scrypt: A key derivation function

Doing our best to thwart TLAs armed with ASICs

Colin Percival
Tarsnap
cperciva@tarsnap.com

December 4, 2012

What are key derivation functions?

- You have a password.

What are key derivation functions?

- You have a password.
- You want a generate a *derived key* from that password.

What are key derivation functions?

- You have a password.
- You want a generate a *derived key* from that password.
    - Verifying passwords for user authentication.

What are key derivation functions?

- You have a password.
- You want a generate a *derived key* from that password.
  - Verifying passwords for user authentication.
  - Encrypting or signing files.

What are key derivation functions?

- You have a password.
- You want a generate a *derived key* from that password.
    - Verifying passwords for user authentication.
    - Encrypting or signing files.
- In most situations where passwords are used, they are passed to a key derivation function first.

What are key derivation functions?

- You have a password.
- You want a generate a *derived key* from that password.
  - Verifying passwords for user authentication.
  - Encrypting or signing files.
- In most situations where passwords are used, they are passed to a key derivation function first.
  - In most situations where key derivation functions aren't used, they should be!

What are key derivation functions?

- You have a password.
- You want a generate a *derived key* from that password.
  - Verifying passwords for user authentication.
  - Encrypting or signing files.
- In most situations where passwords are used, they are passed to a key derivation function first.
  - In most situations where key derivation functions aren't used, they should be!
- Examples of key derivation functions:
  - DES CRYPT [R. Morris, 1979]
  - MD5 CRYPT [P. H. Kamp, 1994]
  - bcrypt [N. Provos and D. Mazières, 1999]
  - PBKDF2 [B. Kaliski, 2000]
  - MD5 (not designed to be a key derivation function!)

Security of key derivation functions

- Attack model: Assume that the attacker can mount an offline attack.

Security of key derivation functions

- Attack model: Assume that the attacker can mount an offline attack.
  - Attacker has access to `/etc/master.passwd` and wants to find the users' passwords.

Security of key derivation functions

- Attack model: Assume that the attacker can mount an offline attack.
    - Attacker has access to `/etc/master.passwd` and wants to find the users' passwords.
    - Attacker has an encrypted file and wants to decrypt it.

Security of key derivation functions

- Attack model: Assume that the attacker can mount an offline attack.
  - Attacker has access to `/etc/master.passwd` and wants to find the users' passwords.
  - Attacker has an encrypted file and wants to decrypt it.
- For strong key derivation functions, the only feasible attack is to repeatedly try passwords until you find the right one.

Security of key derivation functions

- Attack model: Assume that the attacker can mount an offline attack.
  - Attacker has access to `/etc/master.passwd` and wants to find the users' passwords.
  - Attacker has an encrypted file and wants to decrypt it.
- For strong key derivation functions, the only feasible attack is to repeatedly try passwords until you find the right one.
  - Also known as a "brute force" attack.

Security of key derivation functions

- Attack model: Assume that the attacker can mount an offline attack.
    - Attacker has access to `/etc/master.passwd` and wants to find the users' passwords.
    - Attacker has an encrypted file and wants to decrypt it.
- For strong key derivation functions, the only feasible attack is to repeatedly try passwords until you find the right one.
    - Also known as a "brute force" attack.
    - If you can do better than brute force, the crypto is "broken".

Security of key derivation functions

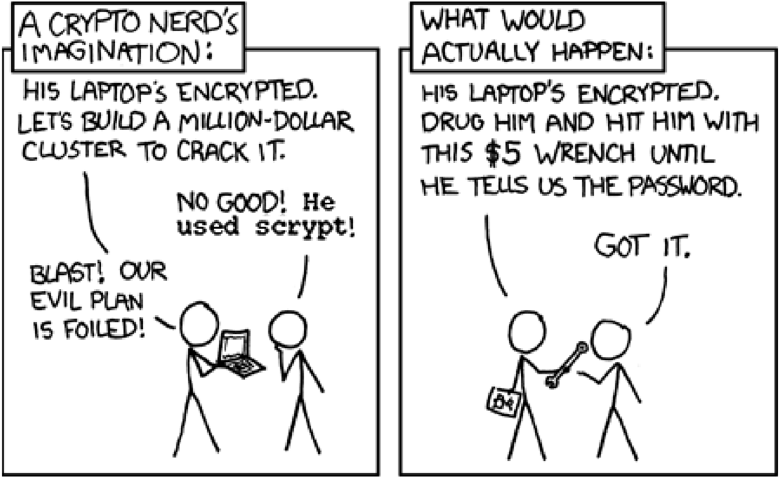
- Attack model: Assume that the attacker can mount an offline attack.
  - Attacker has access to `/etc/master.passwd` and wants to find the users' passwords.
  - Attacker has an encrypted file and wants to decrypt it.
- For strong key derivation functions, the only feasible attack is to repeatedly try passwords until you find the right one.
  - Also known as a "brute force" attack.
  - If you can do better than brute force, the crypto is "broken".
- If it takes twice as long to compute a derived key, it will take twice as long to find the right password.

Security of key derivation functions

- Attack model: Assume that the attacker can mount an offline attack.
  - Attacker has access to `/etc/master.passwd` and wants to find the users' passwords.
  - Attacker has an encrypted file and wants to decrypt it.
- For strong key derivation functions, the only feasible attack is to repeatedly try passwords until you find the right one.
  - Also known as a "brute force" attack.
  - If you can do better than brute force, the crypto is "broken".
- If it takes twice as long to compute a derived key, it will take twice as long to find the right password.
  - . . . as long as the attacker is using the same software as you.

Hardware-based brute force attacks

Hardware-based brute force attacks



CREDIT: Randall Munroe / xkcd.com

Hardware-based brute force attacks

- The challenge of key derivation functions is to make a brute-force attack as expensive as possible.

Hardware-based brute force attacks

- The challenge of key derivation functions is to make a brute-force attack as expensive as possible.
  - Since the attacker can always use the same system as you, this really means *minimizing the attacker's advantage* in computing derived keys.

Hardware-based brute force attacks

- The challenge of key derivation functions is to make a brute-force attack as expensive as possible.
  - Since the attacker can always use the same system as you, this really means *minimizing the attacker's advantage* in computing derived keys.
  - Usually the "good guys" are running software on general-purpose computers.

Hardware-based brute force attacks

- The challenge of key derivation functions is to make a brute-force attack as expensive as possible.
  - Since the attacker can always use the same system as you, this really means *minimizing the attacker's advantage* in computing derived keys.
  - Usually the "good guys" are running software on general-purpose computers.
  - In the worst case (NSA), the attackers have custom-designed ASICs.

Hardware-based brute force attacks

- The challenge of key derivation functions is to make a brute-force attack as expensive as possible.
  - Since the attacker can always use the same system as you, this really means *minimizing the attacker's advantage* in computing derived keys.
  - Usually the "good guys" are running software on general-purpose computers.
  - In the worst case (NSA), the attackers have custom-designed ASICs.
- Using ASICs, it is possible to pack many copies of a cryptographic circuit onto a single piece of silicon.

Hardware-based brute force attacks

- The challenge of key derivation functions is to make a brute-force attack as expensive as possible.
  - Since the attacker can always use the same system as you, this really means *minimizing the attacker's advantage* in computing derived keys.
  - Usually the "good guys" are running software on general-purpose computers.
  - In the worst case (NSA), the attackers have custom-designed ASICs.
- Using ASICs, it is possible to pack many copies of a cryptographic circuit onto a single piece of silicon.
- Moore's law: Every 18–24 months, a new generation of semiconductor manufacturing processes makes CPUs faster.

Hardware-based brute force attacks

- The challenge of key derivation functions is to make a brute-force attack as expensive as possible.
    - Since the attacker can always use the same system as you, this really means *minimizing the attacker's advantage* in computing derived keys.
    - Usually the "good guys" are running software on general-purpose computers.
    - In the worst case (NSA), the attackers have custom-designed ASICs.
- Using ASICs, it is possible to pack many copies of a cryptographic circuit onto a single piece of silicon.
- Moore's law: Every 18–24 months, a new generation of semiconductor manufacturing processes makes CPUs faster.
    - ... password-cracking ASICs get faster AND can fit more copies of a password-cracking circuit.

Hardware brute-force attack cost

- The cost of a hardware brute-force attack is measured in dollar-seconds.

Hardware brute-force attack cost

- The cost of a hardware brute-force attack is measured in dollar-seconds.
  - Password cracking is embarrassingly parallel, so if you use twice as much hardware you can crack the key in half the time.

Hardware brute-force attack cost

- The cost of a hardware brute-force attack is measured in dollar-seconds.
  - Password cracking is embarrassingly parallel, so if you use twice as much hardware you can crack the key in half the time.
- Cost of ASICs $\asymp$ size of ASICs.

Hardware brute-force attack cost

- The cost of a hardware brute-force attack is measured in dollar-seconds.
  - Password cracking is embarrassingly parallel, so if you use twice as much hardware you can crack the key in half the time.
- Cost of ASICs $\asymp$ size of ASICs.
  - A strong key derivation function is one which can only be computed by using a large circuit for a long time.

Hardware brute-force attack cost

- The cost of a hardware brute-force attack is measured in dollar-seconds.
    - Password cracking is embarrassingly parallel, so if you use twice as much hardware you can crack the key in half the time.
- Cost of ASICs $\asymp$ size of ASICs.
    - A strong key derivation function is one which can only be computed by using a large circuit for a long time.
- J. Kelsey, B. Schneier, C. Hall and D. Wagner, 1998: Use "32-bit arithmetic and moderately large amounts of RAM".

Hardware brute-force attack cost

- The cost of a hardware brute-force attack is measured in dollar-seconds.
    - Password cracking is embarrassingly parallel, so if you use twice as much hardware you can crack the key in half the time.
- Cost of ASICs $\asymp$ size of ASICs.
    - A strong key derivation function is one which can only be computed by using a large circuit for a long time.
- J. Kelsey, B. Schneier, C. Hall and D. Wagner, 1998: Use "32-bit arithmetic and moderately large amounts of RAM".
    - An example of a "moderately large amount of RAM": 1 kB.

Hardware brute-force attack cost

- The cost of a hardware brute-force attack is measured in dollar-seconds.
  - Password cracking is embarrassingly parallel, so if you use twice as much hardware you can crack the key in half the time.
- Cost of ASICs $\asymp$ size of ASICs.
  - A strong key derivation function is one which can only be computed by using a large circuit for a long time.
- J. Kelsey, B. Schneier, C. Hall and D. Wagner, 1998: Use "32-bit arithmetic and moderately large amounts of RAM".
  - An example of a "moderately large amount of RAM": 1 kB.
- If we use a *ridiculously* large amount of RAM, hardware attacks will be even more expensive.

Sequential memory-hard functions

Definition

A *sequential memory-hard function* is a function which
(a) can be computed on a Random Access Machine in $T(n)$ operations using $S(n) = O(T(n))$ memory; and
(b) cannot be computed on a Parallel Random Access Machine with $S^*(n)$ processors and $S^*(n)$ space in expected time $T^*(n)$ where $S^*(n)T^*(n) = O(T(n)^{2-x})$ for any $x > 0$.

Sequential memory-hard functions

Definition

A *sequential memory-hard function* is a function which
(a) can be computed on a Random Access Machine in $T(n)$ operations using $S(n) = O(T(n))$ memory; and
(b) cannot be computed on a Parallel Random Access Machine with $S^*(n)$ processors and $S^*(n)$ space in expected time $T^*(n)$ where $S^*(n)T^*(n) = O(T(n)^{2-x})$ for any $x > 0$.

- Since $S^*(n)$ is the circuit area required, this means that the area-time product increases as roughly the *square* of the time spent by the defender, assuming he doesn't run out of RAM.

Sequential memory-hard functions

Definition

A *sequential memory-hard function* is a function which
(a) can be computed on a Random Access Machine in $T(n)$ operations using $S(n) = O(T(n))$ memory; and
(b) cannot be computed on a Parallel Random Access Machine with $S^*(n)$ processors and $S^*(n)$ space in expected time $T^*(n)$ where $S^*(n)T^*(n) = O(T(n)^{2-x})$ for any $x > 0$.

- Since $S^*(n)$ is the circuit area required, this means that the area-time product increases as roughly the *square* of the time spent by the defender, assuming he doesn't run out of RAM.
- Note that this does not say *how* that area-time product is reached — in particular, it does not rule out using less area and more time ("time-memory trade-off").

ROMix

Algorithm (ROMix)

Given a random oracle H, an input B, and an integer parameter N, compute

$$V_i = H^i(B) \qquad 0 \leq i < N$$

and $X = H^N(B)$, then iterate

$$j \leftarrow \mathit{Integerify}(X) \bmod N$$
$$X \leftarrow H(X \oplus V_j)$$

N times; and output X.

ROMix

Algorithm (ROMix)

Given a random oracle H, an input B, and an integer parameter N, compute

$$V_i = H^i(B) \qquad 0 \leq i < N$$

and $X = H^N(B)$, then iterate

$$j \leftarrow \mathit{Integerify}(X) \bmod N$$
$$X \leftarrow H(X \oplus V_j)$$

N times; and output X.

- The function *Integerify* can be any bijection from $\{0,1\}^k$ to $\{0 \ldots 2^k - 1\}$.

ROMix

Algorithm (ROMix)

Given a random oracle H, an input B, and an integer parameter N, compute

$$V_i = H^i(B) \qquad 0 \leq i < N$$

and $X = H^N(B)$, then iterate

$$j \leftarrow \mathit{Integerify}(X) \bmod N$$
$$X \leftarrow H(X \oplus V_j)$$

N times; and output X.

- The function *Integerify* can be any bijection from $\{0,1\}^k$ to $\{0 \ldots 2^k - 1\}$.
- ROMix fills V with pseudorandom values, then accesses them in a pseudorandom order.

ROMix

Lemma

For a fixed input B, given M copies of a random oracle H which can be simultaneously consulted in unit time, and an index of size M, there is no algorithm which for computing $H^x(B)$ for for a random $x \in \{0 \ldots N-1\}$ completes in expected time less than $\frac{N}{4M+2} - \frac{1}{2}$.

Proof (sketch).

Suppose an algorithm exists, and run N copies of algorithm in parallel, one copy with each possible value of x.
We can bound the number of values $H^\alpha(B)$ which have been input to oracles in the first i timesteps by $(2M+1) \cdot (i+1)$ by considering how many different oracles are "consistent with observations" up to that point.
The result follows (with some algebra). $\square$

Theorem

The class of functions ROMix are sequential memory-hard.

Proof.

Since H is a random oracle, the values $j = \mathrm{Integerify}(X) \bmod N$ act as random values which cannot be computed prior to each value of X being available; and computing each $V_j = H^j(B)$ takes (from the lemma) at least $\Omega(n/S^*(n))$ time.
Since we iterate n times, this provides $T^*(n) = \Omega(n^2/S^*(n))$ and thus $S^*(n)T^*(n) = \Omega(n^2) \neq \mathrm{O}(T(n)^{2-x})$ as required, since $T(n) = \mathrm{O}(n)$. $\square$

- Turning ROMix into a key derivation function:

- Turning ROMix into a key derivation function:
  - Use PBKDF2 to convert password and salt into a bitstream.

- Turning ROMix into a key derivation function:
  - Use PBKDF2 to convert password and salt into a bitstream.
  - Feed this bitstream to ROMix.

- Turning ROMix into a key derivation function:
  - Use PBKDF2 to convert password and salt into a bitstream.
  - Feed this bitstream to ROMix.
  - Feed the output of ROMix back to PBKDF2 to generate the derived key.

scrypt

- Turning ROMix into a key derivation function:
  - Use PBKDF2 to convert password and salt into a bitstream.
  - Feed this bitstream to ROMix.
  - Feed the output of ROMix back to PBKDF2 to generate the derived key.
- Since we use PBKDF2 as a one-way entropy "spreading" function, rather than for any computational cost, we can safely set its iteration count to 1.

- Turning ROMix into a key derivation function:
  - Use PBKDF2 to convert password and salt into a bitstream.
  - Feed this bitstream to ROMix.
  - Feed the output of ROMix back to PBKDF2 to generate the derived key.
- Since we use PBKDF2 as a one-way entropy "spreading" function, rather than for any computational cost, we can safely set its iteration count to 1.
- We use ROMix to make the computation expensive.

- Turning ROMix into a key derivation function:
  - Use PBKDF2 to convert password and salt into a bitstream.
  - Feed this bitstream to ROMix.
  - Feed the output of ROMix back to PBKDF2 to generate the derived key.
- Since we use PBKDF2 as a one-way entropy "spreading" function, rather than for any computational cost, we can safely set its iteration count to 1.
- We use ROMix to make the computation expensive.
  - Thanks to the "wrapping" with PBKDF2, we don't need much *cryptographic* strength from ROMix — only that it takes a long time to compute.

Maximizing the constant factor

- H doesn't need to be a random oracle or even anything approximating one: The only real requirement is that it *must not have any shortcuts to iteration*.

Maximizing the constant factor

- H doesn't need to be a random oracle or even anything approximating one: The only real requirement is that it *must not have any shortcuts to iteration*.
- Assuming there are no computational shortcuts, the cost to compute ROMix in hardware is proportional to:

$$\begin{aligned}
&[\text{Memory required}] \cdot [\text{Time required}] \\
&\quad = \frac{T_{\text{Software}} \cdot [\text{Size of } H \text{ output}]}{[\text{Time to compute } H \text{ in software}]} \cdot \frac{T_{\text{Software}} \cdot [\text{Time to compute } H \text{ in hardware}]}{[\text{Time to compute } H \text{ in software}]} \\
&\quad = T_{\text{Software}}^2 \cdot \frac{[\text{Bandwidth of software } H \text{ output}]}{[\text{Hardware:Software speed ratio for } H]} \\
&\quad = T_{\text{Software}}^2 \cdot \frac{[\text{Bandwidth of software } H \text{ output}]^2}{[\text{Bandwidth of hardware } H \text{ output}]}
\end{aligned}$$

Maximizing the constant factor

- H doesn't need to be a random oracle or even anything approximating one: The only real requirement is that it *must not have any shortcuts to iteration*.
- Assuming there are no computational shortcuts, the cost to compute ROMix in hardware is proportional to:

$$
\begin{aligned}
&[\text{Memory required}] \cdot [\text{Time required}] \\
&\quad = \frac{T_{\text{Software}} \cdot [\text{Size of } H \text{ output}]}{[\text{Time to compute } H \text{ in software}]} \cdot \frac{T_{\text{Software}} \cdot [\text{Time to compute } H \text{ in hardware}]}{[\text{Time to compute } H \text{ in software}]} \\
&\quad = T^2_{\text{Software}} \cdot \frac{[\text{Bandwidth of software } H \text{ output}]}{[\text{Hardware:Software speed ratio for } H]} \\
&\quad = T^2_{\text{Software}} \cdot \frac{[\text{Bandwidth of software } H \text{ output}]^2}{[\text{Bandwidth of hardware } H \text{ output}]}
\end{aligned}
$$

- The area required to compute H is irrelevant, since the total area used will be determined almost completely by the RAM.

Maximizing the constant factor

H	Software perf.	Hardware perf.	Score $(= \mathrm{SW}^2/\mathrm{HW})$
SHA256	450 Mbps	1250 Mbps	160 Mbps
Blowfish	800 Mbps	1000 Mbps	640 Mbps
AES-128	1200 Mbps	40000 Mbps	36 Mbps
Salsa20/8	2000 Mbps	2000 Mbps	2000 Mbps
Keccak	fast	very very fast	not very good

Maximizing the constant factor

H	Software perf.	Hardware perf.	Score ($= \mathrm{SW}^2/\mathrm{HW}$)
SHA256	450 Mbps	1250 Mbps	160 Mbps
Blowfish	800 Mbps	1000 Mbps	640 Mbps
AES-128	1200 Mbps	40000 Mbps	36 Mbps
Salsa20/8	2000 Mbps	2000 Mbps	2000 Mbps
Keccak	fast	very very fast	not very good

- Software performance is based on my laptop.

Maximizing the constant factor

H	Software perf.	Hardware perf.	Score ($= \mathrm{SW}^2/\mathrm{HW}$)
SHA256	450 Mbps	1250 Mbps	160 Mbps
Blowfish	800 Mbps	1000 Mbps	640 Mbps
AES-128	1200 Mbps	40000 Mbps	36 Mbps
Salsa20/8	2000 Mbps	2000 Mbps	2000 Mbps
Keccak	fast	very very fast	not very good

- Software performance is based on my laptop.
- Hardware performance is based on a 130 nm CMOS process.

Maximizing the constant factor

H	Software perf.	Hardware perf.	Score $(= \mathrm{SW}^2/\mathrm{HW})$
SHA256	450 Mbps	1250 Mbps	160 Mbps
Blowfish	800 Mbps	1000 Mbps	640 Mbps
AES-128	1200 Mbps	40000 Mbps	36 Mbps
Salsa20/8	2000 Mbps	2000 Mbps	2000 Mbps
Keccak	fast	very very fast	not very good

- Software performance is based on my laptop.
- Hardware performance is based on a 130 nm CMOS process.
    - I think the relative ordering of functions will still be the same with more modern hardware.

Maximizing the constant factor

H	Software perf.	Hardware perf.	Score ($= \mathrm{SW}^2/\mathrm{HW}$)
SHA256	450 Mbps	1250 Mbps	160 Mbps
Blowfish	800 Mbps	1000 Mbps	640 Mbps
AES-128	1200 Mbps	40000 Mbps	36 Mbps
Salsa20/8	2000 Mbps	2000 Mbps	2000 Mbps
Keccak	fast	very very fast	not very good

- Software performance is based on my laptop.
- Hardware performance is based on a 130 nm CMOS process.
  - I think the relative ordering of functions will still be the same with more modern hardware.
  - If there's a cryptographer in the audience working for a semiconductor company, I'd love to have more modern data...

Maximizing the constant factor

- Using a cryptographic primitive H directly turns out to yield poor performance in software due to CPU architecture issues.

Maximizing the constant factor

- Using a cryptographic primitive H directly turns out to yield poor performance in software due to CPU architecture issues.
- Accessing a random 64-byte value from a 1 GB block of RAM takes about as long as computing Salsa20/8.

Maximizing the constant factor

- Using a cryptographic primitive H directly turns out to yield poor performance in software due to CPU architecture issues.
- Accessing a random 64-byte value from a 1 GB block of RAM takes about as long as computing Salsa20/8.
  - Every random access causes a TLB miss.

Maximizing the constant factor

- Using a cryptographic primitive H directly turns out to yield poor performance in software due to CPU architecture issues.
- Accessing a random 64-byte value from a 1 GB block of RAM takes about as long as computing Salsa20/8.
  - Every random access causes a TLB miss.
- To work around this, scrypt accesses data in 1 kB blocks.

Maximizing the constant factor

- Using a cryptographic primitive H directly turns out to yield poor performance in software due to CPU architecture issues.
- Accessing a random 64-byte value from a 1 GB block of RAM takes about as long as computing Salsa20/8.
  - Every random access causes a TLB miss.
- To work around this, scrypt accesses data in 1 kB blocks.
  - We compute $Y_i = H(Y_{i-1} \oplus X_i)$.

Maximizing the constant factor

- Using a cryptographic primitive H directly turns out to yield poor performance in software due to CPU architecture issues.
- Accessing a random 64-byte value from a 1 GB block of RAM takes about as long as computing Salsa20/8.
  - Every random access causes a TLB miss.
- To work around this, scrypt accesses data in 1 kB blocks.
  - We compute $Y_i = H(Y_{i-1} \oplus X_i)$.
  - Output is $Y_0, Y_2, \ldots, Y_{14}, Y_1, Y_3, \ldots, Y_{15}$.

Maximizing the constant factor

- Using a cryptographic primitive H directly turns out to yield poor performance in software due to CPU architecture issues.
- Accessing a random 64-byte value from a 1 GB block of RAM takes about as long as computing Salsa20/8.
  - Every random access causes a TLB miss.
- To work around this, scrypt accesses data in 1 kB blocks.
  - We compute $Y_i = H(Y_{i-1} \oplus X_i)$.
  - Output is $Y_0, Y_2, \ldots, Y_{14}, Y_1, Y_3, \ldots, Y_{15}$.
  - The "chained" computation ensures that there is no opportunity for parallelism.

Maximizing the constant factor

- Using a cryptographic primitive H directly turns out to yield poor performance in software due to CPU architecture issues.
- Accessing a random 64-byte value from a 1 GB block of RAM takes about as long as computing Salsa20/8.
  - Every random access causes a TLB miss.
- To work around this, scrypt accesses data in 1 kB blocks.
  - We compute $Y_i = H(Y_{i-1} \oplus X_i)$.
  - Output is $Y_0, Y_2, \ldots, Y_{14}, Y_1, Y_3, \ldots, Y_{15}$.
  - The "chained" computation ensures that there is no opportunity for parallelism.
  - The permuting of outputs avoids any "pipelining" of multiple hash computations.

Maximizing the constant factor

- Using a cryptographic primitive H directly turns out to yield poor performance in software due to CPU architecture issues.
- Accessing a random 64-byte value from a 1 GB block of RAM takes about as long as computing Salsa20/8.
  - Every random access causes a TLB miss.
- To work around this, scrypt accesses data in 1 kB blocks.
  - We compute $Y_i = H(Y_{i-1} \oplus X_i)$.
  - Output is $Y_0, Y_2, \ldots, Y_{14}, Y_1, Y_3, \ldots, Y_{15}$.
  - The "chained" computation ensures that there is no opportunity for parallelism.
  - The permuting of outputs avoids any "pipelining" of multiple hash computations.
- I believe this improves software performance more than it improves hardware performance, but I have no proof.

Estimating hardware brute force attack costs

- It's hard to get accurate information about how much it costs to build password-cracking machines.

Estimating hardware brute force attack costs

- It's hard to get accurate information about how much it costs to build password-cracking machines.
  - Oddly enough, the NSA doesn't publish this data.

Estimating hardware brute force attack costs

- It's hard to get accurate information about how much it costs to build password-cracking machines.
  - Oddly enough, the NSA doesn't publish this data.
- The best we can do for most KDFs is to count cryptographic operations and assume that they are responsible for most of the time and die area.

Estimating hardware brute force attack costs

- It's hard to get accurate information about how much it costs to build password-cracking machines.
  - Oddly enough, the NSA doesn't publish this data.
- The best we can do for most KDFs is to count cryptographic operations and assume that they are responsible for most of the time and die area.
  - This is probably a fairly accurate approximation, since key derivation functions only have a very small amount of non-cryptographic computations.

Estimating hardware brute force attack costs

- It's hard to get accurate information about how much it costs to build password-cracking machines.
  - Oddly enough, the NSA doesn't publish this data.
- The best we can do for most KDFs is to count cryptographic operations and assume that they are responsible for most of the time and die area.
  - This is probably a fairly accurate approximation, since key derivation functions only have a very small amount of non-cryptographic computations.
- For scrypt we also need to look at the die area required for storage.

Estimating hardware brute force attack costs

- Very approximate estimates of VLSI area and cost on a 130 nm process:
  - Each gate of random logic requires $\approx 5\ \mu\mathrm{m}^2$ of VLSI area.

- Very approximate estimates of VLSI area and cost on a 130 nm process:
  - Each gate of random logic requires $\approx 5\ \mu\mathrm{m}^2$ of VLSI area.
  - Each bit of SRAM requires $\approx 2.5\ \mu\mathrm{m}^2$ of VLSI area.

Estimating hardware brute force attack costs

- Very approximate estimates of VLSI area and cost on a 130 nm process:
  - Each gate of random logic requires $\approx 5\ \mu\mathrm{m}^2$ of VLSI area.
  - Each bit of SRAM requires $\approx 2.5\ \mu\mathrm{m}^2$ of VLSI area.
  - Each bit of DRAM requires $\approx 0.1\ \mu\mathrm{m}^2$ of VLSI area.

Estimating hardware brute force attack costs

- Very approximate estimates of VLSI area and cost on a 130 nm process:
  - Each gate of random logic requires $\approx 5\ \mu\mathrm{m}^2$ of VLSI area.
  - Each bit of SRAM requires $\approx 2.5\ \mu\mathrm{m}^2$ of VLSI area.
  - Each bit of DRAM requires $\approx 0.1\ \mu\mathrm{m}^2$ of VLSI area.
  - VLSI circuits cost $\approx 0.1\$/\mathrm{mm}^2$.

Estimating hardware brute force attack costs

- Very approximate estimates of VLSI area and cost on a 130 nm process:
  - Each gate of random logic requires $\approx$ 5 $\mu\mathrm{m}^2$ of VLSI area.
  - Each bit of SRAM requires $\approx$ 2.5 $\mu\mathrm{m}^2$ of VLSI area.
  - Each bit of DRAM requires $\approx$ 0.1 $\mu\mathrm{m}^2$ of VLSI area.
  - VLSI circuits cost $\approx$ 0.1$\$/\mathrm{mm}^2$.
- These values have a very wide error margin.

Estimating hardware brute force attack costs

- Very approximate estimates of VLSI area and cost on a 130 nm process:
  - Each gate of random logic requires $\approx 5\ \mu\mathrm{m}^2$ of VLSI area.
  - Each bit of SRAM requires $\approx 2.5\ \mu\mathrm{m}^2$ of VLSI area.
  - Each bit of DRAM requires $\approx 0.1\ \mu\mathrm{m}^2$ of VLSI area.
  - VLSI circuits cost $\approx 0.1\$/\mathrm{mm}^2$.
- These values have a very wide error margin.
  - Non-cryptographic parts of ASICs (e.g., I/O), chip packaging, boards, power supplies, and operating costs could increase password-cracking costs by a factor of 10.

Estimating hardware brute force attack costs

- Very approximate estimates of VLSI area and cost on a 130 nm process:
  - Each gate of random logic requires $\approx 5\ \mu\mathrm{m}^2$ of VLSI area.
  - Each bit of SRAM requires $\approx 2.5\ \mu\mathrm{m}^2$ of VLSI area.
  - Each bit of DRAM requires $\approx 0.1\ \mu\mathrm{m}^2$ of VLSI area.
  - VLSI circuits cost $\approx 0.1\$/\mathrm{mm}^2$.
- These values have a very wide error margin.
  - Non-cryptographic parts of ASICs (e.g., I/O), chip packaging, boards, power supplies, and operating costs could increase password-cracking costs by a factor of 10.
  - Improvements in semiconductor technology since 2002 could reduce password-cracking costs by a factor of 10 or more.

Estimating hardware brute force attack costs

- Very approximate estimates of VLSI area and cost on a 130 nm process:
  - Each gate of random logic requires $\approx 5\ \mu\mathrm{m}^2$ of VLSI area.
  - Each bit of SRAM requires $\approx 2.5\ \mu\mathrm{m}^2$ of VLSI area.
  - Each bit of DRAM requires $\approx 0.1\ \mu\mathrm{m}^2$ of VLSI area.
  - VLSI circuits cost $\approx 0.1\$/\mathrm{mm}^2$.
- These values have a very wide error margin.
  - Non-cryptographic parts of ASICs (e.g., I/O), chip packaging, boards, power supplies, and operating costs could increase password-cracking costs by a factor of 10.
  - Improvements in semiconductor technology since 2002 could reduce password-cracking costs by a factor of 10 or more.
  - Improved cryptographic circuits could reduce costs by a factor of 10.

Key derivation functions

- Non-parameterized KDFs:
  - DES CRYPT
  - MD5 CRYPT
  - MD5

Key derivation functions

- Non-parameterized KDFs:
  - DES CRYPT
  - MD5 CRYPT
  - MD5
- KDFs tuned for interactive logins ($t \leq 100\ \mathrm{ms}$):
  - PBKDF2-HMAC-SHA256, $c = 86000$
  - bcrypt, $cost = 11$
  - scrypt, $N = 2^{14}, r = 8, p = 1$

Key derivation functions

- Non-parameterized KDFs:
  - DES CRYPT
  - MD5 CRYPT
  - MD5
- KDFs tuned for interactive logins ($t \leq 100\ \mathrm{ms}$):
  - PBKDF2-HMAC-SHA256, $c = 86000$
  - bcrypt, $cost = 11$
  - scrypt, $N = 2^{14}, r = 8, p = 1$
- KDFs tuned for file encryption ($t \leq 5\ \mathrm{s}$):
  - PBKDF2-HMAC-SHA256, $c = 4300000$
  - bcrypt, $cost = 16$
  - scrypt, $N = 2^{20}, r = 8, p = 1$

Key derivation functions

- Non-parameterized KDFs:
  - DES CRYPT
  - MD5 CRYPT
  - MD5
- KDFs tuned for interactive logins ($t \leq 100\ \mathrm{ms}$):
  - PBKDF2-HMAC-SHA256, $c = 86000$
  - bcrypt, $cost = 11$
  - scrypt, $N = 2^{14}, r = 8, p = 1$
- KDFs tuned for file encryption ($t \leq 5\ \mathrm{s}$):
  - PBKDF2-HMAC-SHA256, $c = 4300000$
  - bcrypt, $cost = 16$
  - scrypt, $N = 2^{20}, r = 8, p = 1$
- Running time based on a 2.5 GHz Core 2 (aka. my laptop).

Passwords

- 6 lower-case letters; e.g., "sfgroy".

Passwords

- 6 lower-case letters; e.g., “sfgroy”.
- 8 lower-case letters; e.g., “ksuvnwyf”.

Passwords

- 6 lower-case letters; e.g., "`sfgroy`".
- 8 lower-case letters; e.g., "`ksuvnwyf`".
- 8 characters selected from the 95 printable 7-bit ASCII characters; e.g., "`6,uh3y[a`".

Passwords

- 6 lower-case letters; e.g., "`sfgroy`".
- 8 lower-case letters; e.g., "`ksuvnwyf`".
- 8 characters selected from the 95 printable 7-bit ASCII characters; e.g., "`6,uh3y[a`".
- 10 characters selected from the 95 printable 7-bit ASCII characters; e.g., "`H.*W8Jz&r3`".

Passwords

- 6 lower-case letters; e.g., "sfgroy".
- 8 lower-case letters; e.g., "ksuvnwyf".
- 8 characters selected from the 95 printable 7-bit ASCII characters; e.g., "6,uh3y[a".
- 10 characters selected from the 95 printable 7-bit ASCII characters; e.g., "H.*W8Jz&r3".
- A 40-character string of text; e.g., "This is a 40-character string of English".
  - Entropy estimated according to formula from NIST: 1st character has 4 bits of entropy; 2nd–8th characters have 2 bits of entropy each; 9th–20th characters have 1.5 bits of entropy each; 21st and later characters have 1 bit of entropy each.

Passwords

- 6 lower-case letters; e.g., "`sfgroy`".
- 8 lower-case letters; e.g., "`ksuvnwyf`".
- 8 characters selected from the 95 printable 7-bit ASCII characters; e.g., "`6,uh3y[a`".
- 10 characters selected from the 95 printable 7-bit ASCII characters; e.g., "`H.*W8Jz&r3`".
- A 40-character string of text; e.g., "`This is a 40-character string of English`".
    - Entropy estimated according to formula from NIST: 1st character has 4 bits of entropy; 2nd–8th characters have 2 bits of entropy each; 9th–20th characters have 1.5 bits of entropy each; 21st and later characters have 1 bit of entropy each.
    - This formula is not very good, but it's the best I have available...

Estimated brute force attack costs

Estimated cost of hardware to crack a password in 1 year.

KDF	6 letters	8 letters	8 chars	10 chars	40-char text
DES CRYPT	$< \$1$	$< \$1$	$< \$1$	$< \$1$	$< \$1$
MD5	$< \$1$	$< \$1$	$< \$1$	\$1.1k	\$1
MD5 CRYPT	$< \$1$	$< \$1$	\$130	\$1.1M	\$1.4k
PBKDF2 (100 ms)	$< \$1$	$< \$1$	\$18k	\$160M	\$200k
bcrypt (95 ms)	$< \$1$	\$4	\$130k	\$1.2B	\$1.5M
scrypt (64 ms)	$< \$1$	\$150	\$4.8M	\$43B	\$52M
PBKDF2 (5.0 s)	$< \$1$	\$29	\$920k	\$8.3B	\$10M
bcrypt (3.0 s)	$< \$1$	\$130	\$4.3M	\$39B	\$47M
scrypt (3.8 s)	\$900	\$610k	\$19B	\$175T	\$210B

KDF brute force attack costs

- When used for interactive logins, scrypt is . . .
  - $\approx 2^5$ times more expensive to attack than bcrypt,

KDF brute force attack costs

- When used for interactive logins, scrypt is . . .
  - $\approx 2^5$ times more expensive to attack than bcrypt,
  - $\approx 2^8$ times more expensive to attack than PBKDF2,

KDF brute force attack costs

- When used for interactive logins, scrypt is ...
  - $\approx 2^5$ times more expensive to attack than bcrypt,
  - $\approx 2^8$ times more expensive to attack than PBKDF2,
  - and $\approx 2^{15}$ times more expensive to attack than MD5 CRYPT.

KDF brute force attack costs

- When used for interactive logins, scrypt is . . .
  - $\approx 2^5$ times more expensive to attack than bcrypt,
  - $\approx 2^8$ times more expensive to attack than PBKDF2,
  - and $\approx 2^{15}$ times more expensive to attack than MD5 CRYPT.
- When used for file encryption, scrypt is . . .
  - $\approx 2^{12}$ times more expensive to attack than bcrypt,

KDF brute force attack costs

- When used for interactive logins, scrypt is ...
  - $\approx 2^5$ times more expensive to attack than bcrypt,
  - $\approx 2^8$ times more expensive to attack than PBKDF2,
  - and $\approx 2^{15}$ times more expensive to attack than MD5 CRYPT.
- When used for file encryption, scrypt is ...
  - $\approx 2^{12}$ times more expensive to attack than bcrypt,
  - $\approx 2^{15}$ times more expensive to attack than PBKDF2,

KDF brute force attack costs

- When used for interactive logins, scrypt is ...
  - $\approx 2^5$ times more expensive to attack than bcrypt,
  - $\approx 2^8$ times more expensive to attack than PBKDF2,
  - and $\approx 2^{15}$ times more expensive to attack than MD5 CRYPT.
- When used for file encryption, scrypt is ...
  - $\approx 2^{12}$ times more expensive to attack than bcrypt,
  - $\approx 2^{15}$ times more expensive to attack than PBKDF2,
  - and $\approx 2^{37}$ times more expensive to attack than MD5.

KDF brute force attack costs

- When used for interactive logins, scrypt is ...
  - $\approx 2^5$ times more expensive to attack than bcrypt,
  - $\approx 2^8$ times more expensive to attack than PBKDF2,
  - and $\approx 2^{15}$ times more expensive to attack than MD5 CRYPT.
- When used for file encryption, scrypt is ...
  - $\approx 2^{12}$ times more expensive to attack than bcrypt,
  - $\approx 2^{15}$ times more expensive to attack than PBKDF2,
  - and $\approx 2^{37}$ times more expensive to attack than MD5.
- `openssl enc` uses MD5 as a key derivation function.

KDF brute force attack costs

- When used for interactive logins, scrypt is ...
  - $\approx 2^5$ times more expensive to attack than bcrypt,
  - $\approx 2^8$ times more expensive to attack than PBKDF2,
  - and $\approx 2^{15}$ times more expensive to attack than MD5 CRYPT.
- When used for file encryption, scrypt is ...
  - $\approx 2^{12}$ times more expensive to attack than bcrypt,
  - $\approx 2^{15}$ times more expensive to attack than PBKDF2,
  - and $\approx 2^{37}$ times more expensive to attack than MD5.
- `openssl enc` uses MD5 as a key derivation function.
- OpenSSH uses MD5 as a key derivation function for passphrases on key files.

KDF brute force attack costs

- When used for interactive logins, scrypt is ...
  - $\approx 2^5$ times more expensive to attack than bcrypt,
  - $\approx 2^8$ times more expensive to attack than PBKDF2,
  - and $\approx 2^{15}$ times more expensive to attack than MD5 CRYPT.
- When used for file encryption, scrypt is ...
  - $\approx 2^{12}$ times more expensive to attack than bcrypt,
  - $\approx 2^{15}$ times more expensive to attack than PBKDF2,
  - and $\approx 2^{37}$ times more expensive to attack than MD5.
- `openssl enc` uses MD5 as a key derivation function.
- OpenSSH uses MD5 as a key derivation function for passphrases on key files.
  - Are you sure that your SSH keys are safe?

Availability

- More details at `http://www.tarsnap.com/scrypt/`.
  - Source code for scrypt.
  - A simple file encryption/decryption utility.
  - A 16-page paper.

Availability

- More details at `http://www.tarsnap.com/scrypt/`.
  - Source code for scrypt.
  - A simple file encryption/decryption utility.
  - A 16-page paper.
- Currently an IETF Internet-Draft.

Availability

- More details at `http://www.tarsnap.com/scrypt/`.
  - Source code for scrypt.
  - A simple file encryption/decryption utility.
  - A 16-page paper.
- Currently an IETF Internet-Draft.

Questions?