SipHash:

a fast short-input PRF

D. J. Bernstein,
 University of Illinois at Chicago &
 Technische Universiteit Eindhoven

Joint work with:

Jean-Philippe Aumasson, Kudelski Security (NAGRA)

https://131002.net/siphash/

Advertisement: Competition coming soon for authenticated ciphers! Several motivations:

1. Optimize secret-key crypto for *short messages*.

2. Build a PRF/MAC that's secure, efficient, *simple*.

3. Application: authenticate Internet packets.

4. Application: defend against hash flooding.

 5. Analyze security of other hash-flooding defenses.
 Followup work with Martin Boßlet pushes this much further.

Today's focus: hash flooding

July 1998 article

"Designing and attacking port scan detection tools" by Solar Designer (Alexander Peslyak) in Phrack Magazine:

"In scanlogd, I'm using a hash table to lookup source addresses. This works very well for the typical case ... average lookup time is better than that of a binary search....

However, an attacker can choose her addresses (most likely spoofed) to cause hash collisions, effectively replacing the hash table lookup with a linear search. Depending on how many entries we keep, this might make scanlogd not be able to pick new packets up in time. . . . I've solved this problem by limiting the number of hash collisions, and discarding the oldest entry with the same hash value when the limit is reached.

This is acceptable for port scans (remember, we can't detect all scans anyway), but might not be acceptable for detecting other attacks. ... It is probably worth mentioning that similar issues also apply to things like operating system kernels. For example, hash tables are widely used there for looking up active connections, listening ports, etc. There're usually other limits which make these not really dangerous though, but more research might be needed." December 1999, Bernstein, dnscache software:

if (++loop > 100) return 0; /* to protect against hash flooding */ Discarding cache entries trivially maintains performance if attacker floods hash table.

But what about hash tables in general-purpose programming languages and libraries? Can't throw entries away! 2003 USENIX Security Symposium, Crosby–Wallach, "Denial of service via algorithmic complexity attacks":

"We present a new class of low-bandwidth denial of service attacks ... if each element hashes to the same bucket, the hash table will also degenerate to a linked list."

Attack examples:

Perl programming language, Squid web cache, etc. No attack on dnscache. 2011 (28C3), Klink–Wälde,

"Efficient denial of service attacks on web application platforms"; oCERT advisory 2011–003:

No attack on dnscache, fixed Perl, fixed Squid; but still problems in Java, JRuby, PHP 4, PHP 5, Python 2, Python 3, Rubinius, Ruby, Apache Geronimo, Apache Tomcat, Oracle Glassfish, Jetty, Plone, Rack, V8 Javascript Engine.

Defending against hash flooding

My favorite solution: switch from hash tables to crit-bit trees. Guaranteed high speed + extra lookup features such as "find next entry after x."

Defending against hash flooding

My favorite solution: switch from hash tables to crit-bit trees. Guaranteed high speed + extra lookup features such as "find next entry after x."

But hash tables are perceived as being smaller, faster, *simpler* than other data structures. Can we protect hash tables?

Classic hash table: ℓ separate linked lists for some $\ell \in \{1, 2, 4, 8, 16, \ldots\}$. Store string s in list #iwhere $i = H(s) \mod \ell$. With *n* entries in table, expect $\approx n/\ell$ entries in each linked list. Choose $\ell \approx n$:

expect very short linked lists, so very fast list operations.

(What if n becomes too big? Rehash: replace ℓ by 2ℓ .) Basic hash flooding: attacker provides strings s_1, \ldots, s_n with $H(s_1) \mod \ell =$ $\cdots = H(s_n) \mod \ell$.

Then all strings are stored in the same linked list. Linked list becomes very slow. Basic hash flooding: attacker provides strings s_1, \ldots, s_n with $H(s_1) \mod \ell =$ $\cdots = H(s_n) \mod \ell$.

Then all strings are stored in the same linked list. Linked list becomes very slow.

Solution: Replace linked list by a safe tree structure, at least if list is big. Basic hash flooding: attacker provides strings s_1, \ldots, s_n with $H(s_1) \mod \ell =$ $\cdots = H(s_n) \mod \ell$.

Then all strings are stored in the same linked list. Linked list becomes very slow.

Solution: Replace linked list by a safe tree structure, at least if list is big.

But implementors are unhappy: this solution throws away the simplicity of hash tables. Non-solution: Use SHA-3 for *H*. SHA-3 is collision-resistant! Non-solution: Use SHA-3 for H. SHA-3 is collision-resistant! Why this is bad: $H(s) \mod \ell$ is not collision-resistant.

 ℓ is small: e.g., $\ell = 2^{20}$. No matter how strong H is, attacker can easily compute $H(s) \mod 2^{20}$ for many sto find multicollisions. 1977, Carter–Wegman, "Universal classes of hash functions": "This paper gives an *input independent* average linear time algorithm for storage and retrieval on keys. The algorithm makes a random choice of hash function from a suitable class of hash functions."

2003 Crosby–Wallach: About 6 cycles/byte on P2 for $H(m_1, m_2, \ldots, m_{12}) =$ $m_1k_1 + m_2k_2 + \cdots + m_{12}k_{12}$. k_1, k_2, \ldots, k_{12} : random, 20-bit. This is "provably secure"! We don't recommend this. The security guarantee assumes that randomness is *independent* of inputs. We don't recommend this. The security guarantee assumes that randomness is *independent* of inputs.

Advanced hash flooding: use, e.g., server timing to detect hash collisions; figure out the hash key; choose inputs accordingly.

2005 Crosby: Maybe trouble for any function with a short key, and for $m_1k_1 + m_2k_2 + \cdots$. Even worse: Some applications (e.g., any application that prints table without sorting) leak more information about *H*.

Some applications simply print $H(s) \mod \ell$, or even H(s).

Even worse: Some applications (e.g., any application that prints table without sorting) leak more information about *H*.

Some applications simply print $H(s) \mod \ell$, or even H(s).

We recommend choosing Has a strong PRF. \Rightarrow Seeing many H values is of no use in predicting others. Even worse: Some applications (e.g., any application that prints table without sorting) leak more information about *H*.

Some applications simply print $H(s) \mod \ell$, or even H(s).

We recommend choosing Has a strong PRF. \Rightarrow Seeing many H values is of no use in predicting others.

Finding *n*-collision in $H(s) \mod \ell$ requires trying $\approx n\ell \approx n^2$ inputs. Damage is only $\sqrt{\text{communication}}$.

The importance of overhead

Crypto design, 1990s: Wow, MD5 is really fast; only about 5 cycles/byte. Let's use HMAC-MD5 as a PRF.

The importance of overhead

Crypto design, 1990s: Wow, MD5 is really fast; only about 5 cycles/byte. Let's use HMAC-MD5 as a PRF.

Crypto design, 2000s: Multipliers are even faster; can reach 1 or 2 cycles/byte. Poly1305-AES, UMAC-AES, et al.

The importance of overhead

Crypto design, 1990s: Wow, MD5 is really fast; only about 5 cycles/byte. Let's use HMAC-MD5 as a PRF.

Crypto design, 2000s: Multipliers are even faster; can reach 1 or 2 cycles/byte. Poly1305-AES, UMAC-AES, et al.

The hash-table perspective: These speed advertisements are only for long inputs, ignoring huge overheads!

SipRound and SipHash



SipHash-2-4 applied to 16 bytes.



Much more in paper:

- Specification: padding etc.
- Discussion of features.
- Statement of security goals.
- Design rationale and credits.
- Preliminary cryptanalysis.
- Benchmarks. e.g. Ivy Bridge: 1.65 cycles/byte + 27 cycles.

Positive SipHash reception: many third-party implementations; now used for hash tables in Ruby, Redis, Rust, OpenDNS, Perl 5.