Hash-flooding DoS reloaded: attacks and defenses

Jean-Philippe Aumasson, Kudelski Security (NAGRA)

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

Martin Boßlet, Ruby Core Team

Hash flooding begins?

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. . . .
Hash flooding begins?

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.
Hash-flooding DoS reloaded: attacks and defenses

Jean-Philippe Aumasson, Kudelski Security (NAGRA)
D. J. Bernstein, University of Illinois at Chicago & Technische Universiteit Eindhoven
Martin Boßlet, Ruby Core Team

Hash flooding begins?

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.
Hash flooding begins?

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 up 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.
Hash flooding begins?
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.
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 other attacks. It's worth mentioning that similar issues also apply to operating system kernels, for example. There are usually other limits which make these not really dangerous, but more research might be needed.
Hash flooding begins?

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 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...
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.
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.”
However, an attacker can choose her addresses (most likely spoofed) to cause hash collisions, effectively replacing the table lookup with a linear search. Depending on how many entries we keep, this might make scanlogd not be able to pick up new packets 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."

Review of classic hash tables
Choose $2, 4, 8, 16, \ldots$

Hash table: separate linked lists. Store string $s$ in list $i$ where $i = H(s) \mod n$.

With $n$ entries in table, expect $\approx n$ entries in each linked list.

Choose $n$ so expect very short linked lists, so very fast list operations.

(What if $n$ becomes too big? Rehash: replace by $2n$.)

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."

Review of classic hash tables
Choose $2, 4, 8, 16, \ldots$

Hash table: separate linked lists. Store string $s$ in list $i$ where $i = H(s) \mod n$.

With $n$ entries in table, expect $\approx n$ entries in each linked list.

Choose $n$ so expect very short linked lists, so very fast list operations.

(What if $n$ becomes too big? Rehash: replace by $2n$.)

However, an attacker can choose her addresses (most likely spoofed) to cause hash collisions, effectively replacing the table lookup with a linear search. Depending on how many entries we keep, this might make scanlogd not be able to pick up new packets 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."

Review of classic hash tables
Choose $2, 4, 8, 16, \ldots$

Hash table: separate linked lists. Store string $s$ in list $i$ where $i = H(s) \mod n$.

With $n$ entries in table, expect $\approx n$ entries in each linked list.

Choose $n$ so expect very short linked lists, so very fast list operations.

(What if $n$ becomes too big? Rehash: replace by $2n$.)
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."

Review of classic hash tables
Choose \( l \in \{1, 2, 4, 8, 16, \ldots\} \).
Hash table: \( l \) separate linked lists.
Store string \( s \) in list \( i \) where \( i = H(s) \mod l \).
With \( n \) entries in the table, expect \( \approx n/l \) entries in each linked list.
Choose \( l \approx n \):
expect very short linked lists, so very fast list operations.
(What if \( n \) becomes too big?
Rehash: replace \( l \) by 2 \( l \).)
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.

Review of classic hash tables

Choose \( l \in \{1, 2, 4, 8, 16, \ldots \} \).

Hash table: \( l \) separate linked lists. Store string \( s \) in list \( \#i \) where \( i = H(s) \mod l \).

With \( n \) entries in table, expect \( \approx n/l \) entries in each linked list.

Choose \( l \approx n \): expect very short linked lists, so very fast list operations.

(What if \( n \) becomes too big? Rehash: replace \( l \) by \( 2l \).)
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.”

Review of classic hash tables
Choose $\ell \in \{1, 2, 4, 8, 16, \ldots\}$.

Hash table: $\ell$ separate linked lists.
Store string $s$ in list $\#i$ where $i = H(s) \mod \ell$.

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 $\ell$ by $2\ell$.)
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 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.

---

Review of classic hash tables

Choose \( \{1, 2, 4, 8, 16, \ldots\} \). Hash table: \( l \) separate linked lists. Store string \( s \) in list \( \# i \), where \( i = H(s) \mod l \).

With \( n \) entries in table, expect \( \approx n/l \) entries in each linked list. Choose \( l \approx n \). With \( n \) entries in table, expect very short linked lists, so very fast list operations.

(What if \( n \) becomes too big? Rehash: replace \( l \) by \( 2l \).)

\( H(s) \) = first byte of \( s \); \( \approx = 256 \).

---

\( \text{e.g. strings one, two, \ldots, ten} \):

\( t \rightarrow \text{two, six, eight} \)

\( q \rightarrow \text{one, five} \)

\( p \rightarrow \text{six, ten} \)

\( o \rightarrow \text{one, nine} \)

\( n \rightarrow \text{one, six} \)

\( i \rightarrow \text{eight, four} \)

\( e \rightarrow \text{eight, one} \)

\( \text{t, q, p, o, n, i, e} \)
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
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.

---

Review of classic hash tables

Choose $l \in \{1, 2, 4, 8, 16, \ldots\}$.  

Hash table: $l$ separate linked lists.

Store string $s$ in list $\#i$
where $i = H(s) \mod l$.

With $n$ entries in table,
expect $\approx n/l$ entries
in each linked list.

Choose $l \approx n$:
expect very short linked lists,
so very fast list operations.

(What if $n$ becomes too big?
Rehash: replace $l$ by $2l$.)

---

e.g. strings one, two, : : : , ten;
$H(s) =$ first byte of $s$; $l = 256$:

: :
e→eight
f→four→five
: :
n→nine
o→one
p
q
r
s→six→seven
t→two→three→...
Review of classic hash tables

Choose $\ell \in \{1, 2, 4, 8, 16, \ldots\}$.

Hash table: $\ell$ separate linked lists.

Store string $s$ in list $\#i$ where $i = H(s) \mod \ell$.

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 $\ell$ by $2\ell$.)

e.g. strings one, two, ..., ten;
$H(s) =$ first byte of $s$; $\ell = 2$

: e→eight
  f→four→five
  : n→nine
  o→one
  p
  q
  r
  s→six→seven
  t→two→three→ten
  :
Review of classic hash tables

Choose $l \in \{1, 2, 4, 8, 16, \ldots\}$.  

Hash table: $l$ separate linked lists.
Store string $s$ in list $\#i$  
where $i = H(s) \mod l$.  

With $n$ entries in table,  
expect $\approx n/l$ entries  
in each linked list.
Choose $l \approx n$:  
expect very short linked lists,  
so very fast list operations.  

(What if $n$ becomes too big?  
Rehash: replace $l$ by $2l$.)

e.g. strings one, two, \ldots, ten;  
$H(s) =$ first byte of $s$; $l = 256$:  
  \begin{align*}  
e &\rightarrow \text{eight}  
f &\rightarrow \text{four} \rightarrow \text{five}  
n &\rightarrow \text{nine}  
o &\rightarrow \text{one}  
p &  
q &  
r &  
s &\rightarrow \text{six} \rightarrow \text{seven}  
t &\rightarrow \text{two} \rightarrow \text{three} \rightarrow \text{ten}  
\end{align*}
Review of classic hash tables

Choose $\ell \in \{1, 2, 4, 8, 16, \ldots\}$.

Table: $\ell$ separate linked lists.

String $s$ in list #\(i\)

\[ i = H(s) \mod \ell. \]

With $n$ entries in table, 

\( \approx n/\ell \) entries 

linked list.

\( \ell \approx n: \)

very short linked lists, 

fast list operations.

If $n$ becomes too big? 

replace $\ell$ by $2\ell$.) 

\[ e \rightarrow \text{eight} \]

\[ f \rightarrow \text{four} \rightarrow \text{five} \]

\[ n \rightarrow \text{nine} \]

\[ o \rightarrow \text{one} \]

\[ p \]

\[ q \]

\[ r \]

\[ s \rightarrow \text{six} \rightarrow \text{seven} \]

\[ t \rightarrow \text{two} \rightarrow \text{three} \rightarrow \text{ten} \]

\[ \vdots \]

\[ H(s) = \text{first byte of } s; \ell = 256: \]

Typical strings often start with t.

Very long t list; very slow.

In some applications, most strings

start with the same letter.
Review of classic hash tables

Choose \(2 \in \{1, 2, 4, 8, 16, \ldots\}\).

Hash table: separate linked lists.

Store string \(s\) in list \(i\) where \(i = H(s) \mod l\).

With \(n\) entries in table, expect \(n/l\) entries in each linked list.

Choose \(l\): expect very short linked lists, so very fast list operations.

(What if \(n\) becomes too big? Rehash: replace \(l\) by \(2l\).)

\[H(s) = \text{first byte of } s; \quad l = 256:\]

\[e \rightarrow \text{eight} \quad f \rightarrow \text{four} \rightarrow \text{five} \quad n \rightarrow \text{nine} \quad o \rightarrow \text{one} \quad p \quad q \quad r \quad s \rightarrow \text{six} \rightarrow \text{seven} \quad t \rightarrow \text{two} \rightarrow \text{three} \rightarrow \text{ten} \]

\(H(s) = \text{first byte of } s\) is not a good hash function!

Typical strings often start with \(t\).

Very long \(t\) list; very slow.

In some applications, most strings start with the same letter.
Choose \( 2 \in \{1, 2, 4, 8, 16, \ldots\} \).

Hash table: separate linked lists.

Store string \( s \) in list \( i \) where \( i = H(s) \mod 2^k \).

With \( n \) entries in table, expect \( n=2^k \) entries in each linked list.

Choose \( n \): expect very short linked lists, so very fast list operations.

(What if \( n \) becomes too big? Rehash: replace \( 2^k \) by \( 2^{k+1} \).)

\[ H(s) = \text{first byte of } s; \quad \ell = 256: \]

\[ H(s) \] is not a good hash function!

Typical strings often start with \( t \).

Very long \( t \) list; very slow.

In some applications, most strings start with the same letter.
e.g. strings one, two, \ldots, ten; $H(s) = \text{first byte of } s$; $\ell = 256$:

\begin{align*}
e &\rightarrow \text{eight} \\
f &\rightarrow \text{four} \rightarrow \text{five} \\
n &\rightarrow \text{nine} \\
o &\rightarrow \text{one} \\
p \\
q \\
r \\
s &\rightarrow \text{six} \rightarrow \text{seven} \\
t &\rightarrow \text{two} \rightarrow \text{three} \rightarrow \text{ten} \\
\end{align*}

$H(s)$ is not a good hash function!

Typical strings often start with $t$. Very long $t$ list; very slow.

In some applications, \textit{most} strings start with the same letter.
e.g. strings one, two, ..., ten;

\[ H(s) = \text{first byte of } s \; ; \; \ell = 256: \]

\[
\begin{align*}
\text{e} & \rightarrow \text{eight} \\
\text{f} & \rightarrow \text{four} \rightarrow \text{five} \\
\text{n} & \rightarrow \text{nine} \\
\text{o} & \rightarrow \text{one} \\
\text{p} & \\
\text{q} & \\
\text{r} & \\
\text{s} & \rightarrow \text{six} \rightarrow \text{seven} \\
\text{t} & \rightarrow \text{two} \rightarrow \text{three} \rightarrow \text{ten} \\
\end{align*}
\]

\[ H(s) = \text{first byte of } s \]

is not a good hash function!

Typical strings often start with \( t \).

Very long \( t \) list; very slow.

In some applications, \textit{most} strings start with the same letter.

So we use fast hash functions that look at the whole string \( s \).

60 years of programmers exploring hash functions for hash tables \( \Rightarrow \) good speed for typical strings.
e.g. strings one, two, ..., ten; $H(s) = \text{first byte of } s$; $l = 256$:

: 

$H(s) = \text{first byte of } s$ is not a good hash function!

Typical strings often start with t.
Very long t list; very slow.

In some applications, most strings start with the same letter.

So we use fast hash functions that look at the whole string $s$.

60 years of programmers exploring hash functions for hash tables
⇒ good speed for typical strings.

What if the strings aren’t typical?
H(s) = first byte of s
is not a good hash function!

Typical strings often start with t.
Very long t list; very slow.

In some applications, most strings start with the same letter.

So we use fast hash functions that look at the whole string s.

60 years of programmers exploring hash functions for hash tables
⇒ good speed for typical strings.

What if the strings aren’t typical?

Hashing malicious strings
Attacker provides strings s₁, …, sₙ with H(s₁) mod ℓ = H(sₙ) mod ℓ.

Then all strings are stored in the same linked list; linked list becomes very slow.
$H(s) =$ first byte of $s$
is not a good hash function!

Typical strings often start with $t$.
Very long $t$ list; very slow.

In some applications, *most* strings start with the same letter.

So we use fast hash functions that look at the whole string $s$.

60 years of programmers exploring hash functions for hash tables
$\Rightarrow$ good speed for typical strings.

What if the strings aren't typical?

Hashing malicious strings

Attacker provides strings $s_1, \ldots, s_n$ with $H(s_1) \mod l = \cdots = H(s_n) \mod l$.

Then all strings are stored in the same linked list;
linked list becomes very slow.
$H(s) = \text{first byte of } s$

is not a good hash function!

Typical strings often start with $t$. 
Very long $t$ list; very slow.

In some applications, most strings start with the same letter.

So we use fast hash functions that look at the whole string $s$.

60 years of programmers exploring hash functions for hash tables \[ \Rightarrow \text{good speed for typical strings.} \]

What if the strings aren’t typical?

Hashing malicious strings

Attacker provides strings $s_1, \ldots, s_n$ with $H(s_1) \mod 2^8 = \cdots = H(s_n) \mod 2^8$.

Then all strings are stored in the same linked list; linked list becomes very slow.
$H(s) =$ first byte of $s$
is not a good hash function!

Typical strings often start with $t$.
Very long $t$ list; very slow.

In some applications, most strings
start with the same letter.

So we use fast hash functions
that look at the whole string $s$.

60 years of programmers exploring
hash functions for hash tables
$\Rightarrow$ good speed for typical strings.

What if the strings aren’t typical?

Hashing malicious strings
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.
$H(s) =$ first byte of $s$
is not a good hash function!

Typical strings often start with t.
Very long t list; very slow.

In some applications, *most* strings
start with the same letter.

So we use fast hash functions
that look at the whole string $s$.

60 years of programmers exploring
hash functions for hash tables
$\Rightarrow$ good speed for typical strings.

What if the strings aren’t typical?

Hashing malicious strings
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.
$H(s) = \text{first byte of } s$

is not a good hash function!

Typical strings often start with $t$. Very long $t$ list; very slow.

In some applications, most strings start with the same letter.

So we use fast hash functions that look at the whole string $s$.

60 years of programmers exploring hash functions for hash tables ⇒ good speed for typical strings.

What if the strings aren’t typical?

---

**Hashing malicious strings**

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.
Hashing malicious strings

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.

December 1999, Bernstein, dnscache software (OpenDNS: >41000000000000 DNS requests from 50 million Internet users):

```
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 we do something?
Hashing malicious strings

Attacker provides strings $s_1, \ldots, s_n$ with $H(s_1) \mod l = \cdots = H(s_n) \mod l$.

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.

December 1999, Bernstein, dnscache software (OpenDNS: >41000000000000 DNS requests from 50 million Internet users):

```c
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!
Hashing malicious strings

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.

December 1999, Bernstein, dnscache software (OpenDNS: > 41000000000000 DNS requests from 50 million Internet users):

```c
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!
Hashing malicious strings

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.

December 1999, Bernstein, dnscache software (OpenDNS: >41000000000000 DNS requests from 50 million Internet users):

```c
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!
Hashing malicious strings

Attacker provides strings \(s_1, \ldots, s_n\) with \(H(s_1) \mod \ell = \cdots = H(s_n) \mod \ell\).

Strings are stored in the same linked list; the list becomes very slow.

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

Implementors are unhappy: this solution throws away the simplicity of hash tables.

December 1999, Bernstein, dnscache software (OpenDNS: >41000000000000 DNS requests from 50 million Internet users):

```c
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!

Bad solution: Use SHA-3 for \(H\). SHA-3 is collision-resistant!
Attacker provides strings $s_1; : : : ; s_n$ with $H(s_1) \mod \ell = \cdots = H(s_n) \mod \ell$.

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

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

But implementors are unhappy: this solution throws away the simplicity of hash tables.

December 1999, Bernstein, dnscache software (OpenDNS: >41000000000000 DNS requests from 50 million Internet users):

```c
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!

Bad solution: Use SHA-3 for $H$. SHA-3 is collision-resistant!
Hashing malicious strings

Attacker provides strings $s_1; : : : ; s_n$ with $H(s_1) \mod \lambda = \cdots = H(s_n) \mod \lambda$.

Then all strings are stored in the same linked list; the 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.

December 1999, Bernstein, dnscache software (OpenDNS: >41000000000000 DNS requests from 50 million Internet users):

```c
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!

Bad solution: Use SHA-3 for $H$. SHA-3 is collision-resistant!
December 1999, Bernstein, dnscache software (OpenDNS: >41000000000000 DNS requests from 50 million Internet users):

   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!

Bad solution:
Use SHA-3 for $H$. SHA-3 is collision-resistant!
December 1999, Bernstein, 
dnscache software (OpenDNS: 
>410000000000000 DNS requests 
from 50 million Internet users):

    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!

Bad solution:
Use SHA-3 for $H$.
SHA-3 is collision-resistant!

Two reasons this is bad:
1. It’s very slow.
2. It doesn’t solve the problem.
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!

Bad solution:
Use SHA-3 for $H$.
SHA-3 is collision-resistant!

Two reasons this is bad:
1. It’s very slow.
2. It doesn’t solve the problem.

$H(s) \mod \ell$

is not collision-resistant.

$\ell$ is small: e.g., $\ell = 2^{20}$.

No matter how strong $H$ is, attacker can easily compute $H(s) \mod 2^{20}$ for many $s$ to find multicollisions.
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!

Bad solution:
Use SHA-3 for $H$.
SHA-3 is collision-resistant!

Two reasons this is bad:
1. It’s very slow.
2. It doesn’t solve the problem.

$H(s) \mod \ell$
is not collision-resistant.

$\ell$ is small: e.g., $\ell = 2^{20}$.
No matter how strong $H$ is, attacker can easily compute $H(s) \mod 2^{20}$ for many $s$ to find multicollisions.

“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.
Bad solution:
Use SHA-3 for $H$.
SHA-3 is collision-resistant!

Two reasons this is bad:
1. It’s very slow.
2. It doesn’t solve the problem.

$H(s) \mod \ell$
is not collision-resistant.

$\ell$ is small: e.g., $\ell = 2^{20}$.

No matter how strong $H$ is, attacker can easily compute $H(s) \mod 2^{20}$ for many $s$ to find multicollisions.

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 degenerate to a linked list.”

Attack examples:
Perl programming language,
Squid web cache, etc.
Bad solution:
Use SHA-3 for $H$.
SHA-3 is collision-resistant!

Two reasons this is bad:
1. It’s very slow.
2. It doesn’t solve the problem.

$H(s) \mod \ell$
is not collision-resistant.

$\ell$ is small: e.g., $\ell = 2^{20}$.

No matter how strong $H$ is, attacker can easily compute $H(s) \mod 2^{20}$ for many $s$ to find multicollisions.

“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.
Bad solution:
Use SHA-3 for $H$.
SHA-3 is collision-resistant!

Two reasons this is bad:
1. It’s very slow.
2. It doesn’t solve the problem.

$H(s) \mod \ell$
is not collision-resistant.

$\ell$ is small: e.g., $\ell = 2^{20}$.
No matter how strong $H$ is, attacker can easily compute
$H(s) \mod 2^{20}$ for many $s$ to find multicollisions.

---


“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.
Bad solution:

Use SHA-3 for $H$.

SHA-3 is collision-resistant!

Two reasons this is bad:

1. It's very slow.
2. It doesn't solve the problem.

Let $\ell$ be collision-resistant.

Well: e.g., $\ell = 2^{20}$.

No matter how strong $H$ is, attacker can easily compute $H(s) \mod 2^{20}$ for many $s$ to find multicollisions.


“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.

Application response:
use secret key to randomize $H$.

. . . but is this secure?

2011 (28C3), Klink–Wälde, “Efficient denial of service attacks on web application platforms”:

oCERT advisory 2011–003.
Bad solution:
Use SHA-3 for $H$.

SHA-3 is collision-resistant!

Two reasons this is bad:
1. It's very slow.
2. It doesn't solve the problem.

$H(s) \mod 2^{20}$ is not collision-resistant.

No matter how strong $H$ is, attacker can easily compute $H(s) \mod 2^{20}$ for many $s$ to find multicollisions.


“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.

2011 (28C3), Klink–Wallach, “Efficient denial of service attacks on web application platforms”:

cCERT advisory 2011–003.

Application response:
use secret key to randomize $H$.

... but is this secure?
Bad solution:
Use SHA-3 for \( H \).
SHA-3 is collision-resistant!

Two reasons this is bad:
1. It's very slow.
2. It doesn't solve the problem.

\( H(\text{s}) \mod 2^{20} \) is not collision-resistant.
If \( s \) is small: e.g., \( 2^{20} \).
No matter how strong \( H \) is, attacker can easily compute
\( H(s) \mod 2^{20} \) for many \( s \) to find multicollisions.

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.

2011 (28C3), Klink–Wälde, “Efficient denial of service attacks on web application platforms”:

oCERT advisory 2011–003.

Application response:
use secret key to randomize \( H \).
... but is this secure?

“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.

2011 (28C3), Klink–Wälde, “Efficient denial of service attacks on web application platforms”:


oCERT advisory 2011–003.

Application response:
use secret key to randomize $H$.

. . . but is this secure?
hash-flooding DoS reloaded: anatomy of an attack
MurmurHash2

“used in code by Google, Microsoft, Yahoo, and many others”

CRuby, JRuby, Redis

http://code.google.com/p/smhasher/wiki/MurmurHash
MurmurHash3

“successor to MurmurHash2”

Oracle & OpenJDK, Rubinius
1. Theory
MurmurHash2, 64 bit CRuby
const uint64_t m = (0xc6a4a793 << 32) | 0x5bd1e995;
uint64_t h = seed ^ len;

while (len >= 8) {

    uint64_t k = *(uint64_t*)data;

    k *= m;
    k ^= k >> 24;
    k *= m;

    h *= m;
    h ^= k;

    data += 8;
    len -= 8;
}
const uint64_t m = (0xc6a4a793 << 32) | 0x5bd1e995;
uint64_t h = seed ^ len;

while (len >= 8) {

    uint64_t k = *(uint64_t*)data;
    k *= m;
    k ^= k >> 24;
    k *= m;

    h *= m;
    h ^= k;

    data += 8;
    len -= 8;
}
block processing independent of seed
/* finalization */

switch (len) {

    case 6: h ^= data[5] << 40;
    case 2: h ^= data[1] << 8;
    case 1: h ^= data[0];
    h *= m;

};

...
8-byte-aligned data $\Rightarrow$ skip finalization
differential cryptanalysis
introduce a difference in the state $h$ via input $k$

cancel it again with a second well-chosen difference
const uint64_t m = (0xc6a4a793 << 32) | 0x5bd1e995;
uint64_t h = seed ^ len;

while (len >= 8) { /* first block */
    uint64_t k = *(uint64_t*)data;
    k *= m;  /* inject difference D1 */
    k ^= k >> 24;
    k *= m;

    h *= m;
    h ^= k;

    data += 8;
    len -= 8;
}
const uint64_t m = (0xc6a4a793 << 32) | 0x5bd1e995;
uint64_t h = seed ^ len;

while (len >= 8) {  /* first block */
    uint64_t k = *(uint64_t*)data;

    k *= m;  /* inject difference D1 */
    k ^= k >> 24;
    k *= m;  /* diff in k: 0x8000000000000000 */

    h *= m;
    h ^= k;

    data += 8;
    len -= 8;
}
const uint64_t m = (0xc6a4a793 << 32) | 0x5bd1e995;
uint64_t h = seed ^ len;

while (len >= 8) {  /* first block */
    uint64_t k = *(uint64_t*)data;
    k *= m;           /* inject difference D1 */
    k ^= k >> 24;
    k *= m;           /* diff in k: 0x8000000000000000 */
    h *= m;
    h ^= k;           /* diff in h: 0x8000000000000000 */
    data += 8;
    len -= 8;
}
const uint64_t m = (0xc6a4a793 << 32) | 0x5bd1e995;
uint64_t h = seed ^ len;

while (len >= 8) {  /* second block */
    uint64_t k = *(uint64_t*)data;

    k *= m;            /* inject difference D2 */
    k ^= k >> 24;
    k *= m;            /* diff in k: 0x8000000000000000 */

    h *= m;
    h ^= k;

    data += 8;
    len -= 8;
}
const uint64_t m = (0xc6a4a793 << 32) | 0x5bd1e995;
uint64_t h = seed ^ len;

while (len >= 8) { /* second block */

  uint64_t k = *(uint64_t*)data;

  k *= m; /* inject difference D2 */
  k ^= k >> 24;
  k *= m; /* diff in k: 0x8000000000000000 */

  h *= m; /* diff in h still: 0x8000000000000000 */
  h ^= k;

  data += 8;
  len -= 8;
}
const uint64_t m = (0xc6a4a793 << 32) | 0x5bd1e995;
uint64_t h = seed ^ len;

while (len >= 8) { /* second block */

    uint64_t k = *(uint64_t*)data;

    k *= m;           /* inject difference D2 */
    k ^= k >> 24;     /* diff in k: 0x80000000000000000000000000000000 */
    k *= m;           /* diff in h still: 0x80000000000000000000000000000000 */

    h *= m;           /* COLLISION !!! */
    h ^= k;
    data += 8;
    len -= 8;
}
chain collisions $\Rightarrow$ multicollisions

$16n$ bytes $\Rightarrow 2^n$ colliding inputs
multicollision works for any seed

⇒ “universal” multicollisions
same principle
slightly more complicated for
MurmurHash3
consequence
systems using MurmurHash2/3 remain vulnerable to hash-flooding
2. Practice
Breaking Murmur:

we’ve got the recipe –

now all we need is the (hash) cake
where are hashes used?
parser symbol tables
method lookup tables
attributes / instance variables
ip addresses
transaction ids
database indexing
session ids
http headers
json representation
url-encoded post form data
deduplication (HashSet)
A* search algorithm
dictionaries
...
where aren’t they used?
just recently

hash-DoS in btrfs file system (!)

http://crypto.junod.info/2012/12/13/hash-dos-and-btrfs/
can’t we use something different?
we could

but amortized constant time
is just too sexy
possible **real-life** attacks
need a high-profile target
web application
example #1

rails
first

attacking MurmurHash in ruby
apply the recipe
le demo
should work with rails

out of the box, no?
unfortunately, no
def parse_nested_query(qs, d = nil)

    params = KeySpaceConstrainedParams.new

    (qs || '').split(d ? /#{d} */n : DEFAULT_SEP).each do |p|
        k, v = p.split('=', 2).map { |s| unescape(s) }
        normalize_params(params, k, v)
    end

    return params.to_params_hash
end
def unescape(s, encoding = Encoding::UTF_8)
    URI.decode_www_form_component(s, encoding)
end
def self.decode_www_form_component(str, enc=Encoding::UTF_8)

    raise ArgumentError, "invalid %-encoding (#{str})"
    unless /\A[^%]*(?:%h%h[^%]*)*\z/ =~ str

    str.gsub(/\+|%h%h/).force_encoding(enc)

end
/\A[^%]*(?::%\h\h[^%]*)*\z/

???
catches invalid % encodings

(e.g. %ZV, %%%1 instead of %2F)
def parse_nested_query(qs, d = nil)

    params = KeySpaceConstrainedParams.new

    (qs || '').split(d ? /[#{d}] */n : DEFAULT_SEP).each do |p|
        k, v = p.split('=', 2).map { |s| unescape(s) }
        normalize_params(params, k, v)
    end

    return params.to_params_hash
end
def normalize_params(params, name, v = nil)
    name =~ %r(\A\[\[\]\]*([\^\[\]\]])\]*)
    k = $1 || ''
    ...
end
%r(\A[\[]\]*([\^\[]\]+)\]*

???
helps transform [[]] to []
idea
pre-generate matching values
create **random values**

passing the regular expressions
that should do it, right?
CONFIDENCE: The feeling you experience

before you fully understand the situation.
def parse_nested_query(qs, d = nil)

    params = KeySpaceConstrainedParams.new

    (qs || '').split(d ? /[#{d}] */n : DEFAULT_SEP).each do |p|
        k, v = p.split('=', 2).map { |s| unescape(s) }
        normalize_params(params, k, v)
    end

    return params.to_params_hash
end
class KeySpaceConstrainedParams

    def []=(key, value)

        @size += key.size if key && !@params.key?(key)

        raise RangeError, 'exceeded available parameter key space'
                              if @size > @limit

        @params[key] = value

    end

end
I am a clever little bastard
what now? rails is **safe**?
remember:

hashes are used everywhere
so if

application/x-www-form-urlencoded

doesn’t work, how about

application/json

?
again, with the encoding...
fast-forward...
le demo
conclusion

patchwork is not helping
too many places
code bloat
yet another loophole will be found
Fix it at the root

at the

root
example #2

java, enterprise™ edition
just apply the *recipe* (?)
String(byte[] bytes)
public String(byte bytes[], int offset, int length, 
        Charset charset) {
  ...
  char[] v = StringCoding.decode(charset, bytes, offset, length);
  ...
}

problem, byte[]?
tough nut to crack
what now? java is **safe**?
String(char[] value)
public String(char value[]) {
    int size = value.length;
    this.offset = 0;
    this.count = size;
    this.value = Arrays.copyOf(value, size);
}

int size = value.length;
this.offset = 0;
this.count = size;
this.value = Arrays.copyOf(value, size);
}
no decoding!
substitute **byte[]** operations with equivalent operations on **char[]**
le demo
disclosure

oracle (java): sep 11

cruby, jruby, rubinius: aug 30
oCERT advisory

CVEs were assigned

http://www.ocert.org/advisories/ocert-2012-001.html
more:
http://emboss.github.com/blog

code:
https://github.com/emboss/schadcode
reactions
java
ruby
cruby & jruby & rubinius == fixed

=> true


http://jruby.org/2012/12/03/jruby-1-7-1.html

https://github.com/rubinius/rubinius/commit/a9a40fc6a1256bcf6382631b710430105c5dd868
they did a fantastic job
(like last year)
so what was the fix?

how can we fix this?
WAIT
I'll fix it
Don’t use MurmurHash
CityHash?
“Inside Google, where CityHash was developed starting in 2010, we use variants of CityHash64() mainly in hash tables such as hash_map<string, int>.”

https://code.google.com/p/cityhash/
CityHash is **weaker** than MurmurHash

```
CityHash64( 0Y|L&:$;+[&HASH!, 16 ) = b553de6f34e878f
CityHash64( JkMR_ 0\7](HASH!, 16 ) = b553de6f34e878f
CityHash64( <jil7g;s`\(HASH!, 16 ) = b553de6f34e878f
CityHash64( e: yn"sg^a(HASH!, 16 ) = b553de6f34e878f
CityHash64( dt6PG8}?oz(HASH!, 16 ) = b553de6f34e878f
CityHash64( 8c-lkB%_Eo)HASH!, 16 ) = b553de6f34e878f
CityHash64( TdIx>DnK-1*HASH!, 16 ) = b553de6f34e878f
CityHash64( iM:9l=S"|e*HASH!, 16 ) = b553de6f34e878f
CityHash64( Z,r_|5xM0l*HASH!, 16 ) = b553de6f34e878f
CityHash64( .QH~S!9P(p*HASH!, 16 ) = b553de6f34e878f
CityHash64( {pF""wkd[F+HASH!, 16 ) = b553de6f34e878f
CityHash64( i< @)`\oy+?,HASH!, 16 ) = b553de6f34e878f
CityHash64( BU9[85WWp/ HASH!, 16 ) = b553de6f34e878f
CityHash64( 8{YDLn;d.2 HASH!, 16 ) = b553de6f34e878f
CityHash64( d+nkK&t?yr HASH!, 16 ) = b553de6f34e878f
CityHash64( {A.#v5i}V{ HASH!, 16 ) = b553de6f34e878f
```
Python’s hash()?
$ python -V
Python 2.7.3
$ time -p python -R poc.py
64 candidate solutions
Verified solutions for _Py_HashSecret:
145cc9aade7d2453 275daf6070a41b99
945cc9aade7d2453 a75daf6070a41b99
real 0.32
user 0.17
sys 0.02
Python 2.x and 3.x

- Randomization of hash() optional (-R)
- Instantaneous key recovery
- Multicollisions with TMTO
.NET’s Marvin32?
Something designed to be secure?
SipHash: a fast short-input PRF

New keyed hash to fix hash-flooding:
• Rigorous security requirements and analysis
• Speed competitive with that of weak hashes
• Can serve as MAC or PRF

Peer-reviewed research paper (A., Bernstein). published at DIAC 2012, INDOCRYPT 2012
SipHash initialization

256-bit state v0 v1 v2 v3
128-bit key k0 k1

v0 = k0 ⊕ 0x736f6d6570736575
v1 = k1 ⊕ 0x646f72616e646f6d
v2 = k0 ⊕ 0x6c7967656e657261
v3 = k1 ⊕ 0x7465646279746573
SipHash initialization

256-bit state $v0 \ v1 \ v2 \ v3$
128-bit key $k0 \ k1$

$v0 = k0 \oplus \text{"somepseud"}$
$v1 = k1 \oplus \text{"dorandom"}$
$v2 = k0 \oplus \text{"lygenera"}$
$v3 = k1 \oplus \text{"tedbytes"}$
SipHash compression

Message parsed as 64-bit words $m_0, m_1, \ldots$

$v_3 \oplus = m_0$

$c$ iterations of SipRound

$v_0 \oplus = m_0$
SipHash compression

Message parsed as 64-bit words $m_0$, $m_1$, ...

$v_3 \oplus = m_1$

c iterations of SipRound

$v_0 \oplus = m_1$
SipHash compression

Message parsed as 64-bit words $m_0$, $m_1$, ...

$v_3 \oplus = m_2$

$c$ iterations of SipRound

$v_0 \oplus = m_2$
SipHash compression

Message parsed as 64-bit words $m_0$, $m_1$, ...

Etc.
SipHash finalization

\[ v_2 \oplus = 255 \]

\( d \) iterations of SipRound

Return \( v_0 \oplus v_1 \oplus v_2 \oplus v_3 \)
SipHash-2-4 hashing 15 bytes
Family SipHash-c-d

Fast proposal: SipHash-2-4

Conservative proposal: SipHash-4-8

Weaker versions for cryptanalysis:
SipHash-1-0, SipHash-2-0, etc.
SipHash-1-1, SipHash-2-1, etc.
Etc.
Security claims

$\approx 2^{128}$ key recovery

$\approx 2^{192}$ state recovery

$\approx 2^{128}$ internal-collision forgery

$\approx 2^s$ forgery with probab. $2^{s-64}$
Fast diffusion of differences, thanks to optimized rotation counts

<table>
<thead>
<tr>
<th>Round</th>
<th>Differences</th>
<th>Prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>........................................ 8.................................... 8..................8...</td>
<td>1 (1)</td>
</tr>
<tr>
<td>2</td>
<td>8..................8... 8..................8..................8..................8... 8.....1...1.8... 8.1........1.1.8...</td>
<td>13 (14)</td>
</tr>
<tr>
<td>3</td>
<td>...1.8........1..... 8.....11a.1.1... 8.1.1...8...1.1 8.1.82.2........2.. a...1...8.1.8.11 8.12b413a2...... 92.8...8.21 82..92..82..82...</td>
<td>42 (56)</td>
</tr>
<tr>
<td>4</td>
<td>22...82...21..211 e835621322.1.235 22...21.8.122613 621.c21.42...42.3 2.11...24ca35e.13 66778453..57bd22 4.1.c...c212641 82..82...8.11.6...</td>
<td>103 (159)</td>
</tr>
<tr>
<td>5</td>
<td>a21182244a24e613 2ec144fcb8.115dd c245d93226674453 e2.18..48a34a6.3 f225f3ce8cd.c6d8 a44f51d8d.9e5616 2.445936ac53e25. a.4.d3.2.a5...51</td>
<td>152 (311)</td>
</tr>
<tr>
<td>6</td>
<td>52652.cc868.c689 27ba9d2d.e.fcd8 7ccdb44684.b.8ee 32246acc8cb4ce93 566.3a5175df891e 2.e5d3.249fb3ea6 4ee9de8a.8bfc67d 2425523ec62cf459</td>
<td>187 (498)</td>
</tr>
</tbody>
</table>

Combination of ADD and XOR ensures a high nonlinearity (e.g. against cube attacks)
How fast is SipHash-2-4?

On an old AMD Athlon II Neo (@1.6GHz)

<table>
<thead>
<tr>
<th>Bytes</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>123</td>
<td>134</td>
<td>158</td>
<td>204</td>
</tr>
<tr>
<td>(per byte)</td>
<td>(15.38)</td>
<td>(8.38)</td>
<td>(4.25)</td>
<td>(3.19)</td>
</tr>
<tr>
<td>MiBps</td>
<td>99</td>
<td>182</td>
<td>359</td>
<td>478</td>
</tr>
</tbody>
</table>

Long messages: **1.44** cycles/byte (1 GiBps)
Proof of simplicity

June 20: paper published online
June 28: 18 third-party implementations

C (Floodyberry, Boßlet, Neves); C# (Haynes)
Cryptol (Lazar); Erlang, Javascript, PHP (Denis)
Go (Chestnykh); Haskell (Hanquez)
Java, Ruby (Boßlet); Lisp (Brown); Perl6 (Julin)
Who is using SipHash?

- Rust
- CRuby
- Perl 5
- Rubinius
- OpenDNS
- redis
- JRuby

Soon:
Take home message

Hash-flooding DoS works by enforcing worst case in data structure operations through large multicollisions in the hash function.

Java and Rubies found vulnerable, due to their use of MurmurHash v2 or v3. CityHash and Python’s hash are weak too...

SipHash offers both security and performance.
SipHash paper, code, etc. available on https://131002.net/siphash

Attacks paper coming soon...