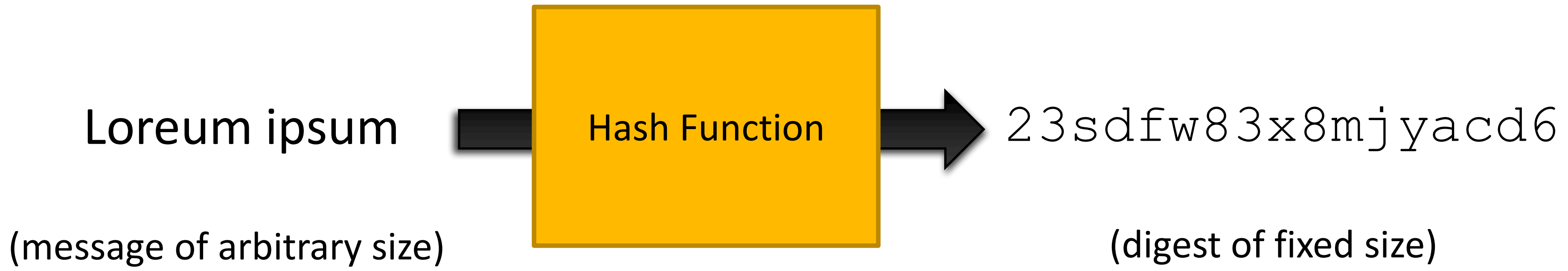
Cryptographic Hash Functions

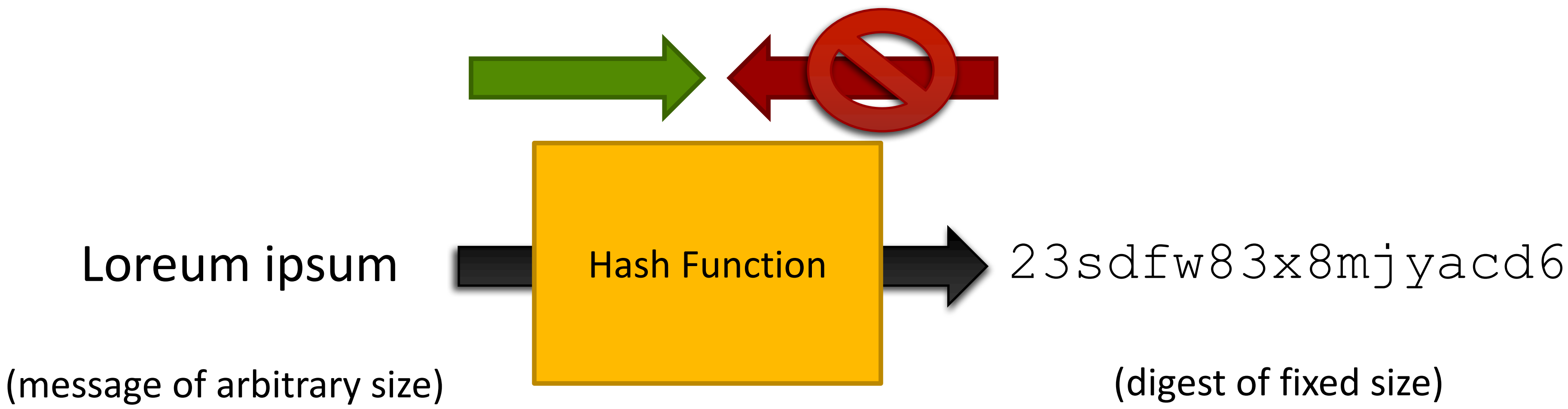
Cryptographic Hash Functions

- Input: Message of arbitrary size
- Output: “Digest” (hashed output) of fixed size



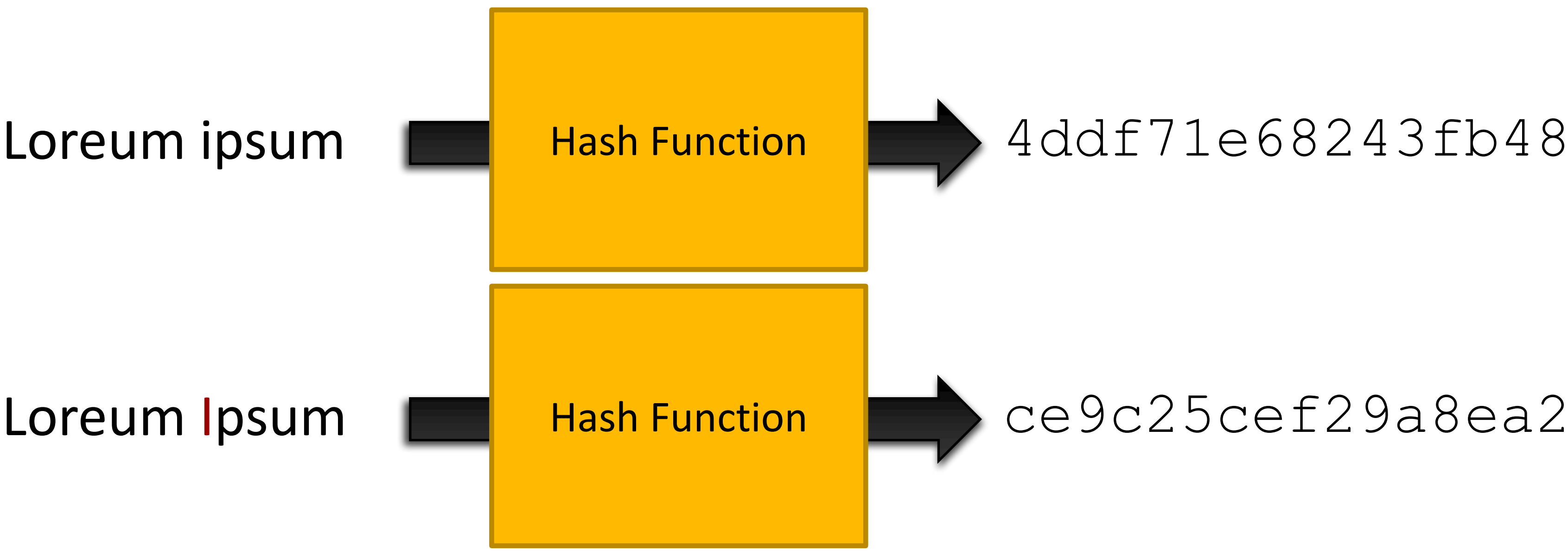
Cryptographic Hash Functions

- Design Goals
  - **Computing hash** should be computationally cheap
  - **Reversing hash** should be computationally expensive ("impossible") – One-way function



Cryptographic Hash Functions

- Design Goals
  - Changing the message a small amount should produce a **large change** in the digest
  - *Each bit in digest has 50% chance of flipping*



Cryptographic Hash Functions

- Design Goals
  - It should be very (very very VERY) hard to find two different messages that have the same digest

Cryptographic Hash Uses

- Security
  - Digital signatures
  - Message authentication
- General computing
  - Detect duplicate files
  - Detect file changes/corruption
  - Index data in hash tables

Cryptographic Hash Functions

- MD5 – **Don’t use!**
  - Input → 128 bit digest
- SHA-1 – **Don’t use!**
  - Input → 160-bit digest
  - Google, Apple, Microsoft, Mozilla retired support for SHA-1 signed SSL/TSL certificates in ‘16-’17
- Vulnerable to collision attacks
  - Attackers have made fake SSL certificates

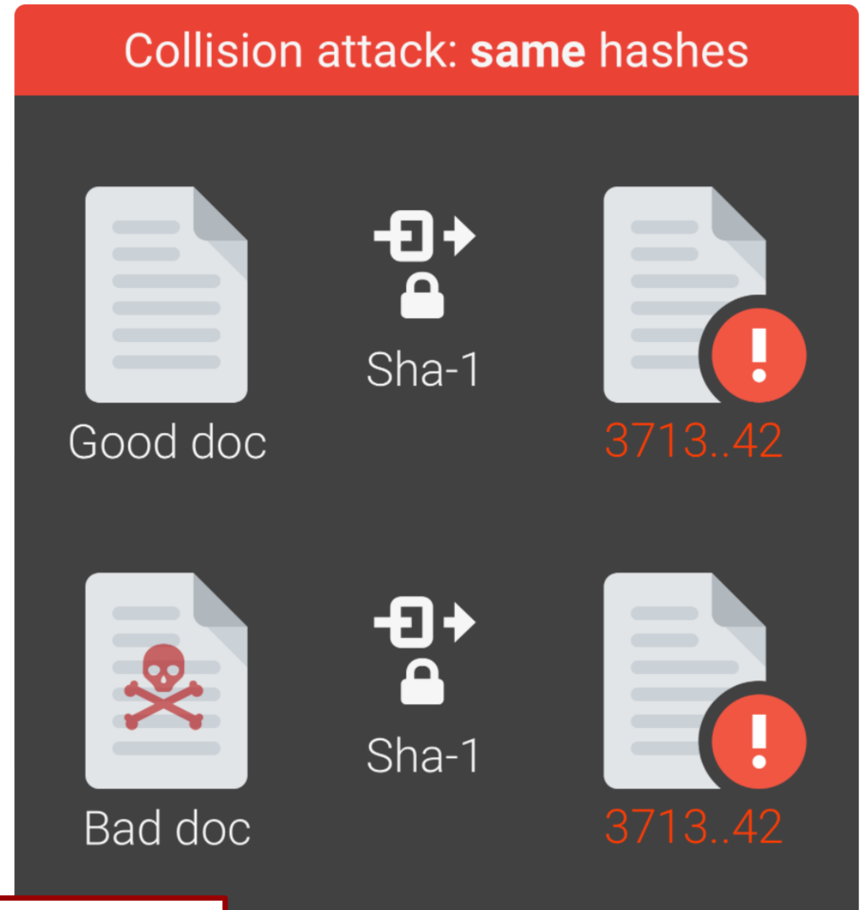
SHATTERED

We have broken SHA-1 in practice.

This industry cryptographic hash function standard is used for digital signatures and file integrity verification, and protects a wide spectrum of digital assets, including credit card transactions, electronic documents, open-source software repositories and software updates.

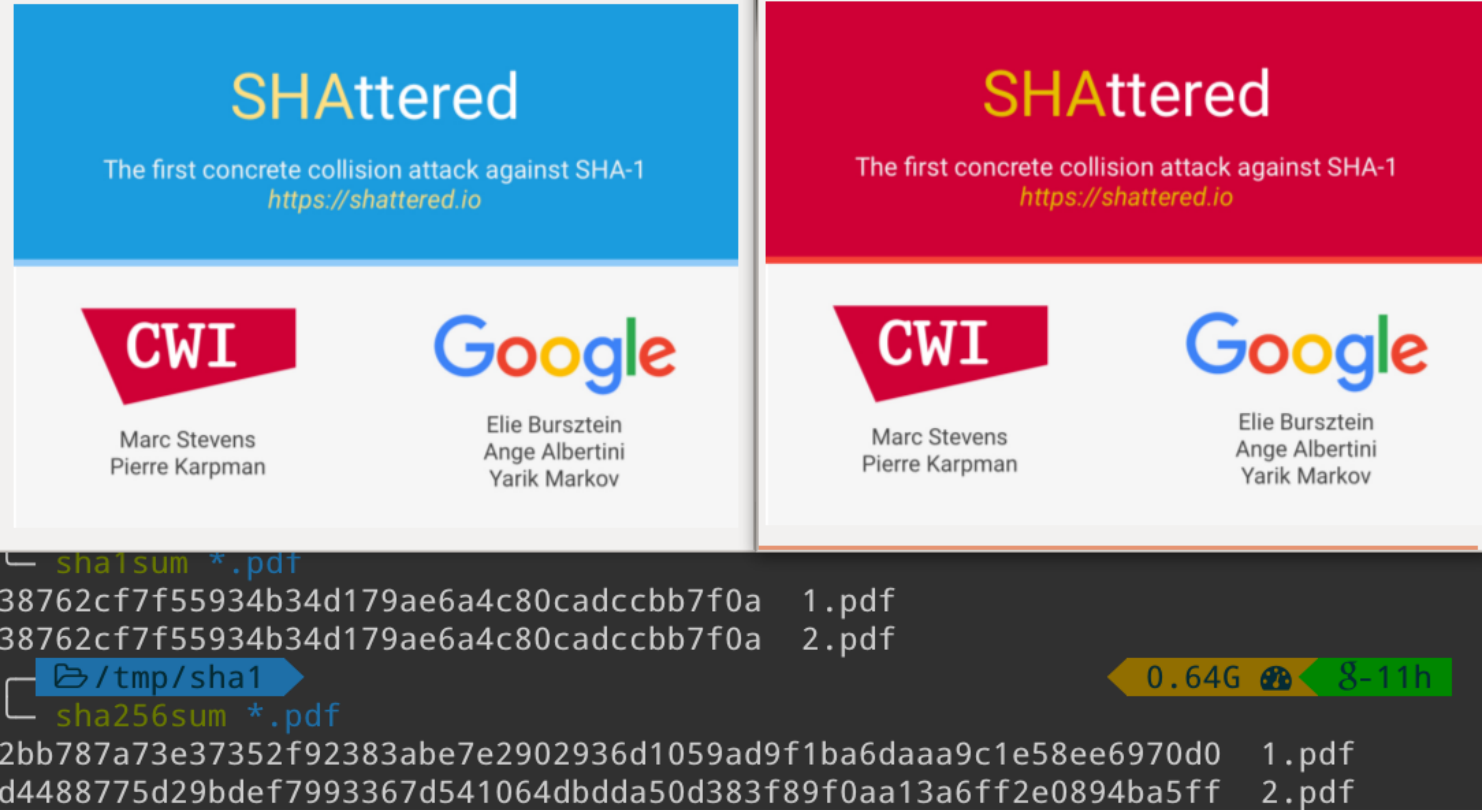
It is now practically possible to craft two colliding PDF files and obtain a SHA-1 digital signature on the first PDF file which can also be abused as a valid signature on the second PDF file.

For example, by crafting the two colliding PDF files as two rental agreements with different rent, it is possible to trick someone to create a valid signature for a high-rent contract by having him or her sign a low-rent contract.



Infographic | Paper

https://shattered.io/
February 2017



Google produced two <u>different</u> PDFs with <u>same SHA-1 hash</u> as proof of danger

Required 9,223,372,036,854,775,808 SHA1 computations

110 years of Single-GPU computation *(but Google has more than one GPU...)*

https://shattered.io/
February 2017

Cryptographic Hash Functions

- SHA-2 family – **Safe (except for length extension)**
  - SHA-256 (256-bit digest, optimized for 32-bit CPUs)
  - SHA-512 (512-bit digest, optimized for 64-bit CPUs)
- SHA-3 – **Safe (including against length extension)**
  - NIST Hash function competition (2007-2012)
    - 51 entries *round 1*, 14 *round 2*, 5 *finalists*
  - Winner: Keccak algorithm
    - Efficient in hardware but slow in software
    - SHA3-256, SHA3-512, …
- Blake2 – **Safe**
  - Another SHA-3 finalist

Length Extension Attacks

- Older hash algorithms output their entire internal state as the hash digest
  - Attack: Pick up exactly where they left off!
(Reconstruct internal state from hash digest)

Plaintext	**Hash (md5, SHA-1, SHA-2)**
`FundsXfer:Account 123456:Amount:123`	`4ddf71e68243fb48ce9c25cef29a8ea2`

`FundsXfer:Account 123456:Amount:123`**000** 😈	Load hash function with state of
	`4ddf71e68243fb48ce9c25cef29a8ea2`
	Continue running hash function over extension attack digits **000**
	New hash:
	`30c6ae0de5369c2637d5c541ef0095d8`

Length Extension Attacks

- HashPump: A tool to exploit the hash length extension attack in various hashing algorithms.
  - Currently supported algorithms: MD5, SHA1, SHA256, SHA512 *(i.e. SHA2 variants)*
  - https://github.com/bwall/HashPump
- Real-world attacks require a bit of brute forcing (trial and error) to reconstruct hash state but nothing impossible

Password Storage

Password Storage

We agree that it's horrible to store plaintext passwords in a database, yes?

- Database theft instantly gives attacker all user passwords ☹
  - Attacker could be rogue system administrator… ☹
- Humans re-use passwords across many sites ☹
- Does a website password reset tool email you your original password? **RUN!!!**

Password Storage

- Encrypting the *entire database* doesn't help
  - Attacker could easily steal encryption keys along with database data – keys must be in the system somewhere
- Encrypting individual passwords is a similar headache
  - Where to store the keys?
  - How to keep the keys safe?
  - *So many keys!!*

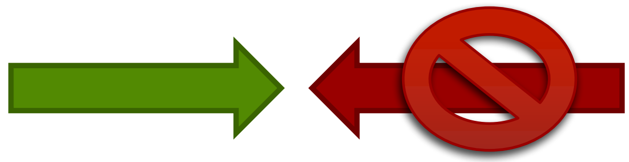
Warning!

Can hashes help us?

Warning: Cryptographic Hashes for password storage are **wrong**!

Password Storage

alice@abc.com
password:



“Swordfish”

4ddf71e68243fb4

- Alice’s plaintext password can’t be instantly reversed from the hash if database stolen ✔
- But what if Bob has the same password? He will have the same hash ☹

Password Storage

- Humans choose **terrible** passwords:
  - `password`, `swordfish`, `passw0rd`, etc…
- There are only a few plausible hash functions in widespread use
- Attackers can **pre-compute** hashes for likely passwords (dictionary words and permutations)
  - Save in "rainbow table"
  - Search for a quick match!

Password Lists

- Large lists of likely passwords are assembled by attackers from prior password leaks (real-world data)
- Free/cheap option for your downloading convenience
  - https://crackstation.net/buy-crackstation-wordlist-password-cracking-dictionary.htm
  - 15GB uncompressed
  - Starting guessing at "password123" instead of "aaaaaaaa"

Password Storage

- Improvement: Don't hash {password}
  - Instead hash {salt | password}
- "Salt" is large (160 bit) cryptographically random number appended/prepended to password
- Best practice
  - Unique salt **per user**, not per-system
  - Store this in database along with hash
- Rainbow tables now worthless
  - Would need a rainbow table for *each* 2^{160} salt values)

Password Storage

- Many systems use just a single salt, so an attacker only needs to compute one rainbow table ☹
- Per-user salts are still fundamentally broken, just *harder* to crack ☹
  - Cryptographic hash functions are intended to be fast
  - Attackers that steal your database also have your salt. With **GPUs** they can brute-force all possible passwords (following the password list and permutations)
  - Broken? Not instantly. But vulnerable? Yes

Password Storage

"Please stop hashing passwords"

https://blog.tjll.net/please-stop-hashing-passwords/

Password Storage

- Password storage should use a **Key Derivation Function** (**KDF**) instead
  - It looks like a hash function, but has a completely different design goal
- Design goals
  - KDF: **hard** to compute
    - Ideally, as slow as your users will tolerate without switching to a competitor product!
  - Cryptographic hash: Easy to compute

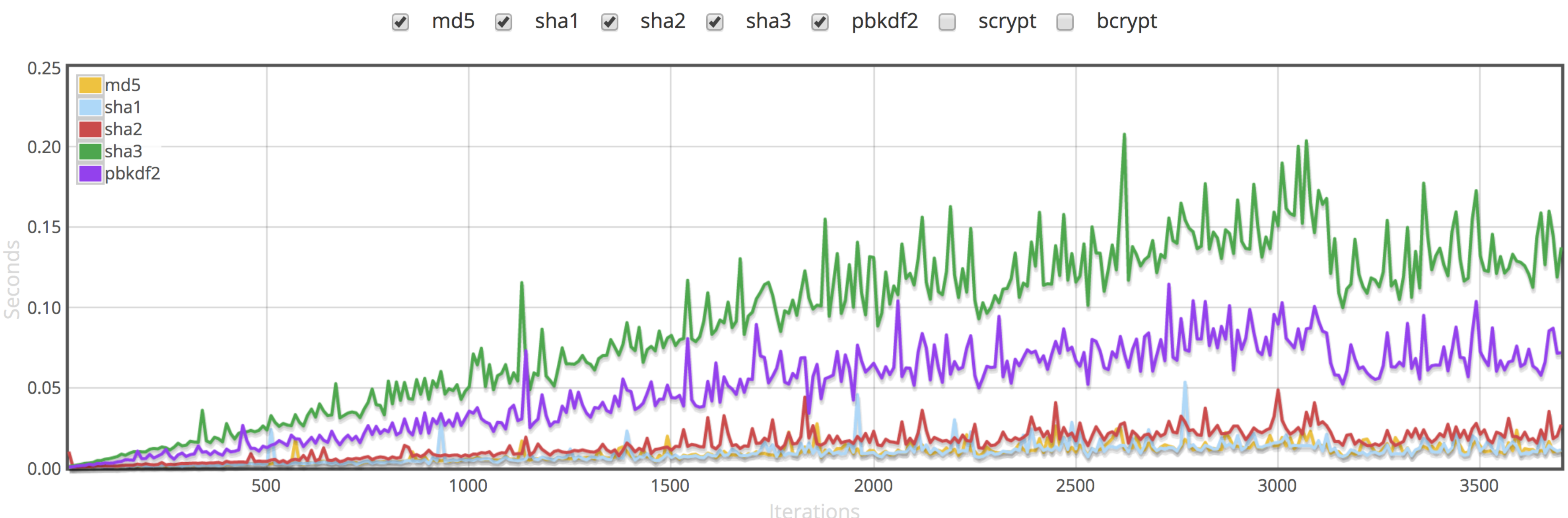
Key Derivation Functions

- **Bcrypt – good**
  - Tunable time-hard – you can configure how much CPU time it takes to calculate a ~~hash~~ key
  - CPUs getting faster? Tune bcrypt to take more time!
- **Scrypt – good**
  - Tunable time (CPU) and space (memory) hard
  - GPUs brute-forcing is hampered due to memory requirements
- **Important: Still use salt with KDF algorithms**

Key Derivation Functions

Comparing hash functions by time to generate digest
md5, sha1, sha2, sha3, pbkdf2

How do you think bcrypt and scrypt will compare?



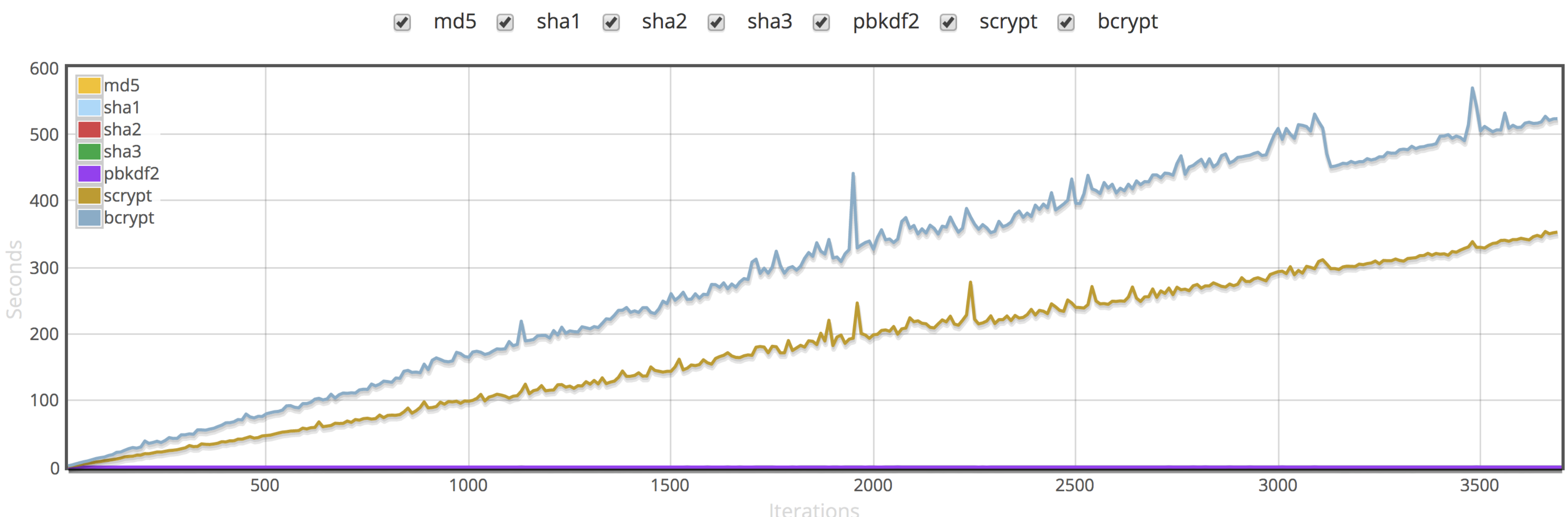
https://blog.tjll.net/please-stop-hashing-passwords/

(CORS policy requires changing JavaScript to load JSON over HTTPS to get interactive graph to appear…)

Key Derivation Functions

Original hashes (md5, sha1, sha2, sha3, pbkdf2) are not even visible at the bottom!

Y-axis (original): 0.00 – 0.25s
Y-axis (new): 0-600s



https://blog.tjll.net/please-stop-hashing-passwords/

(CORS policy requires changing JavaScript to load JSON over HTTPS to get interactive graph to appear…)

Key Derivation Functions

- Ruby script to generate your own dataset
  - https://gist.github.com/tylerjl/10802499