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. . . .*

oding DoS reloaded:
and defenses

ilippe Aumasson,
i Security (NAGRA)

ernstein,
ty of Illinois at Chicago &
che Universiteit Eindhoven

Boßlet,
ore 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
choose h
likely sp
collisions
hash tab
search.
entries w
`scanlog`
new pac
solved th
the num
discardi
the same
limit is r

S reloaded:
ses

nasson,
(NAGRA)

is at Chicago &
siteit Eindhoven

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 attac
choose her addres
likely spoofed) to
collisions, effective
hash table lookup
search. Dependin
entries we keep, t*
`scanlogd` *not be
new packets up in
solved this proble
the number of has
discarding the old
the same hash val
limit is reached.*

:

ago &

hoven

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 replacin hash table lookup with a lin search. Depending on how entries we keep, this might `scanlogd` *not be able to pi new packets up in time. . . . solved this problem by limit the number of hash collision discarding the oldest entry the same hash value when t 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.*

oding begins?

98 article
ing and attacking
n detection tools"
Designer (Alexander
in Phrack Magazine:

nlogd, I'm using a hash lookup source addresses. rks very well for the case … average lookup better than that of a earch. …

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 a
(rememb
all scans
not be a
other at
worth m
issues al
operatin
example
used the
connectr
There're
which m
dangero
more res*

ins?

tacking
n tools"
(Alexander
k Magazine:

n using a hash
ource addresses.
vell for the
verage lookup
n that of a

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
(remember, we ca
all scans anyway),
not be acceptable
other attacks. . . .
worth mentioning
issues also apply t
operating system
example, hash tab
used there for look
connections, listen
There're usually o
which make these
dangerous though,
more research mig

er
e:
ash
esses.
kup
n

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 s
(remember, we can't detect
all scans anyway), but migh
not be acceptable for detect
other attacks. . . . It is prob
worth mentioning that simil
issues also apply to things li
operating system kernels. F
example, hash tables are wi
used there for looking up ac
connections, listening ports,
There're usually other limits
which make these not really
dangerous though, but
more research might be nee

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."

r, an attacker can
her addresses (most
oofed) to cause hash
s, effectively replacing the
ble lookup with a linear
Depending on how many
ve keep, this might make
gd not be able to pick
kets up in time. . . . I've
his problem by limiting
ber of hash collisions, and
ng the oldest entry with
e hash value when the
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 c

Choose

Hash tal
Store str
where i

With n
expect ≈
in each
Choose
expect v
so very f

(What if
Rehash:

ker can
ses (most
cause hash
ely replacing the
with a linear
g on how many
his might make
able to pick
time. . . . I've
m by limiting
sh collisions, and
est entry with
ue when the

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 h

Choose $\ell \in \{1, 2, 4$

Hash table: ℓ sepa
Store string s in li
where $i = H(s)$ m

With n entries in t
expect $\approx n/\ell$ entr
in each linked list.
Choose $\ell \approx n$:
expect very short l
so very fast list op

(What if n becom
Rehash: replace ℓ

h
ng the
ear
many
make
ck
I've
ing
ns, and
with
he

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: ℓ separate linke
Store string s in list $\#i$
where $i = H(s) \bmod \ell$.

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.)

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

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.)

acceptable for port scans
ber, we can't detect
s anyway), but might
acceptable for detecting
tacks. . . . It is probably
entioning that similar
lso apply to things like
g system kernels. For
, hash tables are widely
ere for looking up active
ions, listening ports, etc.
e usually other limits
make these not really
us though, but
search might be needed."

Review of classic hash tables

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

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

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.)

e.g. stri
$H(s) =$

$\vdots$

`e→eigh`
`f→four`
$\vdots$
`n→nine`
`o→one`
`p`
`q`
`r`
`s→six-`
`t→two-`
$\vdots$

for port scans
n't detect
but might
for detecting
It is probably
that similar
o things like
kernels. For
les are widely
king up active
ing ports, etc.
ther limits
not really
, but
ht be needed."

Review of classic hash tables

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

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

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.)

e.g. strings `one`, t
$H(s)$ = first byte

⋮

```
e→eight
f→four→five
```

⋮

```
n→nine
o→one
p
q
r
s→six→seven
t→two→three→
```

⋮

cans

t

ing

ably

ar

ke

or

dely

tive

etc.

s

ded."

Review of classic hash tables

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

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

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.)

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

$\vdots$

```
e→eight
f→four→five
```

$\vdots$

```
n→nine
o→one
p
q
r
s→six→seven
t→two→three→ten
```

$\vdots$

Review of classic hash tables

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

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

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.)

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

$\vdots$

`e`→`eight`

`f`→`four`→`five`

$\vdots$

`n`→`nine`

`o`→`one`

`p`

`q`

`r`

`s`→`six`→`seven`

`t`→`two`→`three`→`ten`

$\vdots$

of classic hash tables

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

ble: ℓ separate linked lists.
ring s in list $\#i$
$= H(s) \bmod \ell$.

entries in table,
$\approx n/\ell$ entries
linked list.
$\ell \approx n$:
very short linked lists,
fast list operations.

f n becomes too big?
replace ℓ by 2ℓ.)

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

$\vdots$

e→eight

f→four→five

$\vdots$

n→nine

o→one

p

q

r

s→six→seven

t→two→three→ten

$\vdots$

$H(s) =$
is not a

Typical s
Very lon

In some
start wit

hash tables

$4, 8, 16, \ldots\}$.

arate linked lists.

st $\#i$

od ℓ.

table,

ries

linked lists,

perations.

es too big?

by 2ℓ.)

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

⋮

e→eight

f→four→five

⋮

n→nine

o→one

p

q

r

s→six→seven

t→two→three→ten

⋮

$H(s)$ = first byte o

is not a good hash

Typical strings oft

Very long `t` list; ve

In some applicatio

start with the sam

s

.}.

d lists.

s,

?

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

$\vdots$

`e`→`eight`

`f`→`four`→`five`

$\vdots$

`n`→`nine`

`o`→`one`

`p`

`q`

`r`

`s`→`six`→`seven`

`t`→`two`→`three`→`ten`

$\vdots$

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

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

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

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

$\vdots$

`e`→`eight`

`f`→`four`→`five`

$\vdots$

`n`→`nine`

`o`→`one`

`p`

`q`

`r`

`s`→`six`→`seven`

`t`→`two`→`three`→`ten`

$\vdots$

$H(s)$ = 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.

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

$\vdots$

`e`→`eight`

`f`→`four`→`five`

$\vdots$

`n`→`nine`

`o`→`one`

`p`

`q`

`r`

`s`→`six`→`seven`

`t`→`two`→`three`→`ten`

$\vdots$

$H(s)$ = 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.

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

$\vdots$

`e`$\to$`eight`

`f`$\to$`four`$\to$`five`

$\vdots$

`n`$\to$`nine`

`o`$\to$`one`

`p`

`q`

`r`

`s`$\to$`six`$\to$`seven`

`t`$\to$`two`$\to$`three`$\to$`ten`

$\vdots$

$H(s)$ = 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?

ngs `one`, `two`, . . . , `ten`;
first byte of s; $\ell = 256$:

`nt`

`r`$\rightarrow$`five`

e

$\rightarrow$`seven`

$\rightarrow$`three`$\rightarrow$`ten`

$H(s) =$ 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

Attacker

$s_1, \ldots, s$
$\cdots = H($

Then all
in the sa
linked lis

two, . . . , ten;
of s; $\ell = 256$:

ten

$H(s) =$ 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

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

Then all strings ar
in the same linked
linked list become

`en;`
256:

$H(s)$ = 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
$\cdots = H(s_n) \bmod \ell$.

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 $H(s_1) \bmod \ell = \cdots = H(s_n) \bmod \ell$.

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 $H(s_1) \bmod \ell = \cdots = H(s_n) \bmod \ell$.

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.

$H(s)$ = 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 $H(s_1) \bmod \ell = \cdots = H(s_n) \bmod \ell$.

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.

first byte of s

good hash function!

strings often start with `t`.

g `t` list; very slow.

applications, *most* strings

th the same letter.

se fast hash functions

k at the whole string s.

of programmers exploring

nctions for hash tables

speed for typical strings.

the strings aren't typical?

Hashing malicious strings

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

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.

Decemb

dnscach

>41000

from 50

```
if (+
  /*
```

Discardi

trivially

if attack

But wha

general-

language

Can't th

of s
n function!

en start with `t`.
ery slow.

ns, *most* strings
e letter.

sh functions
hole string s.

mmers exploring
hash tables
typical strings.

s aren't typical?

Hashing malicious strings

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

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, B
`dnscache` softwar
>41000000000000
from 50 million In

```
if (++loop > 1
  /* to protec
     hash floo
```

Discarding cache e
trivially maintains
if attacker floods

But what about ha
general-purpose pr
languages and libr
Can't throw entrie

ith t.

strings

ns
g s.

ploring
es
rings.

pical?

Hashing malicious strings

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

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 (OpenD
>41000000000000 DNS req
from 50 million Internet use

```
  if (++loop > 100) retur
    /* to protect against
       hash flooding */
```

Discarding cache entries
trivially maintains performan
if attacker floods hash table

But what about hash tables
general-purpose programmin
languages and libraries?
Can't throw entries away!

Hashing malicious strings

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

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 throw entries away!

malicious strings

r provides strings
s_n with $H(s_1) \bmod \ell =$
$(s_n) \bmod \ell$.

strings are stored
ame linked list;
st becomes very slow.

: Replace linked list
e tree structure,
if list is big.

lementors are unhappy:
tion throws away the
y 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 throw entries away!

Bad solu
Use SHA
SHA-3 i

strings

strings
$(s_1) \bmod \ell =$
ℓ.

e stored
list;
s very slow.

linked list
cture,
g.

are unhappy:
vs away the
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 throw entries away!

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

$\ell =$

v.

py:

e

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: >4100000000000 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: $>$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!

Two reasons this is bad:

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

December 1999, Bernstein, `dnscache` software (OpenDNS: >4100000000000 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.

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

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

er 1999, Bernstein,
he software (OpenDNS:
000000000 DNS requests
million Internet users):

```
+loop > 100) return 0;
to protect against
hash flooding */
```

ng cache entries
maintains performance
ker floods hash table.

nt about hash tables in
purpose programming
es and libraries?
row 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.

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

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

2003 US
Symposi
"Denial
algorithm

"We pre
low-banc
attacks
hashes t
the hash
degenera

Attack e
Perl prog
Squid w

Bernstein,
e (OpenDNS:
0 DNS requests
ternet users):

```
00) return 0;
t against
ding */
```

entries
performance
hash table.

ash tables in
rogramming
aries?
es 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.

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

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

2003 USENIX Sec
Symposium, Crosb
"Denial of service
algorithmic comple

"We present a nev
low-bandwidth der
attacks . . . if each
hashes to the sam
the hash table will
degenerate to a lir

Attack examples:
Perl programming
Squid web cache,

NS:
uests
rs):

```
rn 0;
```

.

nce

.

in

ng

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.

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

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

2003 USENIX Security
Symposium, Crosby–Wallach
"Denial of service via
algorithmic complexity attac

"We present a new class of
low-bandwidth denial of serv
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.

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

ℓ is small: e.g., $\ell = 2^{20}$.
No matter how strong H is,
attacker can easily compute
$H(s) \bmod 2^{20}$ 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.

ution:
A-3 for H.
s collision-resistant!

sons this is bad:
ery slow.
esn't solve the problem.

od ℓ
ollision-resistant.

ll: e.g., $\ell = 2^{20}$.
ter how strong H is,
can easily compute
od 2^{20} for many s
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 (28
"Efficien
on web a

Java, JR
Python 2
Ruby, Ap
Tomcat,
Plone, R
Engine.

oCERT

Applicat
use secre
. . . but

-resistant!

s bad:

the problem.

istant.

$= 2^{20}$.

rong H is,

compute

many s

ons.

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), Klin

"Efficient denial o

on web applicatio

Java, JRuby, PHP

Python 2, Python

Ruby, Apache Ger

Tomcat, Oracle G

Plone, Rack, V8 J

Engine.

oCERT advisory 2

Application respo

use secret key to

. . . but is this sec

em.

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 a on web application platform

Java, JRuby, PHP 4, PHP 5 Python 2, Python 3, Rubiniu Ruby, Apache Geronimo, Ap Tomcat, Oracle Glassfish, Je Plone, Rack, V8 Javascript Engine.

oCERT advisory 2011–003.

Application response:
use secret key to *randomize*
. . . but is this secure?

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”:

Java, JRuby, PHP 4, PHP 5, Python 2, Python 3, Rubinius, Ruby, Apache Geronimo, Apache Tomcat, Oracle Glassfish, Jetty, Plone, Rack, V8 Javascript Engine.

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

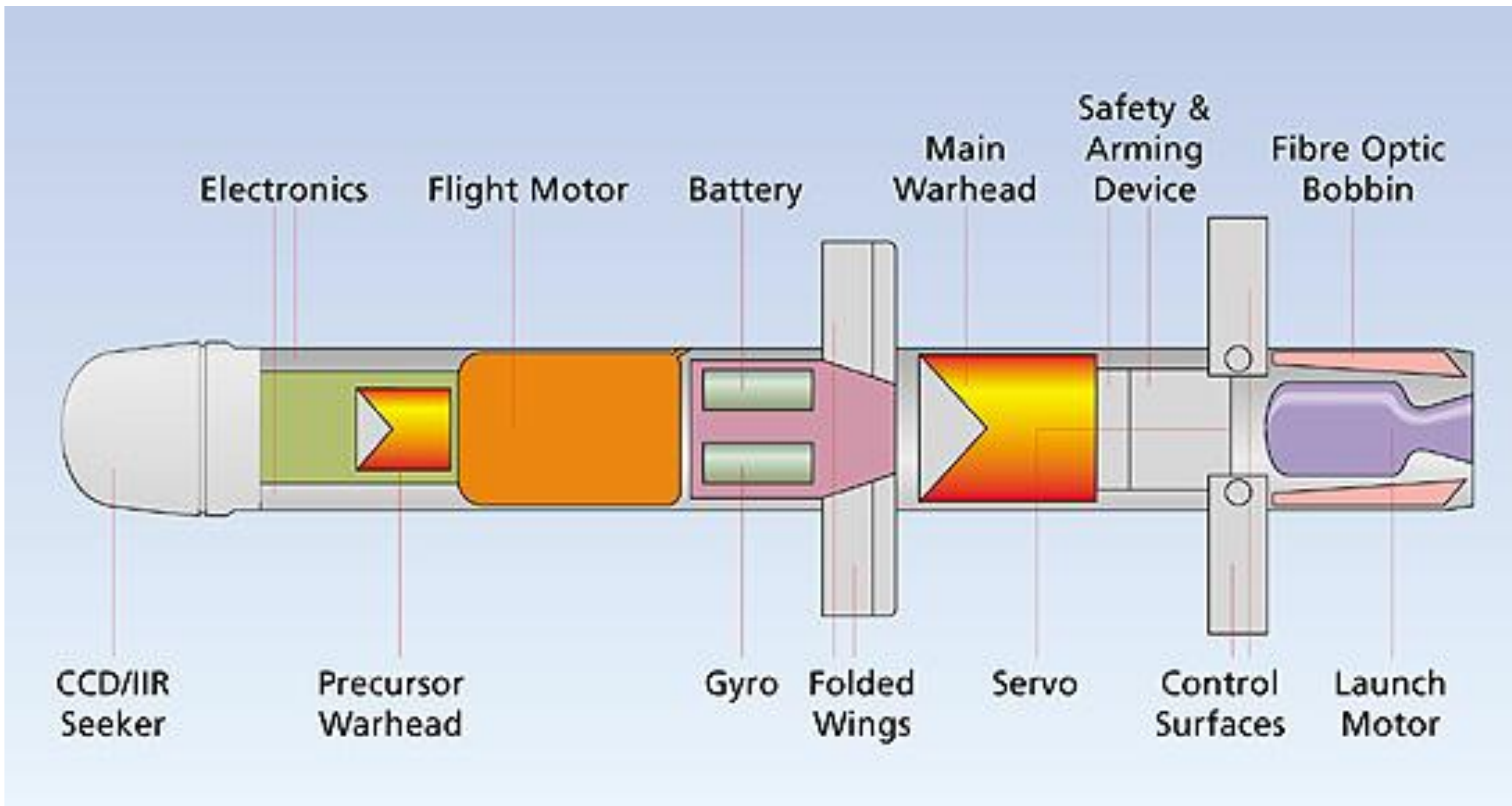
http://code.google.com/p/smhasher/wiki/MurmurHash

MurmurHash3

“successor to MurmurHash2”

Oracle & OpenJDK, Rubinius

1. Theory



Electronics
Flight Motor
Battery
Main Warhead
Safety & Arming Device
Fibre Optic Bobbin
CCD/IIR Seeker
Precursor Warhead
Gyro
Folded Wings
Servo
Control Surfaces
Launch Motor

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 7: h ^= data[6] << 48;
    case 6: h ^= data[5] << 40;
    case 5: h ^= data[4] << 32;
    case 4: h ^= data[3] << 24;
    case 3: h ^= data[2] << 16;
    case 2: h ^= data[1] << 8;
    case 1: h ^= data[0];
    h *= m;

};

…
```

8-byte-aligned data => 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;

}
```

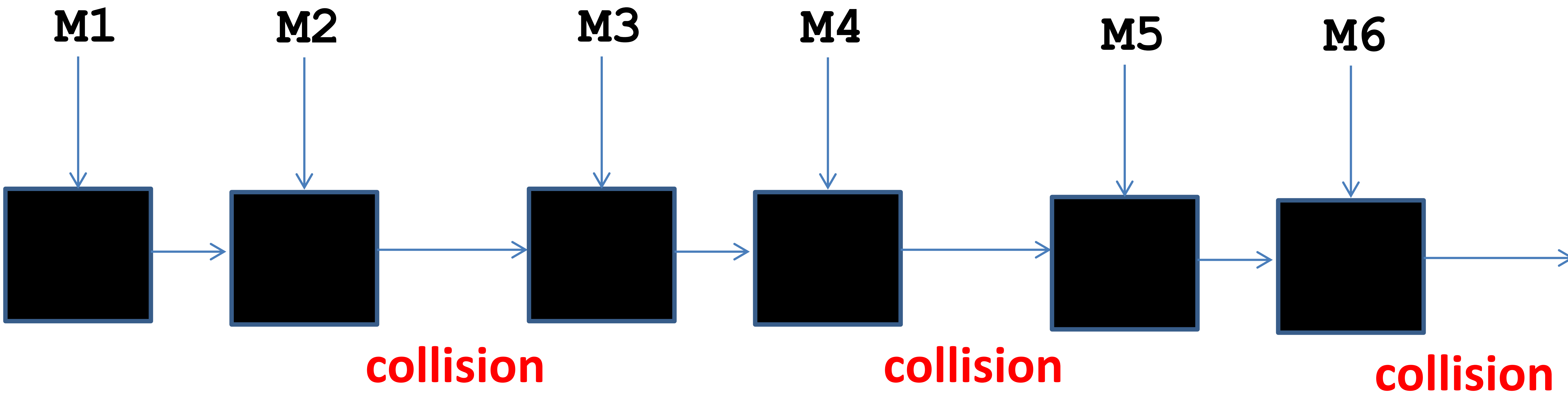
```
const uint64_t m = (0xc6a4a793 << 32) | 0x5bd1e995;
uint64_t h = seed ^ len;

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;

}
```

```c
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;
}
```

```c
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;             /* COLLISION !!!
                          (0x80... ^ 0x80... = 0) */

   data += 8;
   len -= 8;

}
```



chain collisions ⇒ multicollisions

16n bytes ⇒ 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

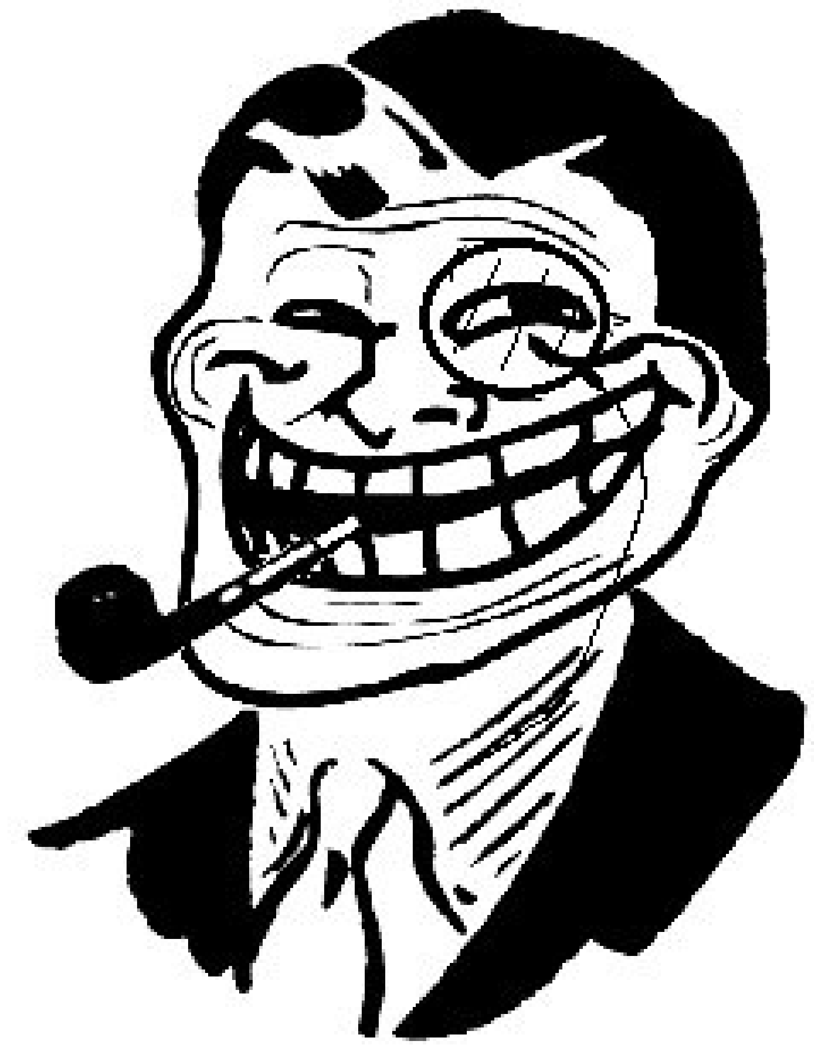
example #1

rails

first

attacking MurmurHash in ruby

apply the **recipe**



le demo

should work with rails

out of the box, no?

unfortunately, no

```ruby
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
```

```ruby
def unescape(s, encoding = Encoding::UTF_8)
  URI.decode_www_form_component(s, encoding)
end
```

```ruby
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/, TBLDECWWWCOMP_).force_encoding(enc)
end
```

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

???

catches **invalid % encodings**

(e.g. %ZV, %%1 instead of %2F)

```ruby
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
```

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

```
%r(\A[\[\]]*([^\[\]]+)\]*)
```

???

helps transform [[]] to []

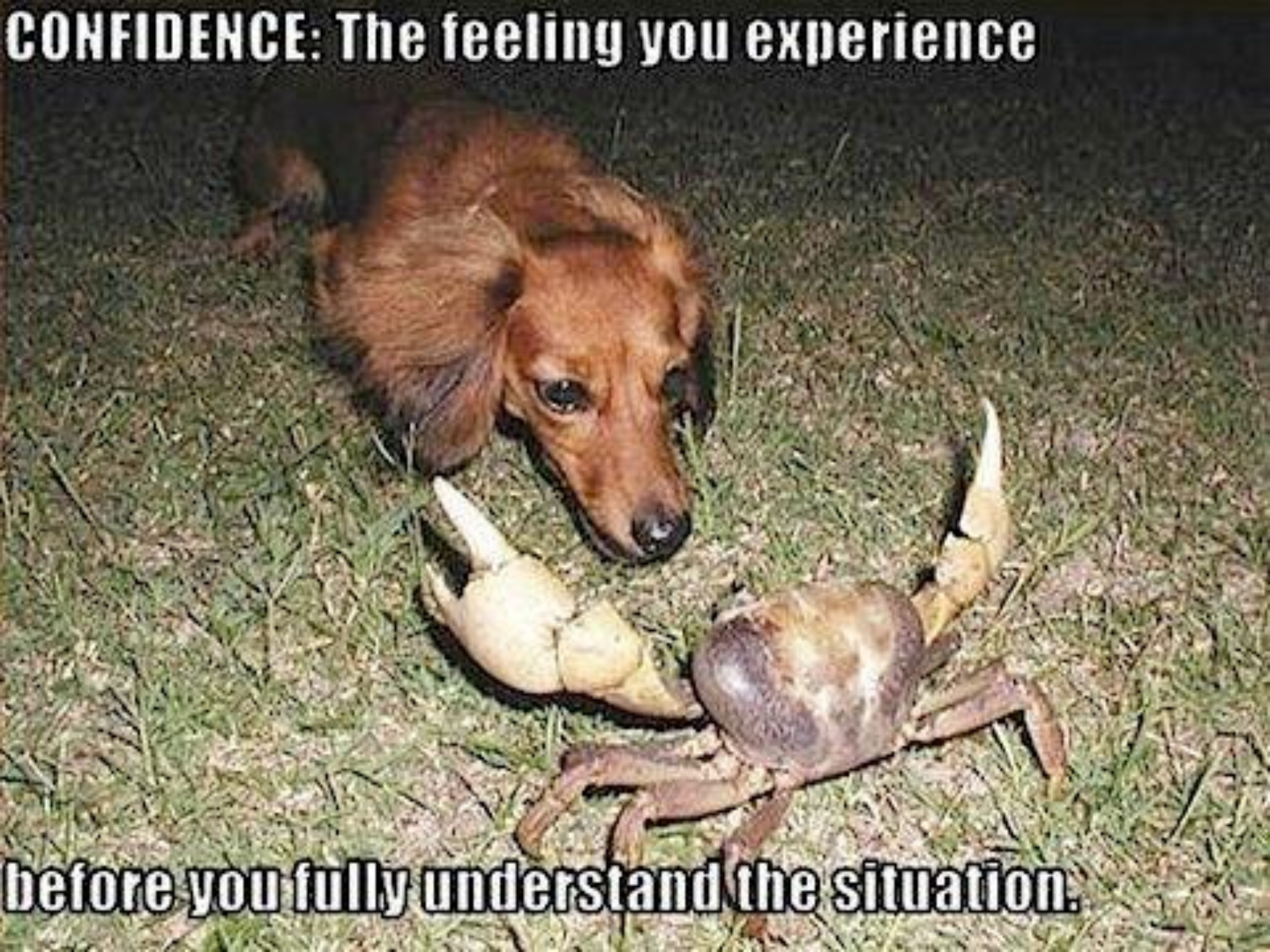
idea

pre-generate matching values

create random values

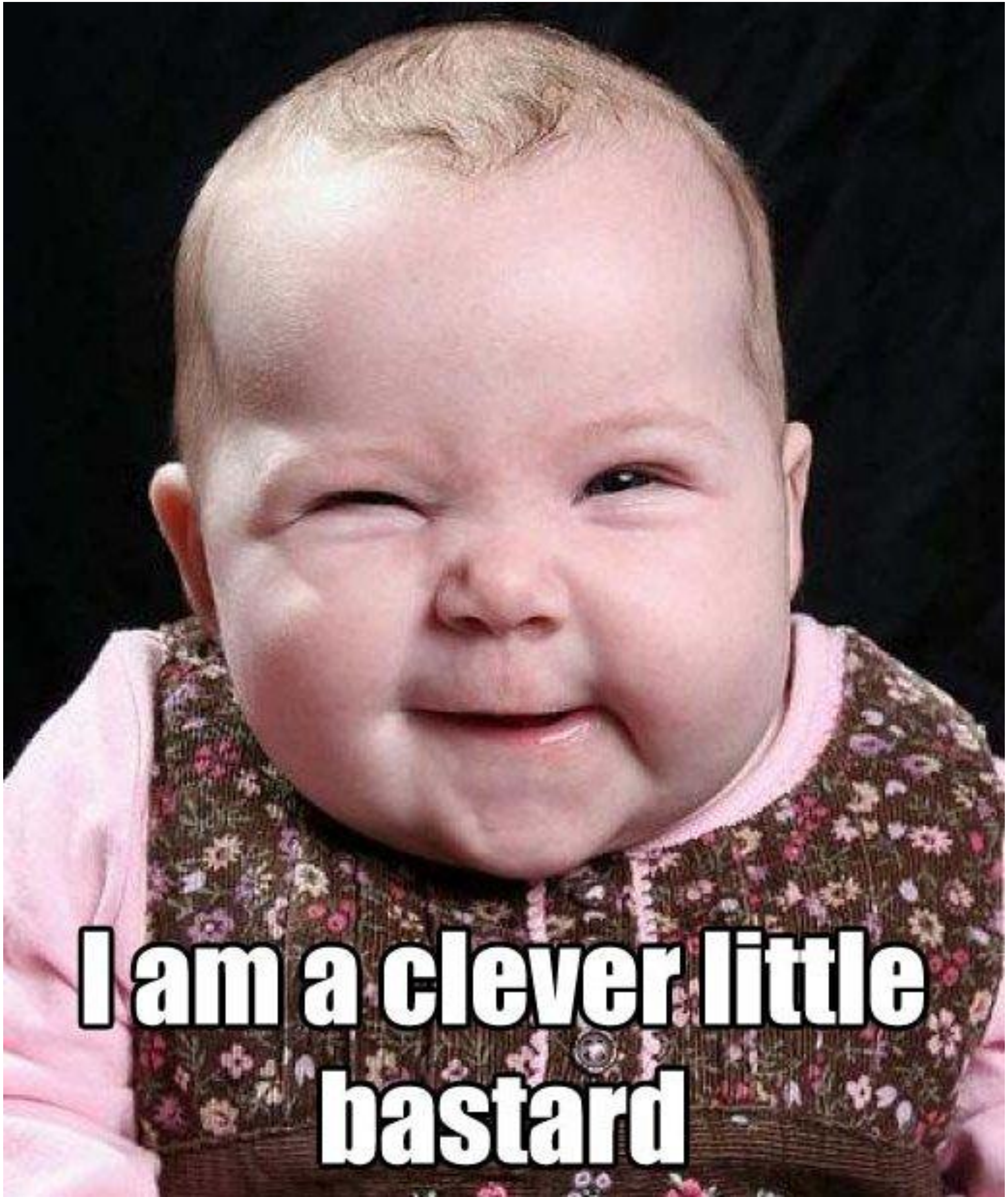
passing the regular expressions

that should do it, right?


CONFIDENCE: The feeling you experience
before you fully understand the situation.

```ruby
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
```

```ruby
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

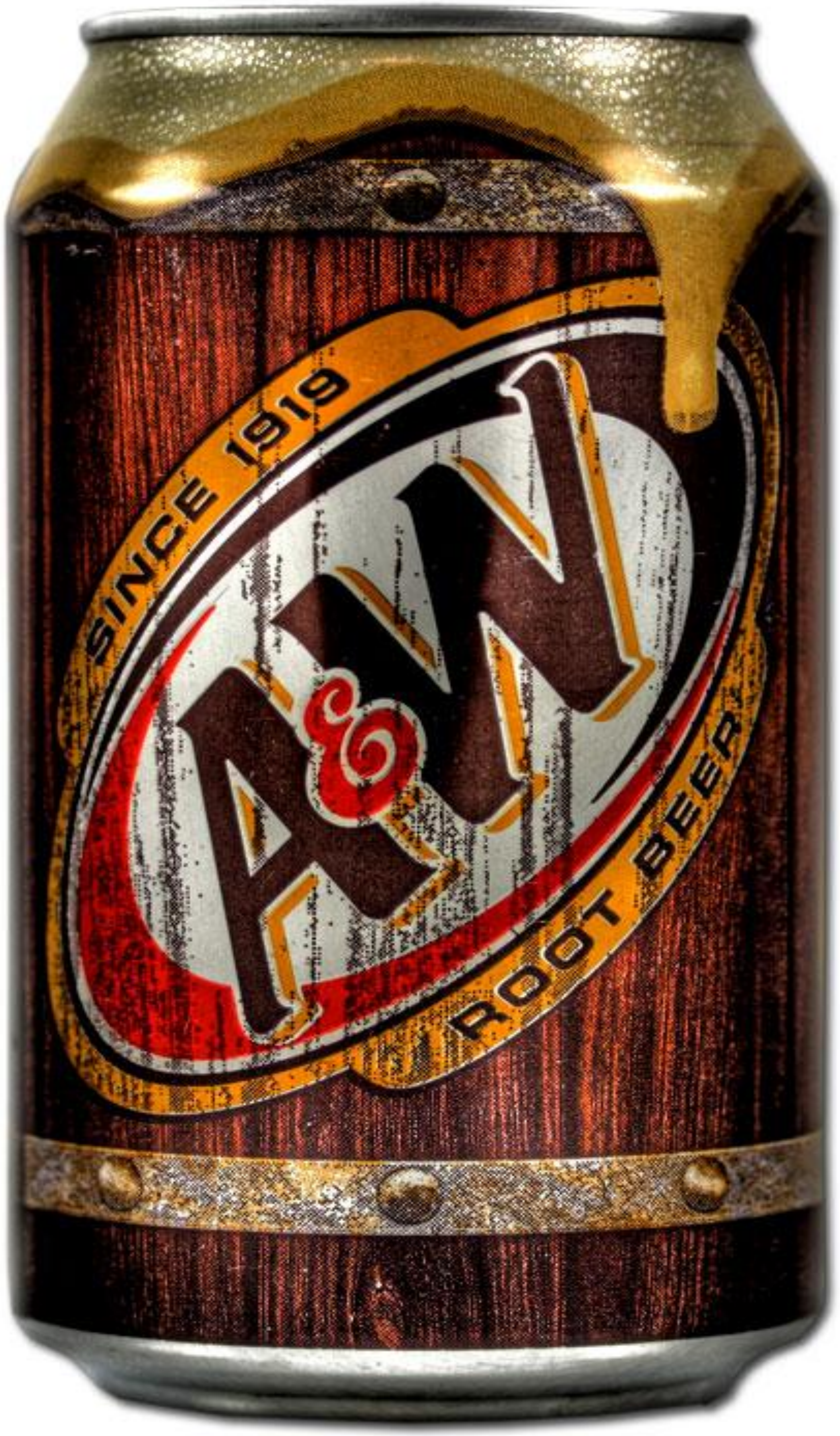
patchwork is not helping

too many places

code bloat

yet another loophole will be found

Fix it



at the

root

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

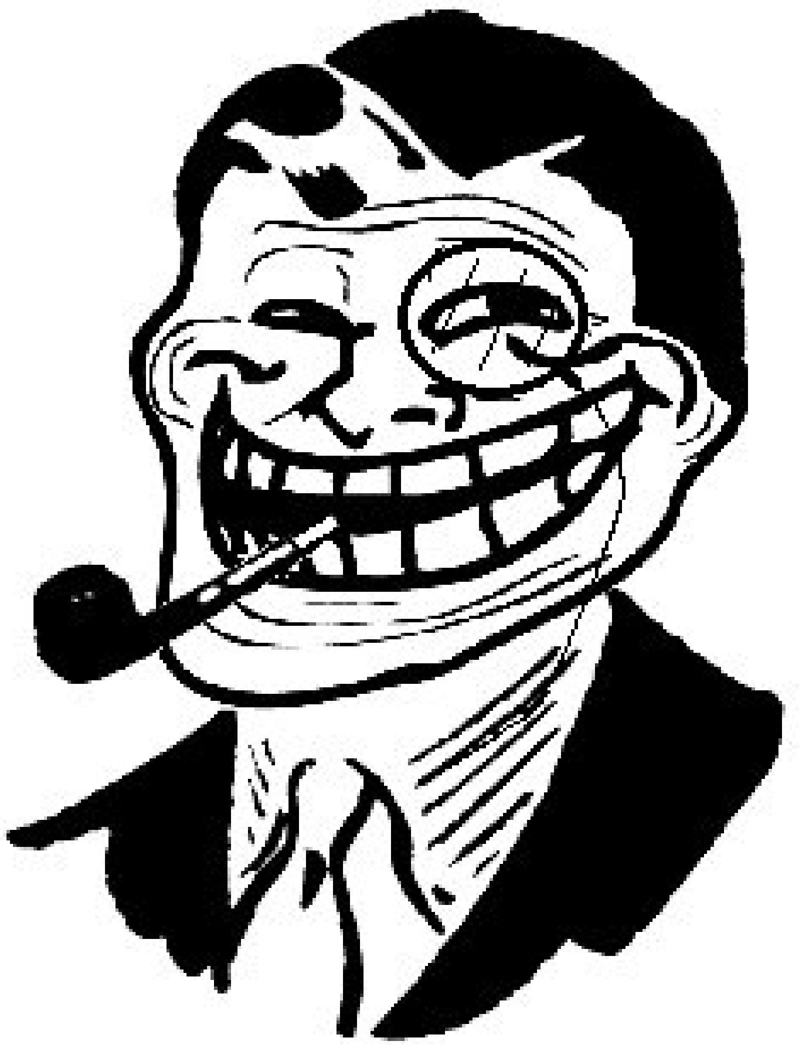
what now? java is safe?

String(char[] value)

```java
public String(char value[]) {

    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://www.ruby-lang.org/en/news/2012/11/09/ruby19-hashdos-cve-2012-5371/

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( <jiI7g;s,`(HASH!, 16 )      = b553de6f34e878f
CityHash64( e: yn"sg^a(HASH!, 16 )      = b553de6f34e878f
CityHash64( dt6PG8}?oz(HASH!, 16 )      = b553de6f34e878f
CityHash64( 8c-lkD%_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

$v0 = k0 \oplus 0x736f6d6570736575$

$v1 = k1 \oplus 0x646f72616e646f6d$

$v2 = k0 \oplus 0x6c7967656e657261$

$v3 = k1 \oplus 0x7465646279746573$

SipHash initialization

256-bit state v0 v1 v2 v3

128-bit key k0 k1

v0 = k0 $\oplus$ “somepseu”

v1 = k1 $\oplus$ “dorandom”

v2 = k0 $\oplus$ “lygenera”

v3 = k1 $\oplus$ “tedbytes”

SipHash compression

Message parsed as 64-bit words **m0**, **m1**, …

v3 $\oplus$= **m0**

c iterations of SipRound

v0 $\oplus$= **m0**

SipHash compression

Message parsed as 64-bit words **m0**, **m1**, …

v3 ⊕= **m1**

c iterations of SipRound

v0 ⊕= **m1**

SipHash compression

Message parsed as 64-bit words **m0**, **m1**, …

v3 $\oplus$= **m2**

c iterations of SipRound

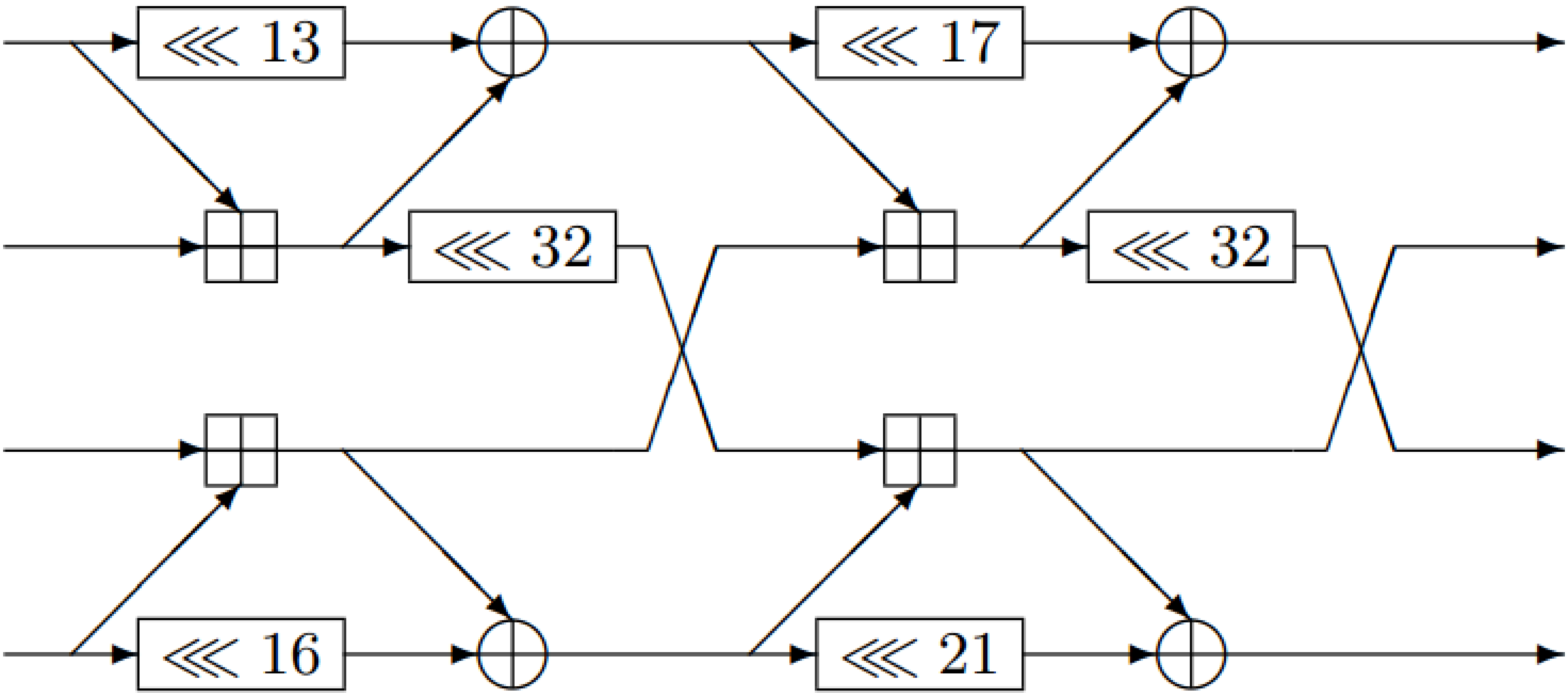
v0 $\oplus$= **m2**

SipHash compression

Message parsed as 64-bit words **m0**, **m1**, …

Etc.

SipRound



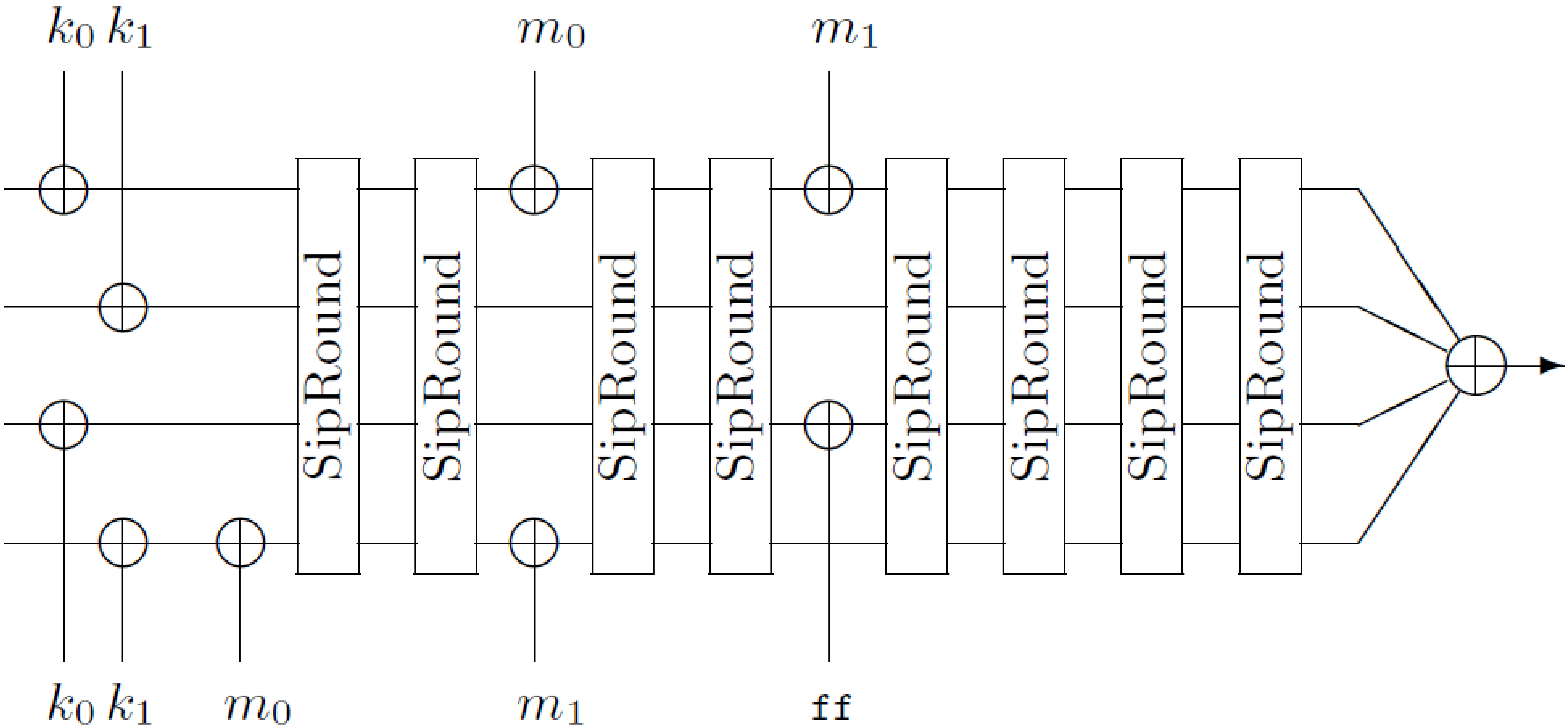
SipHash finalization

$v2 \oplus= 255$

d iterations of SipRound

Return $v0 \oplus v1 \oplus v2 \oplus v3$

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

Round	Differences				Prob.
1	....	....	.... 8...............	8............... 8...........8...	1 (1)
2	8...........8... ....8...........	8............... ....9...	........8....... 8.....1.8.1.8...	8.....1...1.8... 8.1.......1.....	13 (14)
3	..1.8.....1..... a...1...8.1.8.11	8.....11a.1.1... 8.12b413a2......	8.1.1...8.....1. ....92..8....21.	8.1.82.......2.. 82..92..82..82..	42 (56)
4	22..82...21..211 2.11..24ca35e.13	e835621322.1.235 66778453..57bd22	22...21.8.122613 4.1.c...c212641.	621.c21.42..42.3 82..82..8.11.6..	103 (159)
5	a21182244a24e613 f225f3ce8cd.c6d8	2ec144fcb8.115dd a44f51d8d.9e5616	c245d93226674453 2.445936ac53e25.	e2.18..48a34a6.3 a.4.d3.2.a5...51	152 (311)
6	52652.cc868.c689 566.3a5175df891e	27baa9d2d.e.fcd8 2.e5d3.249fb3ea6	7ccdb44684.b.8ee 4ee9de8a.8bfc67d	32246acc8cb4ce93 2425523ec62cf459	187 (498)

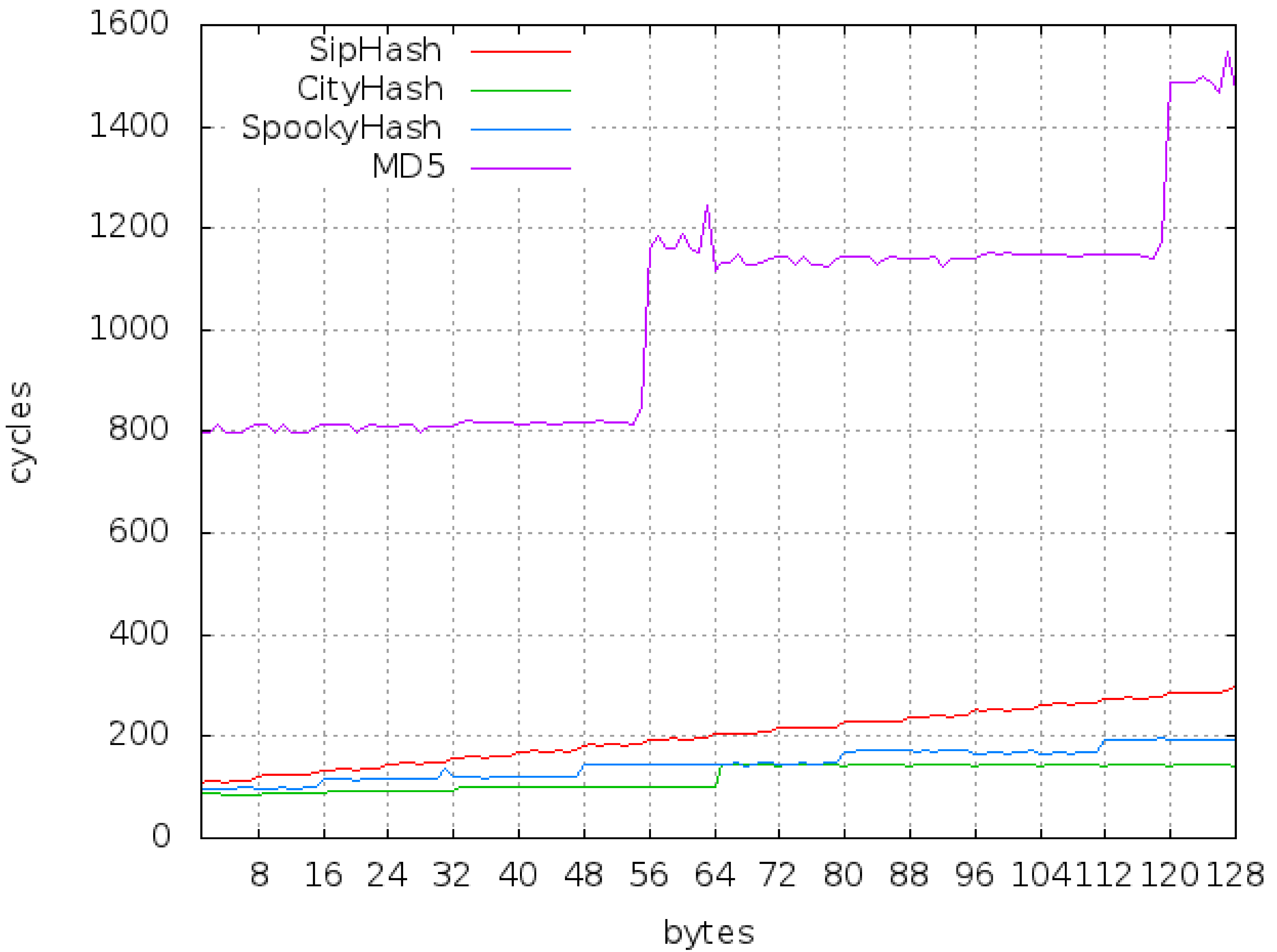
Combination of ADD and XOR ensures a high nonlinearity (e.g. against cube attacks)

How fast is SipHash-2-4?

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

Bytes	8	16	32	64
Cycles (per byte)	123 (15.38)	134 (8.38)	158 (4.25)	204 (3.19)
MiBps	99	182	359	478

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


SipHash
CityHash
SpookyHash
MD5
cycles
bytes
0
200
400
600
800
1000
1200
1400
1600
8
16
24
32
40
48
56
64
72
80
88
96
104
112
120
128

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?

OpenDNS

Perl 5

Rust

CRuby

redis

JRuby

Rubinius

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...