protection (ctd)

Compatible Time-Sharing System (CTSS), 1963

- Introduced the use of a “private code” to protect access to users’ data

## Prehistory of Password Protection (ctd)

Famously failed in 1966

- CTSS editor used a fixed temporary filename
- Admin edited the password file and login message file at the same time…

Problem occurred at 5pm on a Friday

- User noticed it and deliberately executed an HCF instruction in the debugger
- When machine was rebooted, users were told to change their passwords
  - (And given free credit monitoring)

## History of Password Protection

Cambridge Uni Titan timesharing system, 1967, used a one-way cipher to protect the password

Spread to CTSS' successor Multics in the 1970s

- And from there to a Multics successor, Unics^H^Hx

## History of Password Protection (ctd)

Unix originally stored passwords in the clear

- More problems with editor temp files

Encrypt the passwords like Multics had done

- Protect against brute-force by iterating the encryption
- Protect against comparing encrypted passwords by adding a random quantity (salt) to the password

Originally based on a software analogue of the M-209 cipher machine

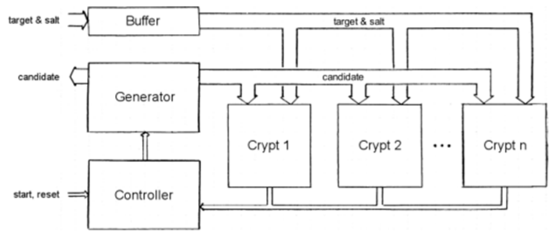
- Encrypt the password using itself as the key
- Found to be too fast, vulnerable to brute-forcing

## History of Password Protection (ctd)

Later Unix `crypt` used 25 iterations of DES encryption

- Salt+password used as a key to encrypt a block of zeroes
- DES was tweaked slightly via the salt to make it impossible to use standard DES hardware for brute-forcing
  - Anticipated ASIC brute-force problem by about 30 years

Experimental FPGA-based `crypt` breaker was created in 2006 by Michael Poppitz



## History of Password Protection (ctd)

Many, many vendors reinvented this system…

… badly

## History of Password Protection (ctd)

DEC Ultrix `crypt16`

- Kludge-and-paste of Unix `crypt`
- Broke the password into two 8-byte halves
- Encrypted the first half with 20 iterations of DES, the second half with 5 iterations
  - The number 25 was sacred to the Aztecs?

Break the password suffix using 5 iterations of DES

- Use the suffix to find the prefix

## History of Password Protection (ctd)

Microsoft LMHASH and NTHASH

- Contrary to popular belief, were not deliberately designed as an enumeration of every mistake you could make in password handling

WiFi Alliance

- Break a 7-digit PIN into a 4-digit and 3-digit part and authenticate them separately (`crypt16` memorial error)

## History of Password Protection (ctd)

Poul-Henning Kamp created `md5crypt` in 1995 to create a more modern replacement for crypt

- 1000 iterations of MD5 over the previous hash concatenated alternately with either the password or the salt

By about 2010 `md5crypt` was as insecure against attack as `crypt` had been in 1995

## History of Password Protection (ctd)

Algorithm was tweaked by different distros…

Use SHA-2 instead of MD5

- Doesn’t help much, since we don’t care about collision-resistance

Use more iterations

- Helps a bit more

## History of Password Protection (ctd)

Another attempt was `bcrypt` in 1999

- Leverage's Blowfish's notoriously slow key setup
  - – Derives a large subkey state array from the main key
- Deliberately make it even slower
- Technical term is "memory-hard function"

Eksblowfish, Expensive Key Schedule Blowfish

- Iteratively use Blowfish to encrypt the main key and salt and replace the subkeys with the new data

Used in BSDs and some Linux distros

## Cryptographic Password Protection

PBKDF2, Password-Based Key Derivation Function #2, 1999

- Created by cryptographers to have/take advantage of specific crypto properties

Replaced the earlier PKCS #5 ("PBKDF1"), 1993

- Built around MD2, MD5, and DES keys
- Hash password and salt with MD5
- Iterate by hashing the MD5 output

## Cryptographic Password Protection (ctd)

PBKDF2 mixes the password, salt, and a block count into each iteration of hashing

- Built around HMAC, keyed hash function, rather than raw hash
  - HMAC is keyed using the password
- Parameterised, any hash function can be used
- First round uses an explicit salt
  - Subsequent rounds use the output of the previous round as the salt
- Each iteration's output is XORed together

## Cryptographic Password Protection (ctd)

Fixes several PBKDF1 problems…

HMAC has better security properties than the underlying hash function

- HMAC-MD5 is still secure even though MD5 has been broken

Password is applied to each round

Output depends on each individual round's output

- Is there an operation SHA1X that performs the equivalent of *n* SHA1 iterations in a single step?
- Was a concern for 3DES, if DES is a group then we can collapse 3DES to something equivalent to single DES

## In the Meantime…

CPUs got faster

- Much, much faster

GPUs allowed parallelisation of password-cracking

- GPGPUs finally lifted the limits of strictly-graphics GPUs
- Try doing crypto with saturating adds…

Mining ASICs appeared

- Original BitCoin ASICs weren't terribly suited to password cracking
  - *Application-Specific* IC
- Still, someone really dedicated and with lots of funding could repurpose one

## In the Meantime… (ctd)

One attempt to deal with this was `scrypt`, 2012

- Create a memory-hard function to mess up GPUs, FPGAs, and ASICS

Build a large array of pseudorandom strings using HMAC-SHA256

- Access them in a pseudorandom manner

`scrypt` design allows time/memory tradeoff, TMTO

- Lots of memory usage, fast operation
- Low memory usage using on-the-fly recalculations, slow operation

## In the Meantime… (ctd)

Efficient on general-purpose CPUs

- Lots of memory, (relatively) high bandwidth

Inefficient on earlier GPUs, ASICS

- Not much memory and/or…
- Low bandwidth and/or…
- Architecture unsuited to implementing `scrypt`

Later GPGPUs resolved this problem

## In the Meantime… (ctd)

Since `scrypt` was a good memory-hard function, it was adopted by cryptocurrencies like Litecoin and Dogecoin

Mining ASICs followed…

- You can buy them on eBay
- Measured in tens of MH/s rather than GH/s, but still…

## Fixing the Problem

How do we fix this once and for all?

- Bring it to the attention of the crypto community

How do we do that?

- Run a competition…

This way of doing things has an established history…

- AES competition to select a replacement for DES/3DES
- SHA-3 competition to select a replacement for SHA-1/SHA-2

## The AES Competition

Run by NIST from 1997 to 2000

Requirements: 128-bit block cipher, 128/192/256-bit key size

- Most block ciphers at the time were 64-bit

Spawned a huge research effort to find the best algorithm

- Some algorithms fell to cryptanalysis
- Some didn't perform so well
- Some didn't work well in certain environments like smart cards

Greatly advanced the state of the art in block cipher design

- Winner was Rijndael, selected as the AES in 2000

## The SHA-3 Competition

Reuse of the AES methodology

- SHA-1 was known to have issues
- SHA-2 was based heavily on SHA-1
- There were concerns that SHA-1 attacks could be extended to SHA-2

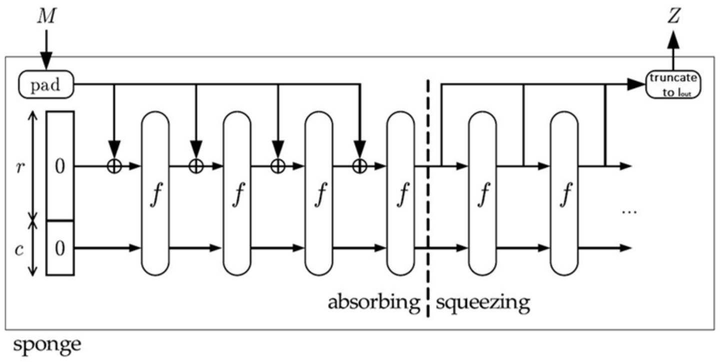
Run by NIST from 2007 to 2012

- Requirements: 224/256/384/512-bit block size
  - NIST has some odd requirements that no-one else much cares about, 256 and 512 would have been fine
  - c.f. AES 192-bit keys

As for AES, lots of competitive cryptanalysis

## The SHA-3 Competition (ctd)

Winner was Keccak, selected as SHA-3 in 2012



- (You are not expected to understand this)

## SHA-3 Problems

NIST chose unnecessarily conservative parameters

- Keccak had to be tweaked during the evaluation process to meet these requirements, making it much slower

NIST then modified Keccak after it had won to make it faster again

- Details are complex and based on its sponge-function design
- "Sponge" absorbs input bits which are then squeezed out to produce output bits
- Can modify the sponge capacity to make the overall hash faster

## SHA-3 Problems (ctd)

SHA-3 wasn't going to be the Keccak that won the competition

- Other algorithms might have had a better chance based on the final criteria

After lots of uproar, the original Keccak design was used as SHA-3

The result hasn't taken the world by storm

- SHA-3 is S-L-O-W

## SHA-3 Competition Redux

As a result of the SHA-3 work, we now know much more about SHA-2

- It’s actually pretty good

SHA-2 does the same thing that SHA-3 does, but much more efficiently

- SHA-2 is already widely deployed

NIST: “SHA3 should *complement* SHA2”

- Why have SHA-3 then?
- An algorithm that complements the incumbent, but with worse performance

## SHA-3 Competition Redux (ctd)

Little interest in SHA-3

- Keep using SHA-2

Use alternatives like BLAKE2, derived from SHA-3 competitor BLAKE

- As secure as SHA-3 but faster than SHA-3, SHA-2, SHA-1, and MD5
- (Beating MD5 for speed is quite a feat)

## SHA-3 Competition Redux (ctd)

Replace MD5 with SHA-3! New! Improved! Really slow!

- ZOMGWTF??!?
- Since when is “really slow” → “improved”?

Replace MD5 with BLAKE2! New! Improved! Quicker!

- Faster, better
- The blender for the next millennium

## SHA-3 Competition Redux (ctd)

Lessons learned

- Set realistic goals
- Don’t move the goalposts during the game
- Winner has to actually be better than the incumbent method

## The Password Hashing Competition

Spur new research as the AES and SHA-3 competitions did

- Select a best-of-breed password-protection mechanism

Run by cryptographer Jean-Philippe Aumasson and others, 2013 to 2015

- Fully open, public version of what NIST did

## The Password Hashing Competition (ctd)

Panel members included

- Cryptographers
- Security practitioners
- Password breakers
  - e.g. a member of the Hashcat project
- People who had previously worked in this field
  - e.g. the creator of `scrypt`
- Some hippy from NZ
- Many others

## The Password Hashing Competition (ctd)

The winner had better be pretty damn good

- It has the potential to be used anywhere and everywhere

Just PBKDF2 alone is used in

- Cisco IOS
- Disk encryption software (lots)
- iOS
- OS X
- Password-based crypto (e.g. WinZip, ODF, password managers)
- S/MIME / CMS apps
- WiFi (WPA/WPA2)
- Windows (DPAPI)

## The Password Hashing Competition (ctd)

Competition attracted 24 submissions, and spawned a lot of new research

- The most significant advance in the state of the art since Unix `crypt`

There were a wide range of entrants…

- How do we choose a winner?

Disclaimer: The following represents my thoughts only, and should not be taken as the opinion of the PHC panel as a whole

- “Opinions are like a**holes, everyone has one”

## Desirable Features

Resistant to attacks from CPUs, GPUs, FPGAs, and ASICs

Oh, and also crypto-assist functions in CPUs

- x86 AES and SHA instructions

All of this is a really tough target to achieve

- Can't just throw lots of memory at it and declare victory
- Needs to work in constrained environments

## Desirable Features (ctd)

Attacks can be parallelised

- Apply e.g. 4 cores to each work on a quarter of the hashing for one password
- Great for SIMD processors like GPUs

Must be resistant to parallelisation

- Expands memory-hardness to require sequential memory-hardness

## Desirable Features (ctd)

Use an established primitive like AES or SHA-2 (or some portion thereof) vs. introduce some new crypto construct

- Need to evaluate a completely new crypto construct alongside the manner in which it's applied

Using/reusing an existing primitive makes the design easier to evaluate

## Desirable Features (ctd)

With current schemes, client and server have to perform the same amount of work

- Fine for a client, but what about a server that has to verify 1,000 connections at once?

Server relief

- The client has to perform the full workload to prove itself
- The server can verify with minimum effort

Simple example: 10,000 iterations of hashing

- Client performs 9,999 rounds
- Server performs 1 round and compares with the stored value

## Desirable Features (ctd)

Deal with improved attacks

- New attack appears
- Update iterations from 10,000 to 50,000 when the user next logs in

About 70% of Facebook and Twitter accounts have an update frequency below once a month

- Can't rely on people logging in to update parameters

Server can update parameters without knowing the password

- Increase the number of rounds from 10,000 to 50,000 without having to do a password re-enrolment
- Helps deal with inactive, easy-target accounts

## Making Sense of the Submissions

Can break the entries down into roughly five categories

- Extension/rehash of `scrypt` and/or PBKDF2
- Established construction widened out for a larger memory footprint
- Iteratively stir a memory block
- Catena-style (see later)
- Exotica
  - "A collection of exotic features with a KDF attached"

## Making Sense of the Submissions (ctd)

Some of the first published work was on a mechanism called Catena

Based on a strong theoretical foundation called the Pebble Game

- Used in theoretical computer science to analyse time/memory tradeoffs (TMTO)

Details are a bit too complex for this talk

- See "The Catena Password-Scrambling Framework", Christian Forler, Stefan Lucks, and Jakob Wenzel
- "Book of Catena", 70+ page password-processing mechanism design guide

## Making Sense of the Submissions (ctd)

This theory-first approach was unique in password-processing function design

Standard approach to this

- Take a hash or crypto primitive
- Mix in the password or salt
- Iterate it
- Analyse the result
- "Seems legit"

Catena work provided a strong theoretical framework through which to analyse new designs

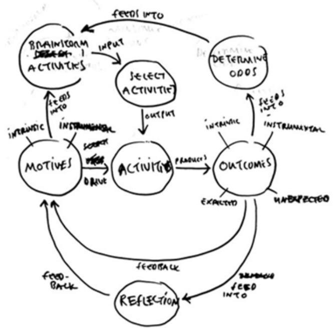
- Other entrants took inspiration from the Catena work

## Initial Selection

So where do we start…

Some entries were poorly-documented, source code-only or a page or two of information

- Difficult to analyse without a strong supporting security argument

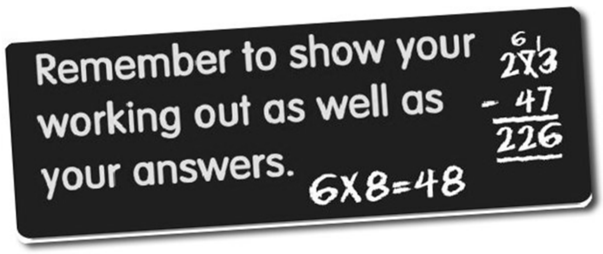


Call for submissions had itemised all the supporting arguments that would need to be provided

## Initial Selection (ctd)

Is behaviour X…

- A design flaw?
- A coding bug?
- A known problem to be worked around?
- A feature?


Remember to show your working out as well as your answers.

$$\begin{array}{r} \overset{6}{2}\overset{1}{\cancel{7}}3 \\ -\ 47 \\ \hline 226 \end{array}$$

$6 \times 8 = 48$

## Initial Selection (ctd)

Some entries used slow, but also architecture-specific, operations to slow things down

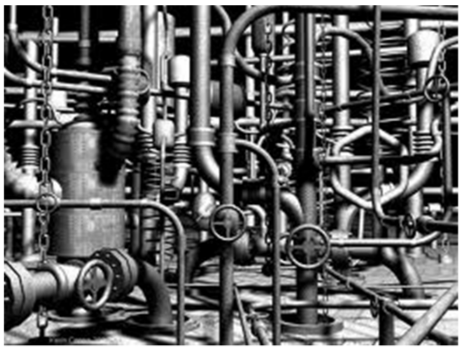
- Pick the slowest FPU ops listed in the hardware reference manual and build on those

Not portable to other systems

## Initial Selection (ctd)

Some entries used nonstandard crypto primitives

- Need to figure out whether the new primitive is secure before you can decide whether the mechanism that it's used in is secure
- Do we want to be analysing entirely new crypto algorithms just to pick a password-processing function?



## Interesting Candidates

Makwa used an interesting approach

- Bignum maths is hard to do efficiently using any currently-known architecture
- Inherently sequential operation, so sequential memory-hardness comes for free
- Iterate the operation as usual to slow down attackers

Server offload is possible by using a trapdoor function

- Think public-key crypto
- Uses modular squaring, $x' = x^2 \bmod n$
- Server knows the private key and can perform the verify operation quickly, client has to do it the hard way

## Interesting Candidates (ctd)

Neat idea, but perhaps a bit too exotic…

Based on a number-theoretic approach

- What if there's a breakthrough in attacking this stuff?

Seems unlikely, but the others are based on laws of physics/engineering

- Typically built from add/rotate/XOR (ARX) constructs
- How many machine instructions can you push down a pipeline in a given time unit?

Analysis based on brute-force instruction assignment is more tractable than analysis based on number-theoretic problems

## First Round

Candidates were selected by the panel based on various criteria

- Defences against attacks from CPU/GPU/FPGA/ASICs
- Resistance to cryptanalytic attacks
- Effectiveness of time/memory tradeoffs
- Clean design
- Quality of the documentation and reference code
- Other factors, e.g. innovative features

All of this is public, see the discussion list and status report

## First Round (ctd)

Finalists mostly shared two common features…

Based on established crypto primitives

- Provide confidence in their security without having to analyse new crypto primitives
- Some of the non-finalists that introduced new primitives were found to be insecure

Detailed security analysis

- Supporting arguments for the design's soundness
- Again, some of the non-finalists that didn't do this were found to have problems

## Finalists

### Argon

- Built on AES primitives
- Thorough security analysis
  - 36 pages for Argon the 1.1 document
  - c.f. standard conference paper size of about 15 pages

### battcrypt

- "Blowfish all the things"
- Iterate over memory with Blowfish
- Relatively brief discussion/analysis

## Finalists (ctd)

### Catena

- See earlier discussion
- 72 pages of analysis
  - "The Book of Catena"

### Lyra2

- Large sponge-based construction
- Also 72 pages of analysis

### Makwa

- See earlier discussion

## Finalists (ctd)

### Parallel

- Same creator as battcrypt
- PBKDF2-like
- Relatively brief discussion/analysis

### POMELO

- (Complex to explain)
- Used its own primitives
- Is the absence of relying on external primitives a feature?
  - Pomelo designer pointed out that being self-contained can be an advantage

## Finalists (ctd)

### Pufferfish

- Evolution of `bcrypt`
- Take an existing, established mechanism and enhance it
- Inspired by work on cracking `bcrypt`
  - These things make `bcrypt` easier to crack
  - Design Pufferfish to eliminate/reduce them
- Very brief security analysis

## Finalists (ctd)

yescrypt

- Evolution of `scrypt`
- Created by someone who really understands password-cracking
- Many, many tunable features
  - Large ROM lookup table, ROMix in `scrypt`
  - `scrypt`-compatibility mode
  - Parallelisation parameters (PWXsimple, PWXgather, PWXrounds, Swidth)
  - Various other parameters
- What am I supposed to evaluate?

## Discussion Points

Obviously, are there any security weaknesses, but then also…

What do we standardise on in terms of parameterisation?

- IPsec/IKE is such a mess because there are too many negotiable options
- Just because SKEME has lots of possible features doesn't mean that you need to use every single one of them

Lots of flexibility looks good on paper, but how do you guarantee interoperability?

- If we allows lots of parameterisation, every standard will choose their own pet values

## Discussion Points (ctd)

Ideally we should have only one knob to adjust operations: easier ↔ harder

Call for submissions specified

One or more cost parameters, to tune time and/or space usage

- The intent was more “one” than “several”
- Certainly not “ten” or “fifteen”

## Discussion Points (ctd)

What about side-channel attacks?

- Timing operations, watching memory access patterns, …

Is this an actual threat?

- Passwords will be hashed on laptops, mobile phones, tablets, …
- If an attacker has compromised those then they don’t need to perform a side-channel attack

If we don’t have to deal with side-channels via constant-time/non-cache-predictable algorithms then we can provide maximum defence against brute-force/cracking attacks

## Discussion Points (ctd)

What if we announce the winner and then someone publishes a paper showing that it's vulnerable to a cache-timing attack?

- Not an *actual* weakness since the design doesn't have to protect against this
- That doesn't mean that the "attack" won't get lots of publicity

Should we compromise the design and defend against an attack that doesn't actually matter?

## Discussion Points (ctd)

Argon2 actually came in two variants for this reason

- Argon2d, data-dependent operations
  - Effective TMTO but subject to some side-channel attacks
- Argon2i, data-independent operations
  - Less effective TMTO but not subject to side-channel attacks

## Discussion Points (ctd)

How easy will it be to fit into existing standards?

- Can it work as a drop-in replacement for (at least) PBKDF2 and `s/bcrypt`?
- Drop-in compatibility was a specific design feature of `yescrypt`

Now that various PHC entries have pointed out new, useful properties for password-processing mechanisms, should we require that the winner have them?

- Server offload
- Client-independent update

## Discussion Points (ctd)

How much tweaking do we allow?

"Competition" was more like a selective-breeding program than a knockout fight

- Try and evolve a best-of-breed solution

Take inspiration from ideas proposed in other submissions

- Not "copy Bob's design" but "that looks like a good idea, maybe we can make our design do that too"

## Discussion Points (ctd)

Very active discussion list provided feedback on designs

- People proposed attacks or questioned design ideas
- Designers responded

Turnaround cycle was often measured in days, not years for conference papers

- FooCrypt'12: Publish new algorithm X
- FooCrypt'13: Publish attack on algorithm X
- FooCrypt'14: Publish amended algorithm X'
- FooCrypt'15: …

## Discussion Points (ctd)

Several entries were modified as the competition ran

Of the finalists

- Argon/Argon2 went to version 3
- Catena went to version 5
- Lyra and POMELO went to version 3
- yescrypt went to version 2

Not necessarily bugs that had to be fixed, but better performance, better features, better analysis

## And the Winner Is…

Argon2 by Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich, University of Luxemburg

Honourable mention for four other finalists

- Catena and Lyra2 for their detailed design and security analysis
- Makwa for its unique features
- `yescrypt` for its feature set and easy upgrade path from `scrypt`

## PHC Lessons Learned

Algorithm competitions, when parameterised appropriately, work

- Collaborative evolution of new crypto mechanisms

Can be run in complete openness

- No need for behind-closed-doors deliberations or government intervention

Dealing with hypothetical but practically irrelevant weaknesses is a problem when the cost to mitigate is significant

- Damned if you do, damned if you don't