



# **Password security: past, present, future (with strong bias towards password hashing)**

**Solar Designer**

**<solar@openwall.com>**

**@solardiz**

**Simon Marechal**

**<bartavelle@openwall.com>**

**@bartavelle**

**<http://www.openwall.com>**

**@Openwall**

**December 2012**

# 1960s to early 1970s: plaintext password storage

---

## Early time-sharing systems

- CTSS

- ▶ "one afternoon [...] any user who logged in found that instead of the usual message-of-the-day typing out on his terminal, he had the entire file of user passwords"

Fernando J. Corbato, "On Building Systems That Will Fail", 1991 (Turing Award Lecture)

(The problem was a text editor temporary file collision, "early 60's" to "1965" by different sources.)

- TENEX had a character-by-character timing leak exacerbated by paging
- "The UNIX system was first implemented with a password file that contained the actual passwords of all the users"

Robert Morris and Ken Thompson, "Password Security: A Case History", 1978

Besides, some typewriters would print the password being typed unless manually prevented from doing so

# Early 1970s - Multics: non-cryptographic hashes

---

Attempted one-way transformations, but no "true" cryptographic hashes yet

- "Multics User Control subsystem stored passwords one-way encrypted [...] I knew people could take square roots, so I squared each password and ANDed with a mask to discard some bits."
- After successful break by the Air Force tiger team doing a security evaluation of Multics in 1972-1974, "we quickly changed the encryption to a new stronger method"

Tom Van Vleck, "How the Air Force cracked Multics Security", 1993 (with later updates)



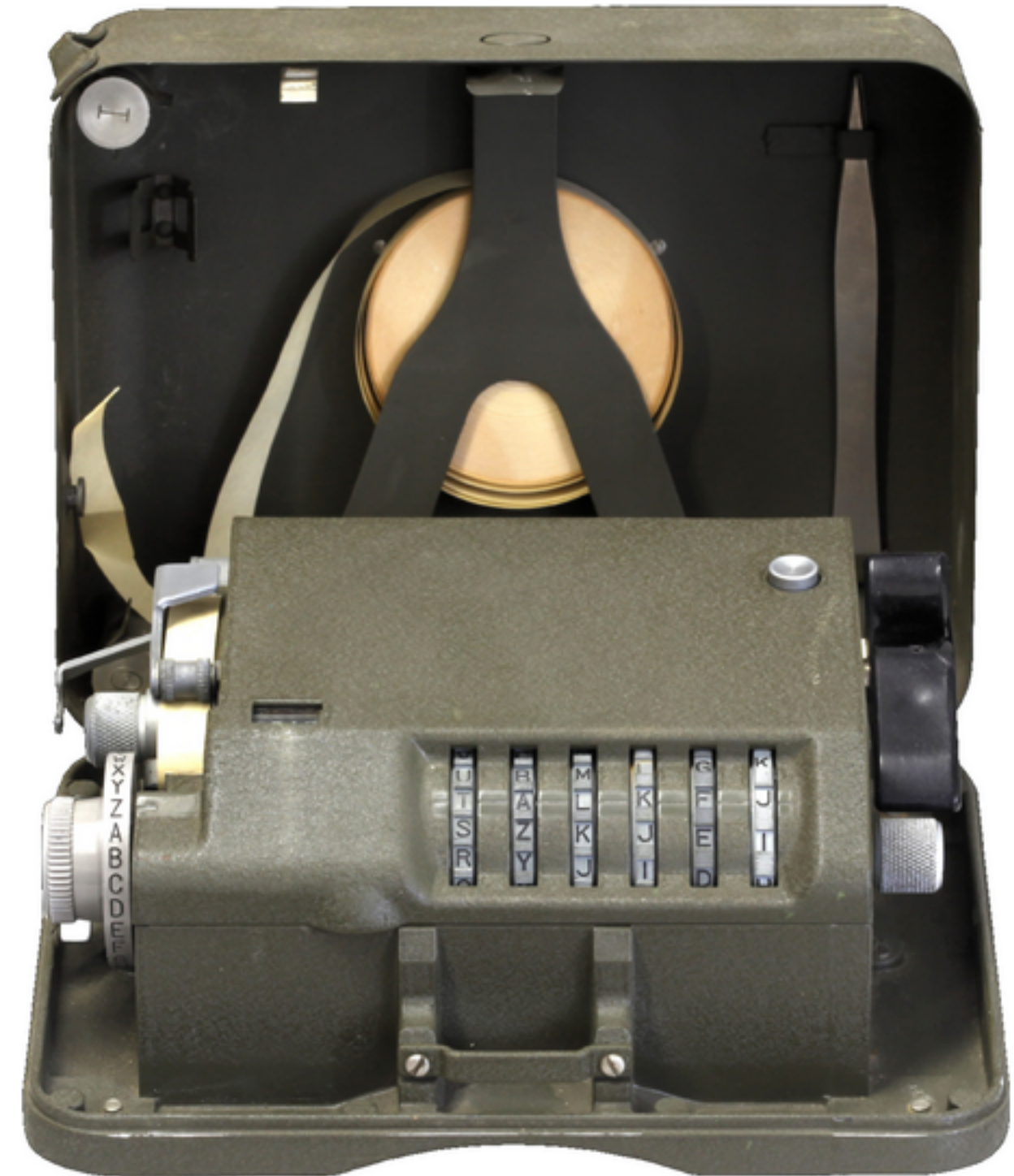
# Early 1970s - Unix: weak cryptographic hash

---

crypt(3) of Unix up to 6th Edition inclusive reused code from an "encryption program [that] simulated the M-209 cipher machine used by the U.S. Army during World War II. [...] the password was used not as the text to be encrypted but as the key, and a constant was encrypted using this key."

M-209B, cryptography collection of the Swiss Army headquarters

Photograph by Rama, Wikimedia Commons, licensed under CeCILL v2 and CC-BY-SA-2.0-FR





# Early 1970s - Unix: too fast and not salted

---

The problems were understood before 7th Edition (1978)

- "The running time to encrypt one trial password and check the result turned out to be approximately 1.25 milliseconds on a PDP-11/70 when the encryption algorithm was recoded for maximum speed."
- "It takes essentially no more time to test the encrypted trial password against all the passwords in an entire password file, or for that matter, against any collection of encrypted passwords, perhaps collected from many installations."

# Late 1970s - Unix: DES-based crypt(3)

---

Unix 7th Edition crypt(3) is a cryptographic one-way hash function built upon the DES block cipher

- "DES is, by design, hard to invert, but equally valuable is the fact that it is extremely slow when implemented in software."
  - ▶ Much faster software implementations of DES were devised later
- "the algorithm is used to encrypt a constant [,which] can be made installation-dependent."
  - ▶ First known mention of local parameterization?
- "Then the DES algorithm is iterated 25 times"
  - ▶ First known use of password stretching?

# Late 1970s - Unix: password strength checking

---

- "The password entry program was modified so as to urge the user to use more obscure passwords."
  - ▶ First known use of automated proactive password strength checking?
- "Salted Passwords": "when a password is first entered, the password program obtains a 12-bit random number [...] and appends this to the password typed in by the user." "When the user later logs in to the system, the 12-bit quantity is extracted from the password file and appended to the typed password."
  - ▶ First known use of salts with password hashing?



# Late 1970s - Unix: salting

---

- Salting "does not increase the task of finding any individual password, starting from scratch, but now the work of testing a given character string against a large collection of encrypted passwords has been multiplied by 4096 ( $2^{12}$ )."
  - ▶ With uniform distribution, there are ~3740 unique salts in 10,000 password file entries
- "it becomes impractical to prepare an encrypted dictionary in advance."
- "It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them"

# Speed of offline attacks (with salts)

## Assumptions:

- Unique per-user salts
- Non-targeted attack
  - ▶ Accounts are of equal value
  - ▶ No password strength hint

It is tough to limit offline attack speed to 1000/s (by password stretching). Obviously, if we need to handle more than 1000 requests/s ourselves, an attacker with the same resources will also be able to try at least as many.

Guesses / second	Users	Daily guesses / user
1,000	1	86,400,000
1,000	1,000	86,400
1,000	1,000,000	86
1,000	100,000,000	1
1,000,000,000	1	86,400,000,000,000
1,000,000,000	1,000	86,400,000,000
1,000,000,000	1,000,000	86,400,000
1,000,000,000	100,000,000	864,000

1 billion/s is a conservative GPU attack speed estimate for hashes without password stretching.

In practice, multi-billion speeds are often achieved.

# Late 1970s - Unix: "The Threat of the DES Chip"

---

- "Chips to perform the DES encryption are already commercially available and they are very fast."
- In `crypt(3)`, "one of the internal tables of the DES algorithm [...] is changed in a way that depends on the 12-bit random number. The E-table is inseparably wired into the DES chip, so that the commercial chip cannot be used."
  - ▶ Except for hashes on which the 12-bit salt happens to be zero

Food for thought: what if DES chips were installed into some computers running Unix, and instead of the E-table hack `crypt(3)` allowed for much higher iteration counts on those machines? (Indeed, salts would be implemented differently.)



# Late 1970s - Unix: salt & hash encoding syntax

---

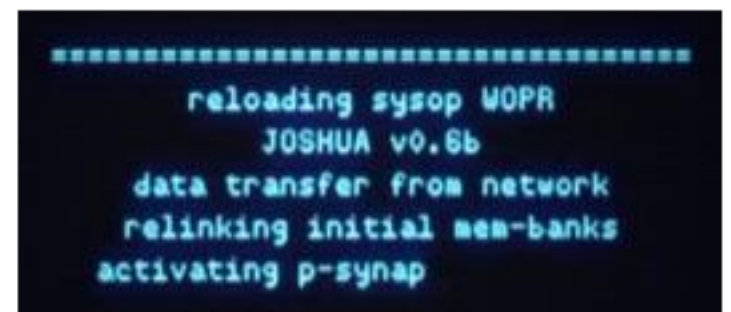
Unix /etc/passwd file excerpt (2.9BSD default, 1983 - but the same hash encoding is used since 7th Edition in late 1970s)

```
wnj:ZDjXDBwX1e2gc:8:2:Bill Joy,457E,7780:/a/guest/wnj:/bin/csh
dmr:AiInt5qKdjmHs:9:2:Dennis Ritchie:/a/guest/dmr:
ken:sq5UDrPlKj1nA:10:2:& Thompson:/a/guest/ken:
mike:KnKNwMkyCt8ZI:11:2:mike karels:/a/guest/mike:/bin/csh
carl:S2KiTfS3pH3kg:12:2:& Smith,508-21E,6258:/a/guest/carl:/bin/csh
joshua::999:2:&:/usr/games:/usr/games/wargames
```

The first two characters are the salt (12-bit), followed by 11 characters of the hash (64-bit)

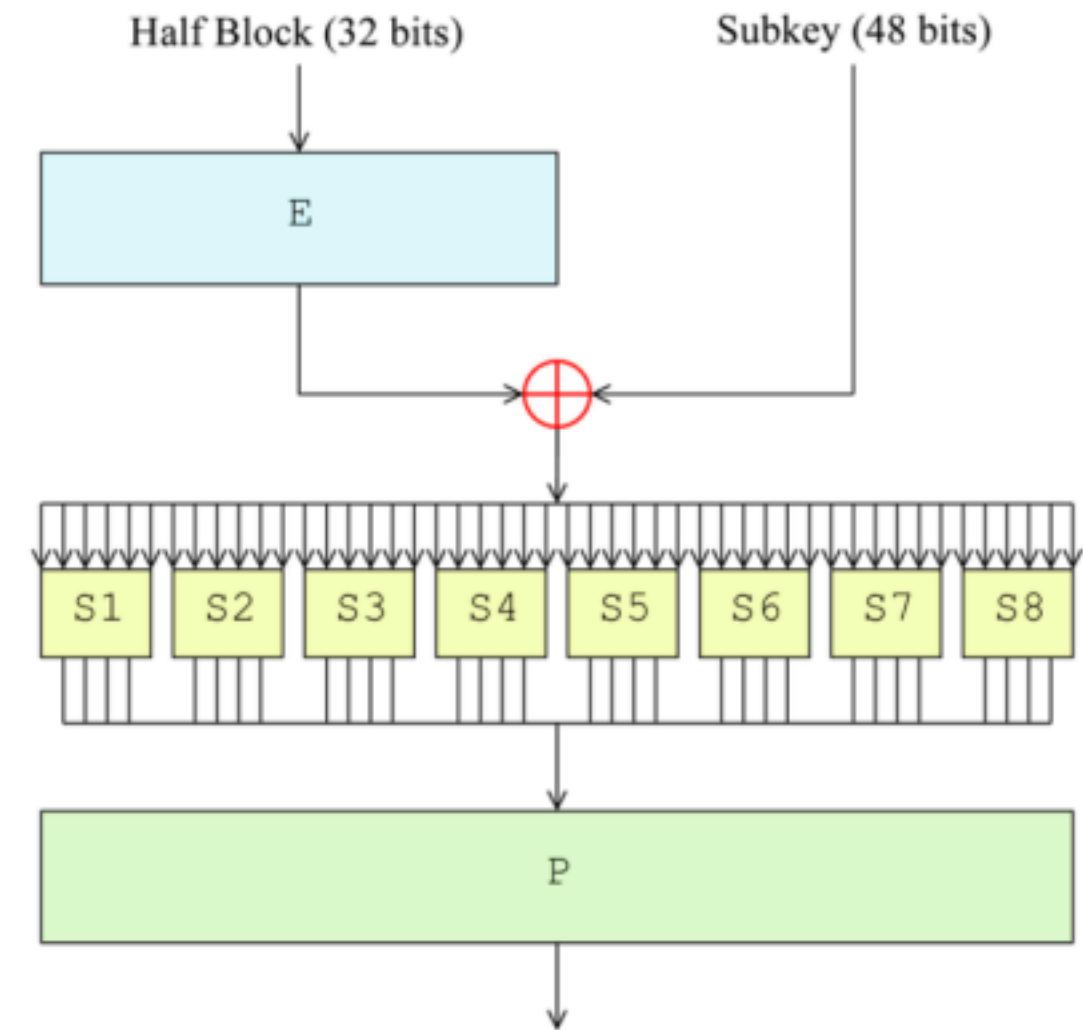
The WarGames movie came out in 1983, featuring war dialing and more.

"Joshua" is a backdoor username in the movie.



# Why was DES slow in software?

- Each S-box uses only 6 bits and produces 4 bits
  - ▶ Typical CPUs have much wider word size (16- to 64-bit, then even wider SIMD)
- Possible optimizations (late 1980s)
  - ▶ Spread the 6 and 4 data bits throughout up to 64-bit words to save on other overhead (E and P lookups, shifts to produce array indices)
  - ▶ Do two S-box lookups at once (12-to-8)
    - It is usually not practical to go further (combined tables become too large for fast access)
- Wasteful even with the above optimizations



This diagram illustrating one round of DES has been released into the public domain by its author, Matt Crypto

# Early 1980s: password cracker contests

---

- "The Second Official UNIX PASSWORD CRACKER CONTEST" (1983)

Newsgroups: net.general

Date: Thu Jan 6 08:02:37 1983

Subject: PASSWD CRACKER CONTEST

We proudly announce  
The Second Official  
UNIX PASSWORD CRACKER CONTEST

- Sensitive gentlewomen of the jury, this was not even the first

Submit your ingenious /etc/passwd password cracker program (source code) to the undersigned by January 31, 1983. We will test all programs for speed, portability, and elegance, on various Unix versions and on different machines. A manual page and a short writeup explaining the algorithm is a plus. The writers of the best three programs will win the Grand Prize, The Super Grand Prize, and The Ultra Grand Prize (and world-wide, ever lasting fame).

Ran Ginosar, Computer Technology Research Center,  
Bell Labs, Murray Hill.  
...!allegra!ran



# 1980s: no progress on the defense side?

---

- "A Fast Version of the DES and a Password Encryption Algorithm" by Matt Bishop, 1987
  - ▶ Speeds up crypt(3) by a factor of 10 to 20 through the use of larger/combined lookup tables (up to 200 KB total) and other optimizations
- Morris worm (1988) uses its own semi-fast implementation of DES-based crypt(3) to crack passwords on local accounts
  - ▶ 9 times faster on a VAX 6800 (45 passwords/second), needs only 6 KB
- Ultrix crypt16: poor attempt to overcome the 8-character limitation
- VMS introduces its own plethora of fancy password hashes

# Late 1980s to 1990s - Unix: password shadowing

---

- "Password shadowing first appeared in UNIX systems with the development of System V Release 3.2 in 1988 and BSD4.3 Reno in 1990."

Wikipedia

- Shadow Password Suite by Julianne Frances Haugh, 1988+
- It took many years for the various Unix-like systems, individual Linux distributions, etc. to catch up (although a few were pretty quick)

Password hashes were moved out of `/etc/passwd` and into file(s) not readable by regular users. Typical filenames are `/etc/shadow` (SysV and others) and `/etc/master.passwd` (BSD), although some "trusted" systems use per-user files under `/tcb` or `/etc/tcb`.

In 2001, Openwall GNU/\*/Linux made use of per-user shadow files under `/etc/tcb` to reduce privileges of the password-changing program, `passwd(1)` - something those "trusted" systems did not do

# Early 1990s: password security tools

---

- COPS by Dan Farmer
  - ▶ A local security auditing tool for Unix systems
  - ▶ Includes detection of poor passwords as one of the features
- Crack (for Unix) by Alec Muffett
- Cracker Jack (for DOS and OS/2) by The Jackal
- goodpass.c, and later CrackLib by Alec Muffett
  - ▶ "to be wired into "passwd" & "yppasswd", etc."
- npasswd, etc. with proactive password strength checking included



# Early 1990s: LM and NTLM hashes

---

- LAN Manager uses a particularly weak password hashing method ("LM hash") in its authentication protocol
  - ▶ Passwords are case-insensitive
  - ▶ An up to 14-character password is split after the 7th character and the two halves are used as DES encryption keys
  - ▶ The two password "halves" may be cracked separately, just like with crypt16 and bigcrypt, but much faster (no salt, shorter, case-insensitive, no iterations)
- Windows NT stores LM hashes, along with MD4-based NTLM hashes
  - ▶ Cracking the weaker LM hashes is usually enough
  - ▶ NTLM hashes are also a step back as compared to Unix crypt(3): no salt, no iterations
  - ▶ Knowledge of the NTLM hash is enough for network authentication
  - ▶ Plaintext passwords and hashes of logged-on users are stored in memory

# Early 1990s: BSDi configurable iteration count

---

- BSDi BSD/OS extends DES-based crypt(3) with proper support for long passwords, configurable iteration count stored along with each hash (the "J9.." below corresponds to 725 iterations), and 24-bit salt

```
_J9..saltLSQbyJrHIzg
```

- HP-UX, OSF/1, Digital Unix bigcrypt: poor attempt to fix crypt16
  - ▶ Still possible to crack the 8-character "halves" separately

```
y3hOhMyjWl8ZgRrZfS5BcgqE  
winniethepooh
```

# Mid 1990s: FreeBSD fixed iteration count

---

- FreeBSD MD5-based crypt(3) by Poul-Henning Kamp, 1994
  - ▶ Long passwords, 1000 iterations of MD5 (not configurable), up to 48-bit salt
  - ▶ "On a 60 Mhz Pentium this takes 34 msec" (source code comment)
    - A password cracking optimized reimplementation later ran 5 times faster (also on original Pentium)
  - ▶ In late 1990s, adopted by most Linux distributions and Cisco IOS
    - ... and eventually EOL'ed by the author in June 2012 - but nevertheless still in use

`$1$longsalt$0QgNqdKo00f5to4mPrBB3.`

- "How should a password algorithm be designed today? I'd use iterated, salted, locally-parameterized SHA or MD5 [...] I'd use an iteration count stored with the hashed password"

# 1995-1997: QCrack - crypt(3) precomputation

---

- During precomputation, each candidate password (typically a dictionary word or the like) is hashed with all 4096 possible salts. Then one byte is written out per hash (thus, 4 KB per candidate password)
- On a Pentium 133 MHz that would do ~12500 c/s with John the Ripper, having a 1 GB (an entire hard drive or tape) of QCrack-precomputed partial hashes would save at most 1 day of computation during an attack
- Usually not practical, but illustrates precomputation attacks prior to the advent of rainbow tables

Although there are anecdotes of people having used tapes with pre-computed DES-based crypt(3) hashes before, QCrack written in 1995-1997 by The Crypt Keeper appears to be the only generally available tool of this nature



# 1997: bitslice DES

---

- "A Fast New DES Implementation in Software", Eli Biham, 1997
  - ▶ "This implementation is about five times faster than the fastest known DES implementation on a (64-bit) Alpha computer, and about three times faster than than our new optimized DES implementation on 64-bit computers. [...] view the processor as a SIMD computer, i.e., as 64 parallel one-bit processors computing the same instruction."
  - ▶ ~100 gates per S-box
- "Reducing the Gate Count of Bitslice DES", Matthew Kwan, 1998+
  - S-box expressions released in 1998, technique presented in 1999, paper posted online many years later
  - ▶ 51 to 56 gates per S-box on average depending on available gates

The gate count was further reduced in later years by Marc Bevand (45.5 using Cell's "bit select" instruction), Dango-Chu (39.875, ditto), Roman Rusakov (32.875 with "bit selects", 44.125 without)

# Why is bitslice DES faster?

---

- A single instance of DES uses at most 12 bits per machine word
  - when doing 12-to-8 dual S-box lookups, which also typically exceed the size of L1 data cache
- With bitslicing, we compute e.g. 64 instances of DES in parallel on a 64-bit CPU - making full use of every bit in the 64-bit machine words
- We could compute multiple non-bitsliced instances of DES side-by-side and more fully use the machine word width in this way, but this requires support for vectorized array lookups for efficient implementation
  - ▶ In 2000s, some CPUs got SIMD permute instructions that are potentially usable: PowerPC AltiVec VPERM, Cell SHUFB, Intel SSSE3 PSHUFB, AMD XOP VPERM
  - ▶ 2013+: Intel Haswell microarchitecture is expected to include AVX2 VSIB (gather)

# 1998: validation vs. cracking speed ratio

---

Bitslice DES made it apparent that even an attacker possessing only the same kind of CPU that is used by the defender (such as in an authentication server) has a speed advantage resulting from the inherent parallelism of password cracking (test many passwords)

- ▶ "You can increase the iteration count, but you're limited with the validation time. [...] it is important to make sure that the best implementation of the same hash, but optimized for cracking (multiple keys at a time), is not much faster than the password validation function."
- ▶ "One-way hash choice: make sure it can't be made faster by a bitslice implementation, or mixing the instructions from two separate hashes (for higher issue rate). That is, the function should have a lot of natural parallelism, so that we can exploit it all in the validation function."

Solar Designer, "bitslice & crypt(3) choice", comp.security.unix posting, 1998

# 1990s: new concepts

---

- Key derivation function (KDF)
- Key stretching (password stretching) was formally defined and studied
  - ▶ J. Kelsey, B. Schneier, C. Hall, and D. Wagner, "Secure Applications of Low-Entropy Keys", 1997
  - ▶ A related concept became known as "strengthening" (throw salt away), but stretching ended up winning
- Passphrase
  - ▶ In PGP, S/Key, SSH, encrypted filesystems



# 1990s: network sniffing

---

- Non-switched Ethernet (10BASE2, 10BASE-T) was prevalent
- Network protocols typically transmitted passwords in the clear
  - ▶ This has started to change in mid-1990s
- A common attack was sniffing of passwords via a machine (such as a compromised server) on the same Ethernet segment with the target server or with some of the users
- This has contributed to the rise of more advanced authentication methods

# 1990s: alternative authentication methods

---

Some of these are an improvement, but they are susceptible to offline password guessing attacks on certain authentication material anyway:

- Challenge/response pairs

What may be worse, common protocols such as POP3 APOP and CRAM-MD5 are poorly defined, requiring that plaintext-equivalents be stored on the server, even though this was easy to avoid (like it is done in SCRAM, which took 13 years - from an RFC draft in 1997 - to become RFC 5802 in 2010, finally)

- Kerberos: TGTs, AFS user database

- S/Key, OPIE: skeykeys file

- SSH: passphrase on private key

- SRP: verifiers

Thus, passwords (or passphrases) are not going away, and proper password hashing or key derivation remains relevant even if as a component of other authentication schemes

# Other uses of passwords

---

A password or passphrase is also used to protect things such as:

- Encrypted home directories, filesystems, full disks
  - ▶ FileVault, EFS, TrueCrypt, LUKS/dm-crypt, eCryptfs, ...
- Archives (WinZip, RAR, ...)
- Wireless networks (WiFi WPA-PSK)
- PGP secret keys
- Mac OS X keychains, other "password vaults"

This will also keep passwords/phrases and decent KDFs relevant for many years to come

# 1996-2000: more password security tools

---

- John the Ripper by Solar Designer (later also by project contributors)
  - ▶ "Incremental mode" orders candidate passwords for decreasing estimated probability considering trigraph frequencies, yet is able to search a keyspace exhaustively given enough time
  - ▶ Takes advantage of 64-bit CPUs, MMX, bitslice DES (1998+)
- pam\_passwdqc by Solar Designer
  - ▶ An alternative to pam\_cracklib, with support for passphrases
  - ▶ Later became passwdqc tool set co-authored by Dmitry V. Levin
- L0phtCrack by L0pht Heavy Industries
  - ▶ Cracks LM and NTLM hashes used by Windows NT



# Effect of hash type and password policy

---

passwdqc vs. KoreLogic's DEFCON 2010 contest passwords

- Of the MD5-based crypt(3) hashes, teams cracked 33%
  - ▶ passwdqc with default policy would permit 3.5% of cracked or 1.1% of all
    - When a user's desired password is rejected, the user would not always pick a password that would not get cracked. Estimate: 1.9% would be crackable.
  - ▶ Of the uncracked passwords, passwdqc would reject 45% and permit 55%
- Of the NTLM hashes, teams cracked 94%
  - ▶ passwdqc with default policy would permit 35% of cracked or 33% of all
    - Estimate: 53% would be crackable
  - ▶ Of the few uncracked passwords, passwdqc would reject 14% and permit 86%

To withstand offline attacks, both a decent hash type and a decent password policy should be used at once

# Late 1990s: OpenBSD bcrypt

---

- "We present two algorithms with adaptable cost -- eksblowfish, a block cipher with a purposefully expensive key schedule, and bcrypt, a related hash function."

Niels Provos and David Mazieres, "A Future-Adaptable Password Scheme", The OpenBSD Project, 1999

- ▶ Configurable iteration count (encoded as base-2 logarithm), 128-bit salt
- ▶ In 2000 and later, adopted by Openwall GNU/\*/Linux, ALT Linux, OpenSUSE and made available as a non-default option on other \*BSDs and Solaris

```
$2a$08$128bitsalt22charslong0lHvsqGDe2t1XUwNgAVQ82BcG8Q8dWfu
```

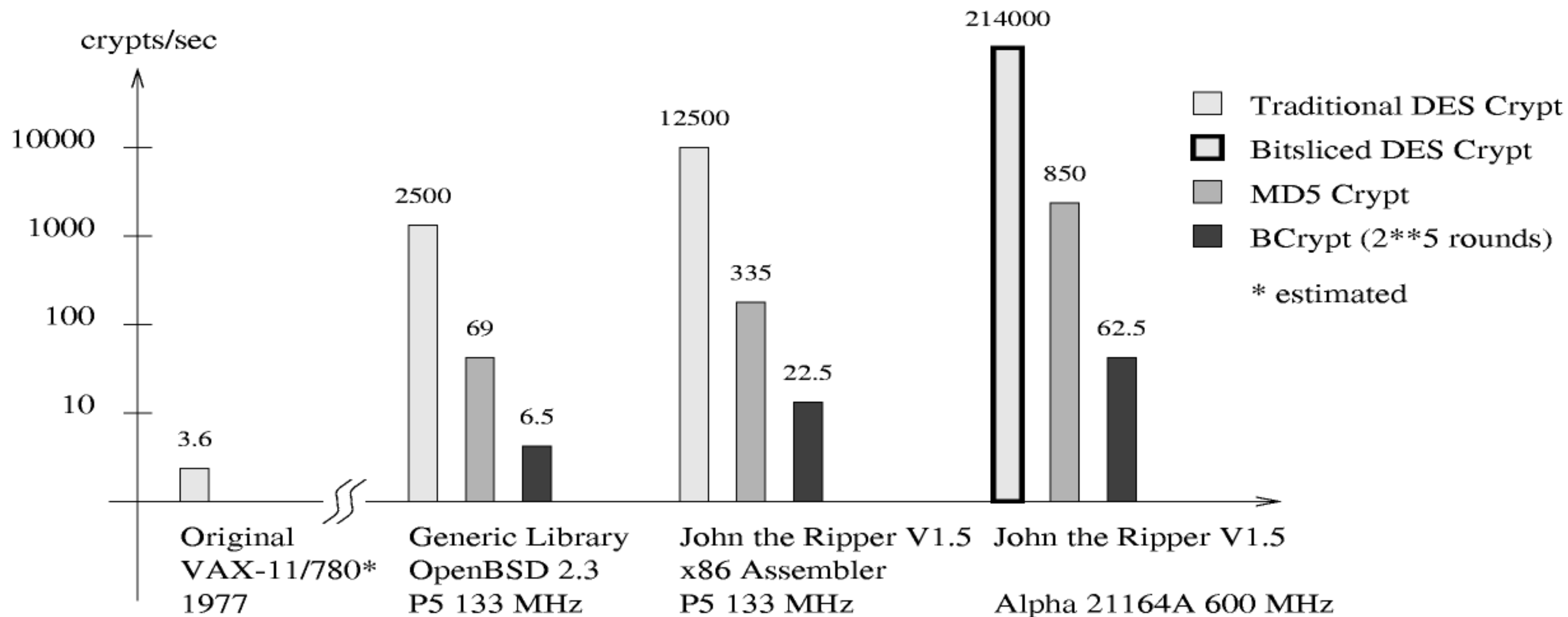
- 8-to-32 variable S-boxes, so uses 32 bits per machine word only
  - ▶ Would need scatter/gather (or at least gather) to overcome that (AVX2 VSIB, Intel MIC)

# What's wrong with bcrypt

---

- No parallelism, 32-bit word size - slows down defender
  - ▶ Low instructions per cycle (attack is ~2x faster), can't use SIMD
  - ▶ Attacker's use of SIMD is also impacted, though - except on devices with scatter/gather addressing (or at least gather)
    - Intel MIC (2012, limited availability), AVX2 (2013, will be widespread?)
- Low memory needs (only 4 KB) - defender's off-chip RAM is not put to use (only L1 cache is), attacker does not need to provide DRAM
  - ▶ Yet due to bcrypt's memory access pattern this turns out to be (barely) enough to defeat GPUs so far (AMD Radeon HD 7970 is only about as fast as a CPU)

# Late 1990s: crypt(3) speed comparison



Niels Provos and David Mazieres, "A Future-Adaptable Password Scheme", The OpenBSD Project, 1999



# 2012: modern crypt(3) cracking speeds

---

Since 1999

- crypt(3) cracking speeds for all flavors discussed so far have increased by a factor of 50 to 200 per CPU chip (for the fastest code and CPUs)
- Configurable iteration counts do help to compensate for that, although system defaults tend to be rather low to support a wide range of hardware and usage scenarios (e.g., bcrypt is typically used at costs only 8 to 32 times larger than the "2<sup>5</sup> rounds" baseline used in 1999 benchmarks)
  - ▶ Cracking speeds increase more rapidly than single password validation speeds
- AMD Radeon HD 7970 "Tahiti" GPU provides an additional boost of a factor of 5 for DES and 20 for MD5 (but none for bcrypt so far)

# 2000s: PBKDF2

---

- PKCS #5 v2.0 (1999), RFC 2898 (2000), NIST SP 800-132 (2010)
- Designed as a building block to use passwords and diversify keys in cryptographic protocols (not a password hash storage system)
- Several tunable parameters
  - ▶ A pseudorandom function to apply (such as HMAC-SHA-1)
  - ▶ Number of iterations
  - ▶ Derived key length
- Mainly used for deriving encryption keys (WinZip, OpenDocument, ...; DPAPI, 1Password, ...; FileVault, TrueCrypt, Android, ...), but also used in WiFi WPA-PSK and for password hash storage (Mac OS X 10.8, Django)

# What's wrong with PBKDF2

---

As commonly used with HMAC-SHA-\*

- No parallelism - slows down defender, but not attacker
  - ▶ When implemented on modern CPUs for defensive use, only a relatively small portion of resources available in one CPU core is used (can't use SIMD, low instructions per cycle)
- Almost no memory needs - defender's RAM is not put to use, attacker does not need to provide RAM
- GPU friendly
  - ▶ More so with SHA-1 than with SHA-512, though

SHA-512 uses 64-bit words, which helps CPUs and hurts current GPUs

# Late 1990s to 2007(+?): web apps use raw MD5

---

"Web apps" started to appear, usually written in PHP, so indeed they directly used PHP's `md5()` function for password hashing

PHP also offers `crypt()`, but those hashes were not sufficiently portable between systems (in 1990s, some Unix-like systems could lack DES-based `crypt(3)` because of US export regulations, and PHP is not Unix-only). This may be too complicated an explanation, though. Chances are that those web apps' developers simply did not know the options.

- ▶ No password stretching: Cracking speeds for one hash are about 1000 times higher than those for FreeBSD's MD5-based `crypt(3)`
- ▶ No salt: Effective cracking speeds (`{account, password}` combinations tested per second) for non-targeted attacks against large raw MD5 hash databases are even higher (times the number of hashes), precomputed hash tables may be used



# 2003: rainbow tables

---

- "As an example we have implemented an attack on MS-Windows password hashes. Using 1.4GB of data (two CD-ROMs) we can crack 99.9% of all alphanumerical passwords hashes ( $2^{37}$ ) in 13.6 seconds"

Philippe Oechslin, "Making a Faster Cryptanalytic Time-Memory Trade-Off", 2003

- ▶ Martin Hellman's time-memory trade-off (1980) enhanced and applied to password hashes
- Storage needs are a lot lower than for QCrack's naive approach
- Nevertheless, infeasible with large random salts
- Each hash being cracked requires extra processing
  - ▶ With a very large number of saltless hashes it may be quicker not to use rainbow tables, but instead to hash each candidate password and compare against all hashes being cracked, with an efficient comparison algorithm

# 2007+: web apps move to phpass

---

- phpass is an easy to use PHP password hashing class
  - ▶ phpass would use bcrypt if available (CRYPT\_BLOWFISH in PHP), and if not then fallback to BSDi-style extended DES-based hashes (CRYPT\_EXT\_DES in PHP), with "a last resort fallback to MD5-based salted and variable iteration count password hashes implemented in phpass itself (also referred to as portable hashes)"
- phpass has started to see some adoption by major web apps (WordPress, phpBB3, and Drupal) in 2007, which has helped further adoption
  - ▶ Not surprisingly, many of them chose to force the use of the "portable hashes", which unfortunately make less efficient use of the server's CPU

phpass was originally written during a security audit of an Openwall client's "web app" in 2004 in response to the findings (so that they could replace their weak password hashing). It was released publicly in 2005.

# phpass portable hashes

---

The portable hashes are very simple, which was key to phpass' acceptance. More elaborate "portable hashes" would likely not be accepted; this may be something to try for a next generation phpass now that the foot is in the door.

```
$P$Blongsalt3aidu9JAoAsLRML86yQuD0
```

In this example, "B" means  $2^{13}$  or 8192 iterations. The salt is 48-bit.

phpBB3 uses 2048 iterations, WordPress uses 8192 (but these hashes are compatible with each other). Drupal 7 uses a revision of these hashes with MD5 replaced by SHA-512, the prefix changed to "\$S\$", and iteration counts at 16384 and beyond (increasing between releases).



# 2007+: password cracking on GPUs

---

- Pioneered by Andrey Belenko of Elcomsoft
  - ▶ Initially for NTLM, LM, and raw MD5 hashes, achieving speeds of over 100M per second  
Beyond reach of existing software on CPUs at the time (except for Cell such as in PS3)
- Andrey and others improved the speeds and implemented other attacks
  - ▶ 2010: Whitepixel by Marc Bevand achieves 33.1 billion passwords/second against a single raw MD5 hash on a sub-\$3000 4x AMD Radeon HD 5970 computer (8 GPU chips)
  - ▶ 2012: oclHashcat-lite by atom does 10.9 billion on a single HD 6990 card (two GPU chips)
  - ▶ 2011: oclHashcat-plus by atom made GPUs usable for a full set of password cracking attacks on a wide variety of hashes (both "fast" and "slow" ones)
- John the Ripper is catching up with GPU support
  - ▶ 2011, 2012: more limited in GPU support than oclHashcat-plus, but Open Source



# 2007: SHA-crypt in glibc 2.7+

---

Introduced for political rather than technical reasons

- "Security departments in companies are trying to phase out all uses of MD5. They demand a method which is officially sanctioned. For US-based users this means tested by the NIST."

Ulrich Drepper, "Unix crypt using SHA-256 and SHA-512", 2007 (revised in 2008)

- Configurable iteration count (5000 by default), large salts

Ulrich's SHA-crypt.txt shows some confusion during SHA-crypt design: "the produced output is 32 or 64 bytes respectively in size. This fulfills the requirement for a large output set which makes rainbow tables less useful to impossible" - that's nonsense

```
$6$saltstring$svn8UoSVapNtMuqlukKS4tPQd8iKwSMHWj1/O817G3uBnIFNjnQJuesI68u4OTLiBFdcbYEdFCoEOfaS35inz1  
$6$rounds=10000$saltstringsaltst$OW1/O6BYHV6BcXZu8QVeXbDWra3Oeqh0sbHbbMCVNSnCM/UrjmM0Dp8vOuZeHBy/YTBmSK6H9qs/y3RnOaw5v.
```

# SHA-crypt analysis

---

- SHA-crypt hashes are decent (albeit not a technical improvement), especially the flavor based on SHA-512
  - ▶ Uses 64-bit machine words; not SIMD, though
- SHA-512 is not as GPU-friendly as e.g. MD5, yet is reasonable to attack on current GPUs (2012)
  - ▶ sha512crypt at default rounds=5000 can be attacked at ~12000 c/s on NVIDIA GTX 580 with John the Ripper (code by Claudio Andre) or hashcat, or at ~32500 c/s with hashcat on HD 6990 (two GPUs)
  - ▶ bcrypt at "\$2a\$08" (256 iterations), which is default on some systems, can be attacked at ~680 c/s on AMD FX-8120 3.1 GHz (combined speed for 8 threads) with John the Ripper; no speedup from GPUs yet (e.g. HD 7970 and 6990 achieve CPU-like speeds at bcrypt)

# 2009: sequential memory-hard functions

---

Defense against specialized hardware (ASICs, FPGAs, GPUs)

- "We introduce the concepts of memory-hard algorithms and sequential memory-hard functions, and argue that in order for key derivation functions to be maximally secure against attacks using custom hardware, they should be constructed from sequential memory-hard functions."

Colin Percival, "Stronger key derivation via sequential memory-hard functions", 2009

- ▶ General-purpose computers spend more die area on memory (RAM, caches) than on computation logic (ALUs, vector units) inside CPUs
- ▶ RAM is about as expensive to implement in cracking-optimized hardware, whereas computation logic is cheaper to implement in a specialized and massively-parallel fashion (avoiding the overhead on instruction decode, out-of-order execution, etc.)



# 2009: scrypt

---

- The scrypt KDF accepts three parameters tunable "according to the amount of memory and computing power available, the latency-bandwidth product of the memory subsystem, and the amount of parallelism desired".
- As defined, scrypt uses Salsa20/8 core as its main cryptographic primitive, which makes efficient use of up to 128-bit SIMD vectors

It should be possible to use wider SIMD vectors when  $p$  is greater than 1, but this is a trade-off

- A variation of scrypt based on another cryptographic primitive is possible (e.g., to please those requiring NIST-approved cryptography)
  - ▶ Alternatively, it may be shown that scrypt's cryptographic security is achieved by its initial and final use of PBKDF2 with SHA-256, whereas other processing is "non-cryptographic"



# ASIC/FPGA attacks on modern hashes

---

- PBKDF2-HMAC-SHA-1
- PBKDF2-HMAC-SHA-256
- sha256crypt
- PBKDF2-HMAC-SHA-512
- sha512crypt
- bcrypt
- scrypt

It is a sound approach to consider attacks with ASICs, but in practice attacks with less flexible devices are also relevant

  
Weaker

Stronger  


# GPU attacks on modern hashes

---

- PBKDF2-HMAC-SHA-1
- PBKDF2-HMAC-SHA-256
- sha256crypt
- PBKDF2-HMAC-SHA-512
- sha512crypt
- scrypt at up to ~1 MB (misuse)
  - Litecoin at 128 KB is ~10x faster on GPU vs. CPU
- bcrypt (uses 4 KB)
- scrypt at multi-megabyte memory
- Revised scrypt with TMTO defeater



Weaker

Stronger



# scrypt at low memory

---

- scrypt accesses memory in cache line sized chunks, which lets it use the memory bus efficiently
  - ▶ The attacker's cost is meant to be RAM itself, not bandwidth
- When scrypt is set to use only a small amount of memory (~1 MB or less), it is weaker than bcrypt at least as it relates to attacks on GPU
- At 128 KB, as demonstrated by scrypt's use in Litecoin, scrypt is ~10x faster on GPU than on CPU (whereas bcrypt is currently not faster on GPU than on CPU)

# script time-memory trade-off

---

- script deliberately allows for a time-memory trade-off
  - ▶ "The design of script puts a lower bound on the area-time product - you can use less memory and more CPU time, but the ratios stay within a constant factor of each other, so for the worst-case attacker (ASICs) the cost per password attempted stays the same"

Colin Percival, script and crypt-dev mailing lists posting, 2011

- Litecoin miners on GPU use this
- script may be revised to defeat the trade-off
  - ▶ Pros: fewer pre-existing hardware devices (GPUs, etc.) are efficient in an attack
  - ▶ Cons: not official script anymore, some defensive uses may be impacted as well (e.g., client-side hashing on mobile devices)



# What's wrong with scrypt

---

- ~100 ms corresponds to 32 MB memory usage on current server hardware
  - ▶ We could afford more RAM on dedicated authentication servers
- OTOH, in crypt(3) used by a Unix system distribution by default, even a few megabytes per thread might not be universally affordable (think VMs)
- At 1 ms, memory usage is so low that bcrypt is stronger
- Time-memory trade-off benefits attackers with GPUs
  - ▶ Can be fairly easily defeated, but then it's not official scrypt
- A single instance of Salsa20/8 might not contain enough natural parallelism to fully use a modern CPU core's execution units
  - ▶ In scrypt, only high-level parallelism is tunable

# Issues with scrypt for mass user authentication

---

At low durations and decent throughput, we face two problems:

- Low memory usage: acceptable at 100 ms (~32 MB), way too low at 1 ms
- Limited scalability on multi-CPU/multi-core when we maximize RAM usage
  - ▶ Optimized scrypt's SMix achieves a throughput of ~1500/s on a dual Xeon E5649 machine (12 cores, 24 logical CPUs) when running 24 threads (thus, latency 16 ms) at 4 MB/each
  - ▶ A cut-down hack (Salsa20 round count reduced from 8 to 2, SMix second loop iteration count reduced from N to N/4) achieves the same at 8 MB

This is sane speed and sane memory usage, but we want to do better - and we can

- ▶ scrypt paper recommends at least 16 MB
- ▶ scrypt at 128 KB (Litecoin) is ~10x faster to attack on GPU than on CPU

# 2012: current uses of scrypt

---

- As KDF in scrypt author's simple file encryption program
- As KDF in Tarsnap, scrypt author's "online backups for the truly paranoid"
- As KDF in Chromium OS to better protect the user's vault keyset
  - ▶ as well as to mitigate offline attacks on the Google Account password
- As proof-of-work scheme in Litecoin, "a coin that is silver to Bitcoin's gold"
  - ▶ 128 KB (misuse: too low) makes Litecoin ~10x faster to mine on GPUs than on CPUs
- As KDF in Phideli.us, constructing asymmetric keypairs from passwords
- There's an RFC draft on scrypt
- `crypt(3)` encoding syntax for scrypt is being considered

# A drawback of memory-hard KDFs

---

This applies to use of memory-hard KDFs for authentication in general, it is not specific to scrypt

- To use a lot of RAM fast, we need to get close to the full memory bandwidth, but this means poor scalability when many concurrent instances are run
- Thus, we have to choose between using more RAM per instance (and using CPUs' resources poorly when there are concurrent instances) and using CPU cores more fully (but at a lower RAM setting per instance)



# Revising scrypt's ROMix algorithm

Algorithm  $\text{ROMix}_H(B, N)$

Parameters:

$H$  A hash function.  
 $k$  Length of output produced by  $H$ , in bits.  
Integerify A bijective function from  $\{0, 1\}^k$  to  $\{0, \dots, 2^k - 1\}$ .

Input:

$B$  Input of length  $k$  bits.  
 $N$  Integer work metric,  $< 2^{k/8}$

Output:

$B'$  Output of length  $k$  bits.

Steps:

```
1:  $X \leftarrow B$ 
2: for  $i = 0$  to  $N - 1$  do
3:    $V_i \leftarrow X$ 
4:    $X \leftarrow H(X)$ 
5: end for
6: for  $i = 0$  to  $N - 1$  do
7:    $j \leftarrow \text{Integerify}(X) \bmod N$ 
8:    $X \leftarrow H(X \oplus V_j)$ 
9: end for
10:  $B' \leftarrow X$ 
```

◀ Can do it once

◀ Any iteration count

What if we reuse the same ROM across hash computations?

- Not a sequential memory-hard function anymore, but the ROM can be arbitrarily large for any throughput and latency
- Attacker's cost per candidate password tested is in ROM access ports and bandwidth

# ROM-port-hard functions

---

- Not a precise definition, more like word play on Colin Percival's sequential memory-hard functions concept
- We might access only a tiny fraction of array elements per hash computed (vs. scrypt's 100% write, 63% read), but that's OK as long as each element is equally likely to be needed and the access pattern is not predictable
- Because of the above and since the ROM must stay read-only, can't defeat the time-memory trade-off by modifying the array (we could in scrypt)
- However, can defeat the TMTO by pre-filling the ROM differently (not allowing for one array element to be quickly computed from another)

# Pros of ROM-port-hardness (vs. scrypt)

---

- Can use ~1000 times more memory on current server hardware (and require as much memory in each node for efficient attack: anti-botnet)
  - ▶ e.g., 10 to 240 GB of ROM (actually in RAM) vs. scrypt's 4 MB or 8 MB per thread
- Can use a server's full RAM capacity even when the current request rate is low (that is, when few hashes are being computed concurrently)
- Excellent scalability to more CPUs and CPU cores
  - ▶ We only need to increase the amount of processing between memory accesses to be such that we stay just below saturating the memory bandwidth when all CPU cores are in use (unlike with scrypt, this does not result in reduction of memory usage)
- Can use types of memory other than RAM (e.g., SSDs)



# Cons of ROM-port-hardness (vs. scrypt)

---

- Good scalability of attacks to more computing power (CPUs, CPU cores, GPUs, etc.) while not having to provide more memory
  - ▶ However, if the defender stayed just below saturating the memory bandwidth then the attacker may have to provide more bandwidth first
- A custom or otherwise more suitable hardware setup would have a larger number of ports to the same ROM capacity
  - ▶ Moreover, those don't have to be ports to the same large ROM - instead, separate smaller ROM banks may be used as long as bank conflicts are fairly rare
  - ▶ Yet at below ~1000 cores sharing a ROM the attack speed per die area will be lower than for scrypt, whereas with more cores the attacker will have to provide more interconnect and ROM ports, which will have a cost of its own



# The best of both worlds

---

- We can have a function that is
  - ▶ sequential memory-hard with a small amount of RAM (a few MB)
  - ▶ ROM-port-hard with a large amount of ROM (many GB)
- Trivial: compute e.g. scrypt and our ROM-port-hard function sequentially (feeding the output of one into the other) or independently (then combine the outputs e.g. using a fast hash)
  - ▶ Drawback: an attacker may use smaller memory machines to compute the scrypt portion
- Smarter: merge the two functions, interleave the memory access types
  - ▶ In scrypt terms, this can be done in SMix or BlockMix
  - ▶ The block size and relative frequency of small RAM and large ROM accesses may be tunable in case different memory types are being used or caching plays a role

# How about obese script?

---

- Computing our ROM content from a site-specific(?) seed value at service startup may take ~1 minute (e.g., 60 GB at 1 GB/s)
- Loading it from an SSD may take ~5 minutes (e.g., 60 GB at 200 MB/s)
  - ▶ Alternatively, we can mmap() it and start serving requests right away, getting to full speed in a few minutes as the content gets cached in RAM
- Then we don't have to store the seed value on the server - in fact, we don't have to store it at all (nor to ever have had it)
- The entire site-specific ROM would have to be stolen and distributed to all attack nodes (in addition to them needing this much RAM for sane speed)

# Taking obese script a step further

---

- Besides using an SSD to load our ROM content into RAM, we can keep a larger ROM on SSD - and use it from there
  - ▶ This may be achieved with `mmap()` and making less frequent, larger block size accesses
  - ▶ May want to prefetch data on a previous loop iteration, to avoid stalling computation
    - Consider: `madvise()`, `aio_read()`, helper thread
- If there's no seed value stored on the server, the intruder will have to copy and likely distribute the SSD ROM content to attack nodes
- Multiple SSDs and potentially even a NAS/SAN based on SSDs may be used to make the ROM even larger
  - ▶ Compared to blind hashing (refer to our ZeroNights 2012 slides) with database size similar to our SSD ROM's, this may be more practical and it has more obvious properties

# Issues with ROM on SSD

---

- SSD read disturb errors are a potential concern
  - ▶ The firmware would presumably prevent these by rewriting or relocating the block after "too many" reads have been made
  - ▶ Theoretically, these rewrites could in turn wear the SSD out
  - ▶ A back of the envelope calculation suggests that theoretically it'd take on the order of a million years until this problem would occur, assuming smart firmware
- If the firmware maintains per-block read counts and they're retrieved by an intruder, this may potentially allow for early-reject of some candidate passwords (block never accessed? this password must be wrong!)
  - ▶ Mitigation: start using the ROM SSD half way through computation of a hash



# Light use of ROM on SSD

---

- Rather than use our SSD ROM throughout hash computation, we can access it just once before a final cryptographically secure step (e.g., before the final PBKDF2-HMAC-SHA-256 invocation in a revision of scrypt)
  - ▶ This is much simpler to implement and it avoids the issues/concerns with using SSDs
  - ▶ It is friendly towards other uses of the same SSDs since we would only be making ~1000 requests/s from each machine (one request per hash computed), which is more than an order of magnitude below SSDs' IOPS capacity
- The attacker will need to have access to a copy of the SSD ROM for offline password cracking, but will not need to distribute it to attack nodes

# Late 1990s to 2000+: 2FA goes mainstream

---

- Many online services and especially banks have started to treat user-targeted attacks such as trojans and phishing seriously
- To this end, they deployed 2-factor authentication where passwords are augmented with one-time codes or another second authentication factor
  - ▶ There's some debate as to whether and which kinds of 2FA are effective against which types of attacks. "Two-factor authentication isn't our savior. It won't defend against phishing. It's not going to prevent identity theft. It's not going to secure online accounts from fraudulent transactions. It solves the security problems we had ten years ago, not the security problems we have today."





Bruce Schneier, "The Failure of Two-Factor Authentication", 2005

- Passwords remain relevant as one of the factors: "something you know"



# MULTIFACTOR AUTHENTICATION



KNOW	HAVE	ARE	DO
			
Passwords ID Questions Secret Images	Token (Smart) Card Phone	Face Iris Hand/Finger	Behavior Location Reputation



# Threat models

---

- Offline attacks

- ▶ Decent hash type
- ▶ Proper password stretching settings
- ▶ Random per-account salts

With targeted attacks (on few high-value accounts as opposed to lots of low-value ones), salts are of less help, yet they should be used in those cases as well

- ▶ Strict password policy

- Password reuse across sites

- Online attacks

- ▶ Password policy
  - At least ban top N most common passwords
- ▶ Per-source rate limiting
- ▶ Multi-factor authentication
- ▶ Behavior analysis
  - Akin to a "spam filter"
- ▶ User-targeted attacks
  - Phishing, trojans, client vulnerability exploits
- ▶ Network-based attacks
  - DNS, routing, MITM, old-fashioned sniffing
- ▶ Server vulnerability exploits



# Desirable properties of a future KDF

---

- These need to be configurable
  - ▶ With settings encoded along with password hashes, etc. - depending on specific use case
- Tunable high-level and instruction-level parallelism within one instance
  - ▶ Barely sufficient if the KDF is sequential memory-hard, otherwise abundant will do
- Ability to use almost arbitrarily wide SIMD vectors within one instance
- Running time almost independent of password length
- Existing hashes upgradable to higher iteration counts
  - ▶ without knowledge of the plaintext passwords
  - ▶ Maybe to higher memory cost and higher available parallelism as well? - tricky
- Friendly to whatever hardware we have in the defender's system
  - ▶ and to hardware that we might have there in the foreseeable future

# KDFs unfriendly to hardware we do not have

---

- This is controversial
  - ▶ Concept pioneered in DES-based crypt(3) being unfriendly to existing DES chips
- If our authentication server only has CPUs and RAM, then the KDF being GPU-unfriendly is a plus
  - ▶ However, future server CPUs might have embedded GPUs or similar
- If we have an FPGA or ASIC in the server, being CPU-unfriendly is a plus
  - ▶ e.g., this may slow down attacks with a botnet, where victims' computers will generally not have specialized hardware
- Multiple blocks friendly to different hardware components that we have
  - ▶ However, complexity is the enemy of security and reliability
- Configurable unfriendliness (set of blocks to use and their weights)

# CPU + RAM friendliness

---

- CPU-friendly

- ▶ 32- or 64-bit integers, SIMD
- ▶ Sequential memory-hard functions
- ▶ "Large" variable S-boxes

However, this prevents use of SIMD until AVX2 VSIB

- ▶ Use of specialized instructions - e.g., AES-NI

- CPU-unfriendly

- ▶ Small S-boxes (variable or/and fixed) - waste machine word bits

Beware of bitslicing and byte permute instructions

- ▶ Bit permutations

Such instructions are to appear in 2013 in non-SIMD form, though

- ▶ Unusual transforms

e.g., integer addition with only partial carry (an adder with some holes punched)

# GPU friendliness

---

- GPU-friendly

- ▶ 32-bit integers, SIMD
- ▶ No/low memory needs
- ▶ Low register pressure

- GPU-unfriendly

- ▶ Exceeding total memory available on typical GPU cards

However, there may be a practical recomputation vs. memory trade-off - in fact, it's deliberate in scrypt

- ▶ Variable S-boxes exceeding fast memory size per thread (work-item in OpenCL parlance)
- ▶ Data-dependent branching - may work, but is tricky and risky - let's not do it

Deep enough tree (not just an if/else in a loop) to make eager execution inefficient

Beware of side-channel leaks - a sufficient reason not to use data-dependent branching

- ▶ CPU-unfriendly algorithms

Small S-boxes, bit permutations, unusual transforms



# FPGA/ASIC friendliness

---

- FPGA/ASIC-friendly

- ▶ Small S-boxes (variable or/and fixed)
- ▶ Bit permutations

- ASIC-friendly

- ▶ Unusual yet simple transforms

e.g., integer addition with only partial carry (an adder with some holes punched)

```
#define pcadd(a, b, mask) (((a) ^ (b)) + (((a) & (b) & (mask)) << 1)) & 0xff)
```

- FPGA/ASIC-unfriendly

- ▶ Sequential memory-hard functions

All of the friendliness examples in this and previous slides assume that sufficient parallelism is available in one instance of a KDF, as suggested before

# Local parameter

---

- Must contain sufficient entropy
  - ▶ way beyond a typical password or even passphrase
- Hashes are not crackable offline without knowledge of the local parameter
- However, if the local parameter is stored right on the authentication server or in the password database, then it may be stolen/leaked along with the hashes
- Problem: migration of locally-parameterized hashes between systems with different local parameters, changing the local parameter after a compromise
  - ▶ Solution: embed a "local parameter ID" in the hash encodings, support multiple local parameters at once

# Unreadable local parameter

---

- When a KDF is at least partially implemented in a dedicated device (e.g., in a hardware security module or even a dedicated server), it becomes possible to embed a local parameter in the device
- If the local parameter is unreadable by the host system (e.g., by a server doing password authentication), this buys us an extra layer of security
  - ▶ Need to have a backup copy - e.g., a cluster of multiple HSMs or/and a piece of paper in CEO's safe

Companies like Google and Facebook could use this approach to substantially reduce the impact of a possible user/password database compromise. Clearly, they can afford to move password hashing onto HSMs or dedicated servers. In fact, they could benefit from hardware acceleration of password hashing.

# Network structure (logical)

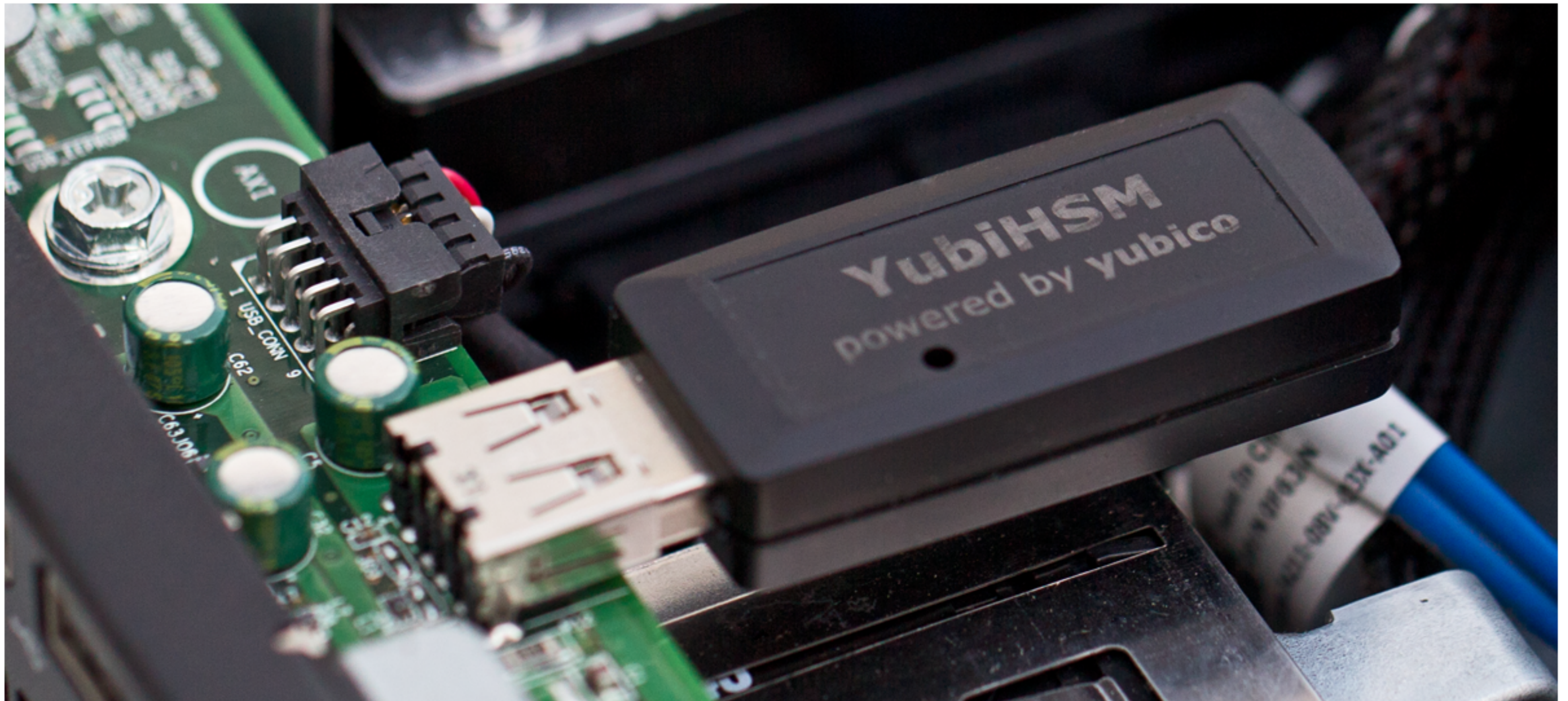
---

- Authentication servers
  - ▶ Receive usernames and passwords, reply with yes/no or a token
  - ▶ Optionally perform the costly portion of password hashing
  - ▶ Access the database, talk to password hashing HSMs or servers for the portion involving the local parameter
- Password hashing HSMs or servers
  - ▶ Are accessible from the authentication servers only
  - ▶ Receive partially computed hashes or passwords to hash, return computed hashes
- Other servers needing user authentication
  - ▶ Talk to authentication servers or/and accept tokens



# YubiHSM - a USB dongle for servers

---



YubiHSM in a server's internal USB port. Photo (c) Yubico, reproduced under the fair use doctrine.



# Local parameter in YubiHSM

---

YubiHSM provides several suitable functions.

If we use HMAC-SHA-1:

- Key is the local parameter
- "Key handle" is its ID
- "Data" is output of a KDF
- HMAC is password hash

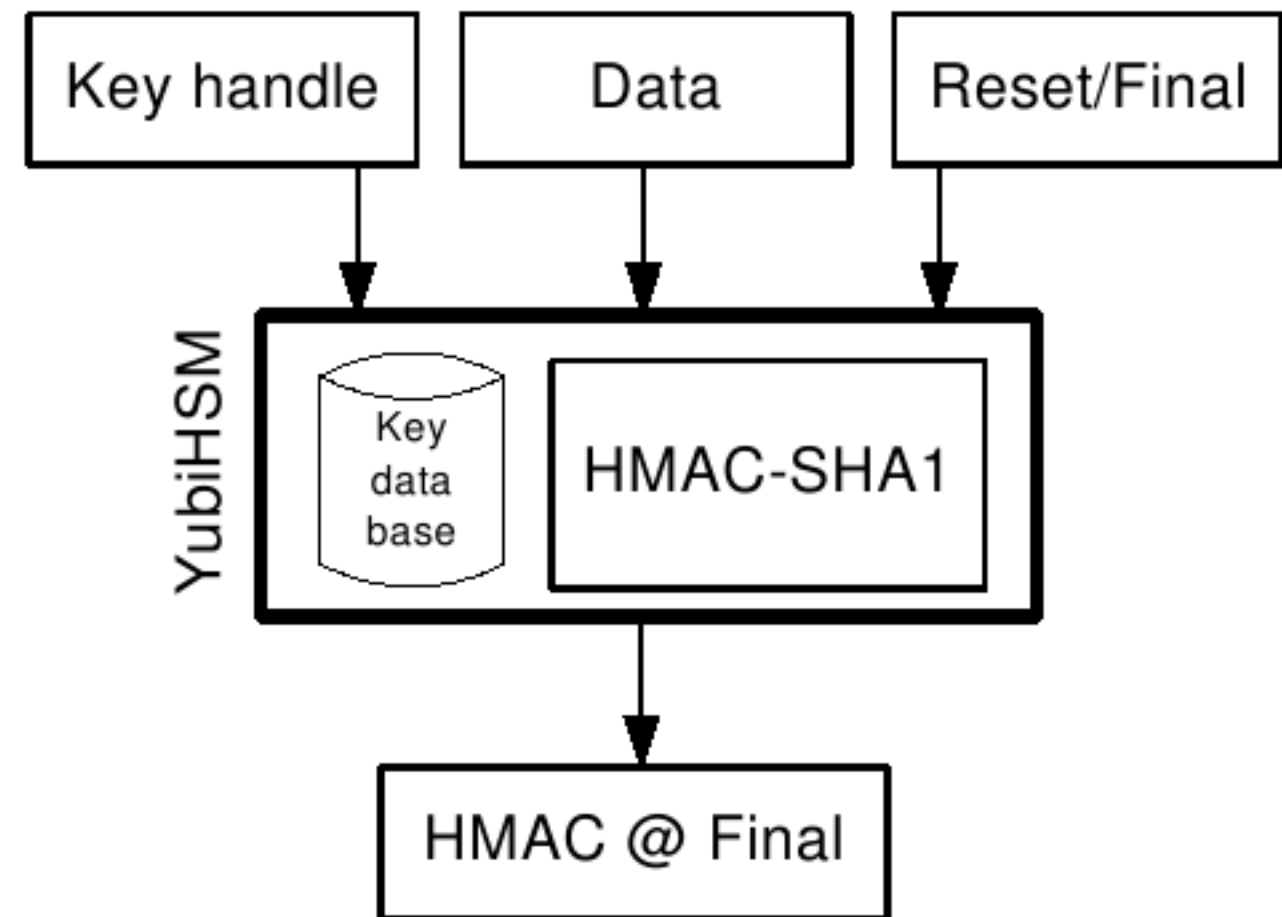


Diagram (c) Yubico, reproduced under the fair use doctrine

# YubiHSM pros

---

- Similar purpose, thus the right threat model
- Per key permission flags
- No custom OS kernel level driver required (USB CDC)
- Well-documented APIs, sample code
- Low cost (\$500; other HSMs may be \$10k to 20k EUR)
  - ▶ You need at least two for redundancy
- Independent formal analysis of the Yubikey protocol
  - ▶ "YubiSecure? Formal Security Analysis Results for the Yubikey and YubiHSM" by Robert Künnemann and Graham Steel, INRIA

Assumes "that the implementation is correct with respect to the documentation"

Found an oversight, which Yubico has since released a security advisory on

# YubiHSM cons

---

- No independent whitebox audits
- No independent blackbox audits
  - ▶ probing for implementation issues
- USB CDC is slow, up to ~500 requests per second sustained throughput
- Serial interface for block-oriented data is risky
  - ▶ "careful design is required not to lose synchronization in a serial byte stream" (Yubico)
- Not tamper-resistant (physical attacks are outside of the threat model)



**Matthew Green**

@matthew\_d\_green

.@agl\_\_ @ioerror HSMs can and should be audited. Obviously this isn't trivial, but it's critical if you're going to use one.



# Issues with HSMs in general

---

- Purpose and threat model are not always suitable

- ▶ Crypto acceleration or/and security

- At least symmetric crypto is often not faster than optimized code on CPU anyway

- ▶ Attacks from compromised host or/and physical

- Potential vulnerabilities

- ▶ Firmware bugs, design errors, side-channels

- Attack surface (too many features, each being a risk - can disable or not?)

- ▶ No known whitebox audits, source code not available for review

- Interfaces (physical, driver, API) and their reliability

- Cost is often significant

- ▶ Especially given that multiple HSMs need to be installed

# KDFs in scripting languages (future phpass?)

---

- We're limited in our choice of cryptographic primitives, especially if portability to other scripting languages is desired
  - ▶ MD5 is the most ubiquitous common denominator, but use of SHA-512 is more appropriate by other criteria
- Include parallelism so that we may eventually benefit from it
  - ▶ e.g., when support for next generation phpass hashes gets embedded into PHP proper
- Feed moderately large amounts of data into the available cryptographic primitives so that we save on interpreter or VM overhead
  - ▶ e.g., invoke PHP's SHA-512 implementation on strings that are several kilobytes long - enough to keep the call overhead to a minimum, yet still within L1 cache
- Sequential memory-hard functions may be practical

# Need to resist the temptation

---

Please resist the temptation to customize password hashing in your own web application or the like

- It is too easy to get it wrong
- Having too many different password hash types in active use is undesirable
  - ▶ Difficult to migrate hashes between systems
  - ▶ Existing password security auditing tools are not immediately usable, so administrators of individual installs of your app won't be able to audit the security of their users' passwords

Yet a determined attacker will implement this, then distribute the tool to others

- Further research, experiments, discussions within the community are needed - to arrive at as few next generation KDFs as practical



# Questions?

<http://www.openwall.com>

@Openwall

**Solar Designer**

<[solar@openwall.com](mailto:solar@openwall.com)>

@solardiz

**Simon Marechal**

<[bartavelle@openwall.com](mailto:bartavelle@openwall.com)>

@bartavelle