SipHash: a fast short-input PRF

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

Joint work with:

Jean-Philippe Aumasson,
Kudelski Security (NAGRA)

https://131002.net/siphash/

Several motivations:

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

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

3. Application: authenticate Internet packets.

4. Application: defend against hash flooding.

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

:

ort-input PRF

ernstein,
ty of Illinois at Chicago &
che Universiteit Eindhoven

rk with:

ilippe Aumasson,
i Security (NAGRA)

//131002.net/siphash/

ement:
ition coming soon
enticated ciphers!

Several motivations:

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

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

3. Application: authenticate Internet packets.

4. Application: defend against hash flooding.

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

Today's

July 199
"Designi
port sca
by Solar
Peslyak)

"In sca
table to
This wo
typical c
time is b
binary s

PRF

is at Chicago &
siteit Eindhoven

nasson,
(NAGRA)

.net/siphash/

ng soon
ciphers!

Several motivations:

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

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

3. Application: authenticate Internet packets.

4. Application: defend against hash flooding.

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

Today's focus: has

July 1998 article
"Designing and at
port scan detectio
by Solar Designer
Peslyak) in Phrack

"*In* `scanlogd`*, I'm*
table to lookup so
This works very w
typical case . . . av
time is better than
binary search. . . .

ago &
hoven

ash/

Several motivations:

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

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

3. Application: authenticate Internet packets.

4. Application: defend against hash flooding.

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

Today's focus: hash floodin

July 1998 article
"Designing and attacking
port scan detection tools"
by Solar Designer (Alexande
Peslyak) in Phrack Magazin

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

Several motivations:

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

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

3. Application: authenticate Internet packets.

4. Application: defend against hash flooding.

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

Today's focus: hash flooding

July 1998 article "Designing and attacking port scan detection tools" by Solar Designer (Alexander Peslyak) in Phrack Magazine:

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

motivations:

mize secret-key crypto
t *messages*.

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

cation:
cate Internet packets.

cation:
gainst hash flooding.

yze security of
sh-flooding defenses.
o work with Martin Boßlet
his much further.

Today's focus: hash flooding

July 1998 article
"Designing and attacking
port scan detection tools"
by Solar Designer (Alexander
Peslyak) in Phrack Magazine:

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

However
choose h
likely sp
collision
the hash
linear se
many en
make sc
new pac
solved th
the num
discardin
the same
limit is r

ns:

t-key crypto
s.

MAC that's
imple.

net packets.

sh flooding.

y of
g defenses.
h Martin Boßlet
further.

Today's focus: hash flooding

July 1998 article
"Designing and attacking
port scan detection tools"
by Solar Designer (Alexander
Peslyak) in Phrack Magazine:

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

*However, an attac
choose her addres
likely spoofed) to
collisions, effective
the hash table loo
linear search. Dep
many entries we k
make scanlogd no
new packets up in
solved this proble
the number of has
discarding the old
the same hash val
limit is reached.*

to

s

s.

g.

s.
Boßlet

Today's focus: hash flooding

July 1998 article
"Designing and attacking
port scan detection tools"
by Solar Designer (Alexander
Peslyak) in Phrack Magazine:

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

However, an attacker can
choose her addresses (most
likely spoofed) to cause has
collisions, effectively replaci
the hash table lookup with a
linear search. Depending on
many entries we keep, this m
make scanlogd not be able t
new packets up in time. . . .
solved this problem by limiti
the number of hash collision
discarding the oldest entry w
the same hash value when t
limit is reached.

Today's focus: hash flooding

July 1998 article
"Designing and attacking
port scan detection tools"
by Solar Designer (Alexander
Peslyak) in Phrack Magazine:

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

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

focus: hash flooding

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

sh flooding

tacking

n tools"

(Alexander

k Magazine:

n using a hash

urce addresses.

ell 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

g

er
e:

hash
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 proba
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 n table lookup with a arch. Depending on how ntries we keep, this might anlogd 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.”

Decemb

```
dnscach
  if (++
    /*
```

Discardi
trivially
if attack

But wha
general-p
language
Can't th

ker can
ses (most
cause hash
ely replacing
kup with a
pending on how
eep, this might
t be 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."

December 1999, B
`dnscache` softwar

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

Discarding cache e
trivially maintains
if attacker floods

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

h
ng
a
how
might
to pick
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."

December 1999, Bernstein,
dnscache software:

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

Discarding cache entries trivially maintains performar if attacker floods hash table

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

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

December 1999, Bernstein, `dnscache` software:

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

Discarding cache entries trivially maintains performance if attacker floods hash table.

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

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
nake these not really
us though, but
search might be needed."

December 1999, Bernstein,
`dnscache` software:

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

Discarding cache entries
trivially maintains performance
if attacker floods hash table.

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

2003 US
Symposi
"Denial
algorithr

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

Attack e
Perl prog
Squid we
No attac

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."

December 1999, Bernstein, `dnscache` software:

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

Discarding cache entries trivially maintains performance if attacker floods hash table.

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

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

"We present a ne
low-bandwidth de
attacks . . . if each
hashes to the sam
the hash table will
degenerate to a li

Attack examples:
Perl programming
Squid web cache,
No attack on `dnsc`

cans

t
ing
ably
ar
ke
or
dely
tive
etc.
;

ded."

December 1999, Bernstein,
`dnscache` software:

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

Discarding cache entries trivially maintains performance if attacker floods hash table.

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

2003 USENIX Security Symposium, Crosby–Wallach
"Denial of service via algorithmic complexity 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.
No attack on `dnscache`.

December 1999, Bernstein, `dnscache` software:

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

Discarding cache entries trivially maintains performance if attacker floods hash table.

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

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

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

Attack examples:
Perl programming language,
Squid web cache, etc.
No attack on `dnscache`.

er 1999, Bernstein,
ne software:

```
+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!

2003 USENIX Security Symposium, Crosby–Wallach, “Denial of service via algorithmic complexity attacks”:

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

Attack examples:
Perl programming language,
Squid web cache, etc.
No attack on `dnscache`.

2011 (28
“Efficien
on web a
oCERT

No attac
fixed Pe
but still
PHP 4,
3, Rubin
Geronim
Oracle G
Rack, V8

Bernstein,
e:

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

entries
performance
hash table.

ash tables in
rogramming
aries?
es away!

2003 USENIX Security Symposium, Crosby–Wallach, “Denial of service via algorithmic complexity attacks”:

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

Attack examples:
Perl programming language,
Squid web cache, etc.
No attack on `dnscache`.

2011 (28C3), Klin
“Efficient denial o
on web application
oCERT advisory 2

No attack on `dnsc`
fixed Perl, fixed Sq
but still problems
PHP 4, PHP 5, Py
3, Rubinius, Ruby,
Geronimo, Apache
Oracle Glassfish, J
Rack, V8 Javascrip

```
rn 0;
```

.

nce

r

in

ng

2003 USENIX Security Symposium, Crosby–Wallach, “Denial of service via algorithmic complexity attacks”:

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

Attack examples:
Perl programming language,
Squid web cache, etc.
No attack on `dnscache`.

2011 (28C3), Klink–Wälde,
“Efficient denial of service a
on web application platform
oCERT advisory 2011–003:

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

2003 USENIX Security Symposium, Crosby–Wallach, “Denial of service via algorithmic complexity attacks”:

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

Attack examples:
Perl programming language,
Squid web cache, etc.
No attack on `dnscache`.

2011 (28C3), Klink–Wälde, “Efficient denial of service attacks on web application platforms”; oCERT advisory 2011–003:

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

SENIX Security
um, Crosby–Wallach,
of service via
mic complexity attacks”:

sent a new class of
dwidth denial of service
. . . if each element
o the same bucket,
table will also
ate to a linked list.”

examples:
gramming language,
eb cache, etc.
ck on `dnscache`.

2011 (28C3), Klink–Wälde,
“Efficient denial of service attacks
on web application platforms”;
oCERT advisory 2011–003:

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

Defendin

My favo
switch fr
to crit-b
Guarant
extra loc
“find ne

urity
oy–Wallach,
via
exity attacks":

w class of
nial of service
n element
e bucket,
also
nked list."

language,
etc.
`cache`.

2011 (28C3), Klink–Wälde, "Efficient denial of service attacks on web application platforms"; oCERT advisory 2011–003:

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

Defending against

My favorite solutio
switch from hash t
to crit-bit trees.
Guaranteed high s
extra lookup featu
"find next entry af

n,

cks":

vice

2011 (28C3), Klink–Wälde, "Efficient denial of service attacks on web application platforms"; oCERT advisory 2011–003:

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

Defending against hash floo

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

2011 (28C3), Klink–Wälde, "Efficient denial of service attacks on web application platforms"; oCERT advisory 2011–003:

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

Defending against hash flooding

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

2011 (28C3), Klink–Wälde, “Efficient denial of service attacks on web application platforms”; oCERT advisory 2011–003:

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

Defending against hash flooding

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

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

3C3), Klink–Wälde,
nt denial of service attacks
application platforms";
advisory 2011–003:

ck on `dnscache`,
rl, fixed Squid;
problems in Java, JRuby,
PHP 5, Python 2, Python
nius, Ruby, Apache
o, Apache Tomcat,
Glassfish, Jetty, Plone,
8 Javascript Engine.

Defending against hash flooding

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

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

Classic h
ℓ separa
for some

Store str
where i

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

(What if
Rehash:

k–Wälde,
f service attacks
n platforms”;
011–003:

cache,
quid;
in Java, JRuby,
ython 2, Python
Apache
e Tomcat,
Jetty, Plone,
pt Engine.

Defending against hash flooding

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

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

Classic hash table:
ℓ separate linked li
for some $\ell \in \{1, 2$

Store string s in li
where $i = H(s)$ m

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

(What if n becom
Rehash: replace ℓ

ttacks

s”;

Ruby,

Python

e,

Defending against hash flooding

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

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

Classic hash table: ℓ separate linked lists for some $\ell \in \{1, 2, 4, 8, 16, .$

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.)

Defending against hash flooding

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

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

Classic hash table:
ℓ separate linked lists
for some $\ell \in \{1, 2, 4, 8, 16, \ldots\}$.

Store string s in list $\#i$
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.)

ng against hash flooding

rite solution:
rom hash tables
it trees.
eed high speed +
okup features such as
xt entry after x."

n tables
eived as being
faster, *simpler*
her data structures.
protect hash tables?

Classic hash table:
ℓ separate linked lists
for some $\ell \in \{1, 2, 4, 8, 16, \ldots\}$.

Store string s in list $\#i$
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.)

Basic ha
attacker
$s_1, \ldots, s$
$\cdots = H($

Then all
in the sa
Linked li

hash flooding

on:

tables

peed +

res such as

fter x."

eing

npler

ructures.

sh tables?

Classic hash table:

ℓ separate linked lists
for some $\ell \in \{1, 2, 4, 8, 16, \ldots\}$.

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.)

Basic hash floodin

attacker provides s

$s_1, \ldots, s_n$ with $H($

$\cdots = H(s_n)$ mod

Then all strings ar

in the same linked

Linked list become

ding

ns

Classic hash table:
ℓ separate linked lists
for some $\ell \in \{1, 2, 4, 8, 16, \ldots\}$.

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.)

Basic hash flooding:
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 slo

Classic hash table:
ℓ separate linked lists
for some $\ell \in \{1, 2, 4, 8, 16, \ldots\}$.

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.)

Basic hash flooding:
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.

Classic hash table:
ℓ separate linked lists
for some $\ell \in \{1, 2, 4, 8, 16, \ldots\}$.

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.)

Basic hash flooding:
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.

Classic hash table:
ℓ separate linked lists
for some $\ell \in \{1, 2, 4, 8, 16, \ldots\}$.

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.)

Basic hash flooding:
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.

hash table:
te linked lists
e $\ell \in \{1, 2, 4, 8, 16, \ldots\}$.

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.)

Basic hash flooding:
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.

Non-solu
Use SH
SHA-3 i

:

ists

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

st $\#i$

od ℓ.

table,

ries

linked lists,

erations.

es too big?

by 2ℓ.)

Basic hash flooding:
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.

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

$\ldots\}$.

s,

?

Basic hash flooding:
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.

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

Basic hash flooding:
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.

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

Basic hash flooding: 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.

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

Why this is bad: $H(s) \bmod \ell$ 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.

sh flooding:
provides strings
s_n with $H(s_1) \bmod \ell =$
$(s_n) \bmod \ell$.

strings are stored
ame linked list.
ist becomes very slow.

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

lementors are unhappy:
tion throws away the
y of hash tables.

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

Why this is bad: $H(s) \bmod \ell$ 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.

1977, Ca
classes o
paper gi
average
storage a
algorithr
of hash
class of

2003 Cr
About 6
$H(m_1, m$
$m_1 k_1 +$
$k_1, k_2, .$
This is '

g:
strings
(s_1) mod ℓ $=$
ℓ.

re stored
list.
es very slow.

linked list
cture,
g.

are unhappy:
ws away the
tables.

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

Why this is bad: $H(s)$ mod ℓ
is not collision-resistant.

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

1977, Carter–Weg
classes of hash fun
paper gives an *inp*
average linear tim
storage and retrie
algorithm makes a
of hash function fr
class of hash funct

2003 Crosby–Wall
About 6 cycles/by
$H(m_1, m_2, \ldots, m$
$m_1k_1 + m_2k_2 + \cdot$
$k_1, k_2, \ldots, k_{12}$: ra
This is “provably s

$\ell =$

w.

py:

e

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

Why this is bad: $H(s) \bmod \ell$
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.

1977, Carter–Wegman, "Un
classes of hash functions": "
paper gives an *input indepe*
average linear time algorithn
storage and retrieval on keys
algorithm makes a random c
of hash function from a suit
class of hash functions."

2003 Crosby–Wallach:
About 6 cycles/byte on P2 f
$H(m_1, m_2, \ldots, m_{12}) =$
$m_1k_1 + m_2k_2 + \cdots + m_{12}k$
$k_1, k_2, \ldots, k_{12}$: random, 20-
This is "provably secure"!

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

Why this is bad: $H(s) \bmod \ell$ 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.

1977, Carter–Wegman, "Universal classes of hash functions": "This paper gives an *input independent* average linear time algorithm for storage and retrieval on keys. The algorithm makes a random choice of hash function from a suitable class of hash functions."

2003 Crosby–Wallach:
About 6 cycles/byte on P2 for
$H(m_1, m_2, \ldots, m_{12}) =$
$m_1k_1 + m_2k_2 + \cdots + m_{12}k_{12}$.
$k_1, k_2, \ldots, k_{12}$: random, 20-bit.
This is "provably secure"!

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

s is bad: $H(s) \bmod \ell$
ollision-resistant.

ll: e.g., $\ell = 2^{20}$.
ter how strong H is,
can easily compute
od 2^{20} for many s
multicollisions.

1977, Carter–Wegman, "Universal classes of hash functions": "This paper gives an *input independent* average linear time algorithm for storage and retrieval on keys. The algorithm makes a random choice of hash function from a suitable class of hash functions."

2003 Crosby–Wallach:
About 6 cycles/byte on P2 for
$H(m_1, m_2, \ldots, m_{12}) =$
$m_1k_1 + m_2k_2 + \cdots + m_{12}k_{12}$.
$k_1, k_2, \ldots, k_{12}$: random, 20-bit.
This is "provably secure"!

We don'
The secu
assumes
is *indepe*

-resistant!

$H(s)$ mod ℓ
istant.

$= 2^{20}$.
rong H is,
compute
many s
ons.

1977, Carter–Wegman, “Universal classes of hash functions”: “This paper gives an *input independent* average linear time algorithm for storage and retrieval on keys. The algorithm makes a random choice of hash function from a suitable class of hash functions.”

2003 Crosby–Wallach:
About 6 cycles/byte on P2 for
$H(m_1, m_2, \ldots, m_{12}) =$
$m_1 k_1 + m_2 k_2 + \cdots + m_{12} k_{12}$.
$k_1, k_2, \ldots, k_{12}$: random, 20-bit.
This is “provably secure”!

We don’t recomm
The security guara
assumes that rand
is *independent* of

ℓ

1977, Carter–Wegman, "Universal classes of hash functions": "This paper gives an *input independent* average linear time algorithm for storage and retrieval on keys. The algorithm makes a random choice of hash function from a suitable class of hash functions."

2003 Crosby–Wallach:
About 6 cycles/byte on P2 for
$H(m_1, m_2, \ldots, m_{12}) =$
$m_1 k_1 + m_2 k_2 + \cdots + m_{12} k_{12}$.
$k_1, k_2, \ldots, k_{12}$: random, 20-bit.
This is "provably secure"!

We don't recommend this. The security guarantee assumes that randomness is *independent* of inputs.

1977, Carter–Wegman, "Universal classes of hash functions": "This paper gives an *input independent* average linear time algorithm for storage and retrieval on keys. The algorithm makes a random choice of hash function from a suitable class of hash functions."

2003 Crosby–Wallach:
About 6 cycles/byte on P2 for
$H(m_1, m_2, \ldots, m_{12}) =$
$m_1 k_1 + m_2 k_2 + \cdots + m_{12} k_{12}$.
$k_1, k_2, \ldots, k_{12}$: random, 20-bit.
This is "provably secure"!

We don't recommend this.
The security guarantee assumes that randomness is *independent* of inputs.

1977, Carter–Wegman, “Universal classes of hash functions”: “This paper gives an *input independent* average linear time algorithm for storage and retrieval on keys. The algorithm makes a random choice of hash function from a suitable class of hash functions.”

2003 Crosby–Wallach:
About 6 cycles/byte on P2 for
$H(m_1, m_2, \ldots, m_{12}) =$
$m_1 k_1 + m_2 k_2 + \cdots + m_{12} k_{12}$.
$k_1, k_2, \ldots, k_{12}$: random, 20-bit.
This is “provably secure”!

We don’t recommend this.
The security guarantee
assumes that randomness
is *independent* of inputs.

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

2005 Crosby: Maybe trouble for
any function with a short key,
and for $m_1 k_1 + m_2 k_2 + \cdots$.

arter–Wegman, “Universal
of hash functions”: “This
ves an *input independent*
linear time algorithm for
and retrieval on keys. The
m makes a random choice
function from a suitable
hash functions.”

osby–Wallach:
cycles/byte on P2 for
$n_2, \ldots, m_{12}) =$
$m_2 k_2 + \cdots + m_{12} k_{12}$.
$. . , k_{12}$: random, 20-bit.
‘provably secure”!

We don’t recommend this.
The security guarantee
assumes that randomness
is *independent* of inputs.

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

2005 Crosby: Maybe trouble for
any function with a short key,
and for $m_1 k_1 + m_2 k_2 + \cdots$.

Even wo
(e.g., an
prints ta
leak mor

Some ap
$H(s)$ mo

man, "Universal
nctions": "This
ut independent
e algorithm for
val on keys. The
random choice
rom a suitable
tions."

ach:
te on P2 for
$_{12}) =$
$\cdots + m_{12}k_{12}$.
ndom, 20-bit.
secure"!

We don't recommend this.
The security guarantee
assumes that randomness
is *independent* of inputs.

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

2005 Crosby: Maybe trouble for
any function with a short key,
and for $m_1k_1 + m_2k_2 + \cdots$.

Even worse: Some
(e.g., any applicat
prints table withou
leak more informa

Some applications
$H(s)$ mod ℓ, or ev

iversal
"This
ndent
n for
s. The
choice
able

for

c_{12}.
-bit.

We don't recommend this. The security guarantee assumes that randomness is *independent* of inputs.

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

2005 Crosby: Maybe trouble for any function with a short key, and for $m_1k_1 + m_2k_2 + \cdots$.

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

Some applications simply pr
$H(s)$ mod ℓ, or even $H(s)$.

We don't recommend this. The security guarantee assumes that randomness is *independent* of inputs.

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

2005 Crosby: Maybe trouble for any function with a short key, and for $m_1k_1 + m_2k_2 + \cdots$.

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

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

We don't recommend this. The security guarantee assumes that randomness is *independent* of inputs.

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

2005 Crosby: Maybe trouble for any function with a short key, and for $m_1k_1 + m_2k_2 + \cdots$.

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

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

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

We don't recommend this. The security guarantee assumes that randomness is *independent* of inputs.

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

2005 Crosby: Maybe trouble for any function with a short key, and for $m_1k_1 + m_2k_2 + \cdots$.

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

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

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

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

t recommend this.
urity guarantee
that randomness
endent of inputs.

ed hash flooding:
, server timing
t hash collisions;
ut the hash key;
nputs accordingly.

osby: Maybe trouble for
ction with a short key,
$m_1 k_1 + m_2 k_2 + \cdots$.

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

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

We recommend choosing H as a strong PRF. ⇒ Seeing many H values is of no use in predicting others.

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

<u>The imp</u>

Crypto c
Wow, M
only abo
Let's use

end this.
antee
omness
inputs.

oding:
ming
lisions;
n key;
ordingly.

ybe trouble for
a short key,
$_2k_2 + \cdots$.

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

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

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

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

The importance of

Crypto design, 199
Wow, MD5 is real
only about 5 cycle
Let's use HMAC-M

e for
ey,
.

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

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

We recommend choosing H as a strong PRF. ⇒ Seeing many H values is of no use in predicting others.

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

The importance of overhead

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

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

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

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

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

The importance of overhead

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

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

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

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

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

The importance of overhead

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

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

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

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

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

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

The importance of overhead

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

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

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

orse: Some applications
y application that
ble without sorting)
re information about H.

pplications simply print
od ℓ, or even $H(s)$.

mmend choosing H
ong PRF. ⇒
many H values
use in predicting others.

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

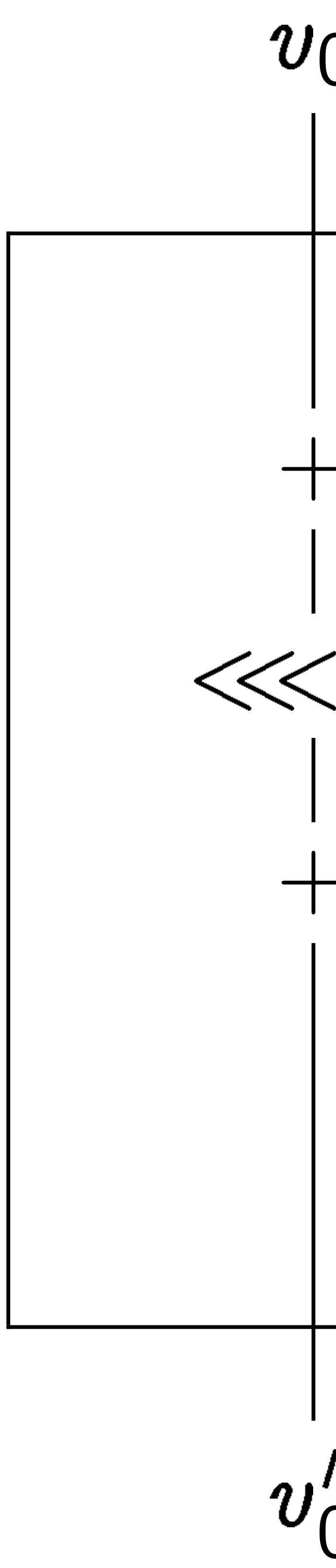
The importance of overhead

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

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

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

SipRoun



This is S
SipHash

e applications
ion that
ut sorting)
tion about H.

simply print
en $H(s)$.

hoosing H
$\Rightarrow$
lues
dicting others.

n in $H(s)$ mod ℓ
$n\ell \approx n^2$ inputs.

communication.

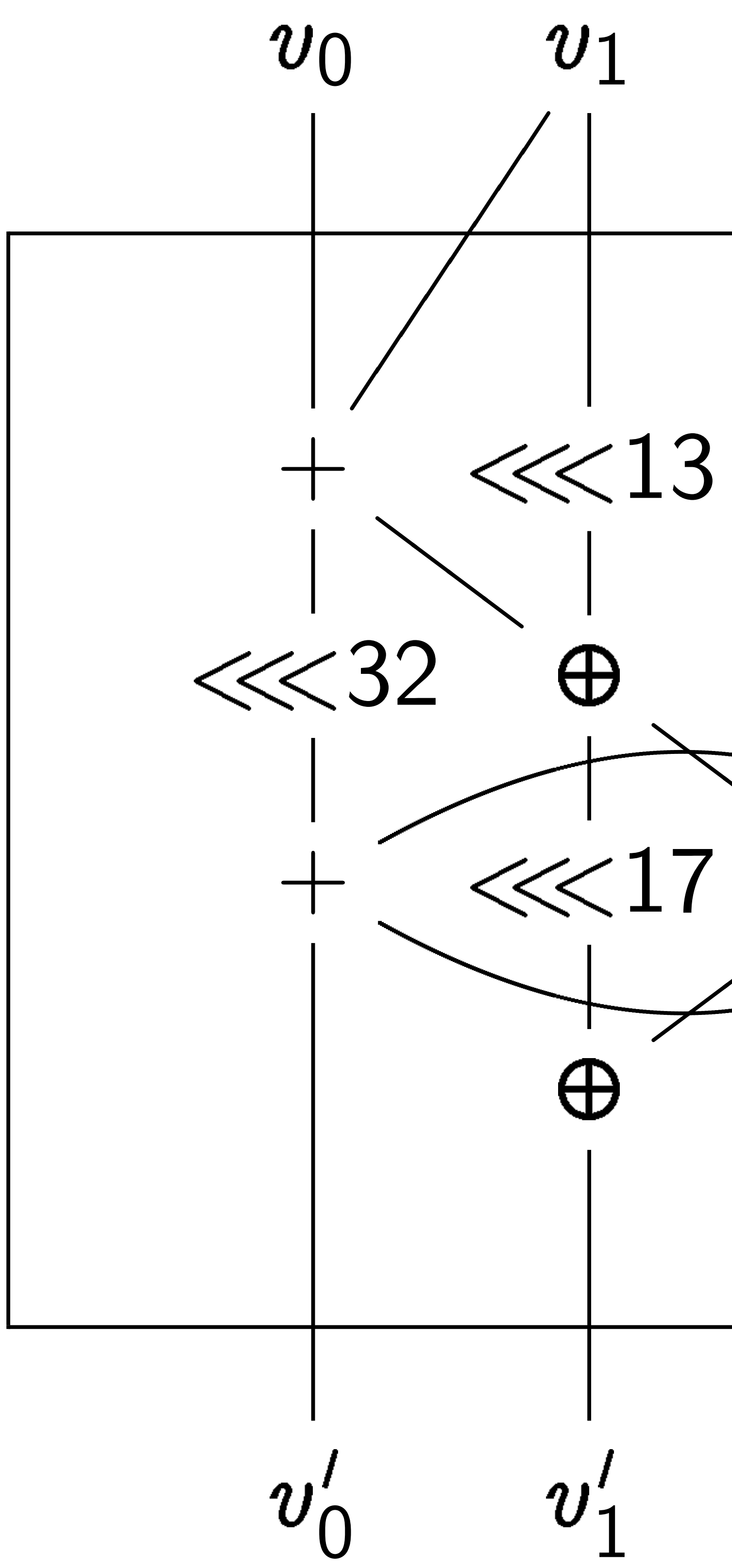
The importance of overhead

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

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

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

SipRound and Sip



This is SipRound.
SipHash-2-4 applie

ons

H.

int

hers.

mod ℓ

nputs.

cation.

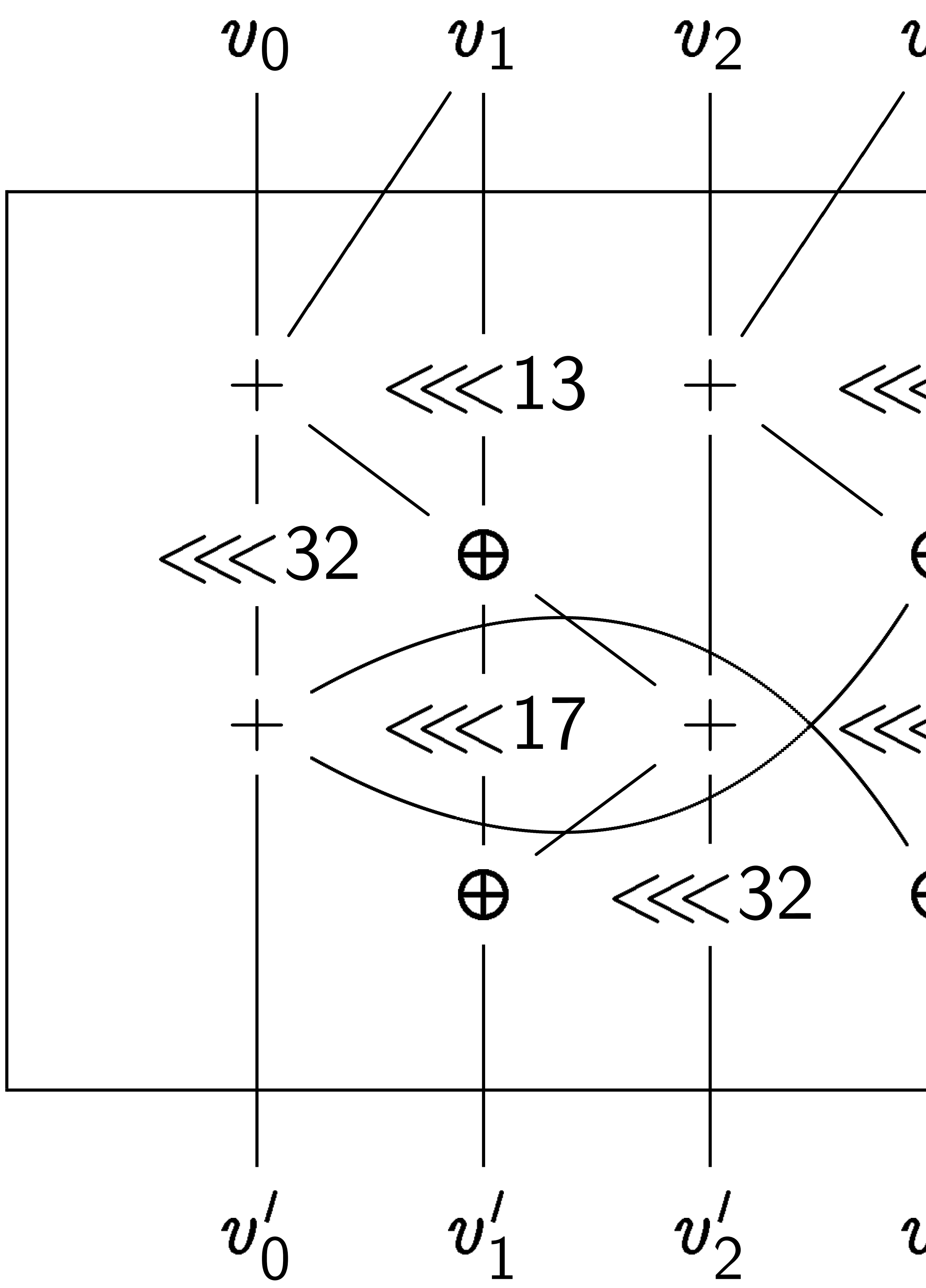
The importance of overhead

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

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

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

SipRound and SipHash



This is SipRound. Next pag
SipHash-2-4 applied to 16 b

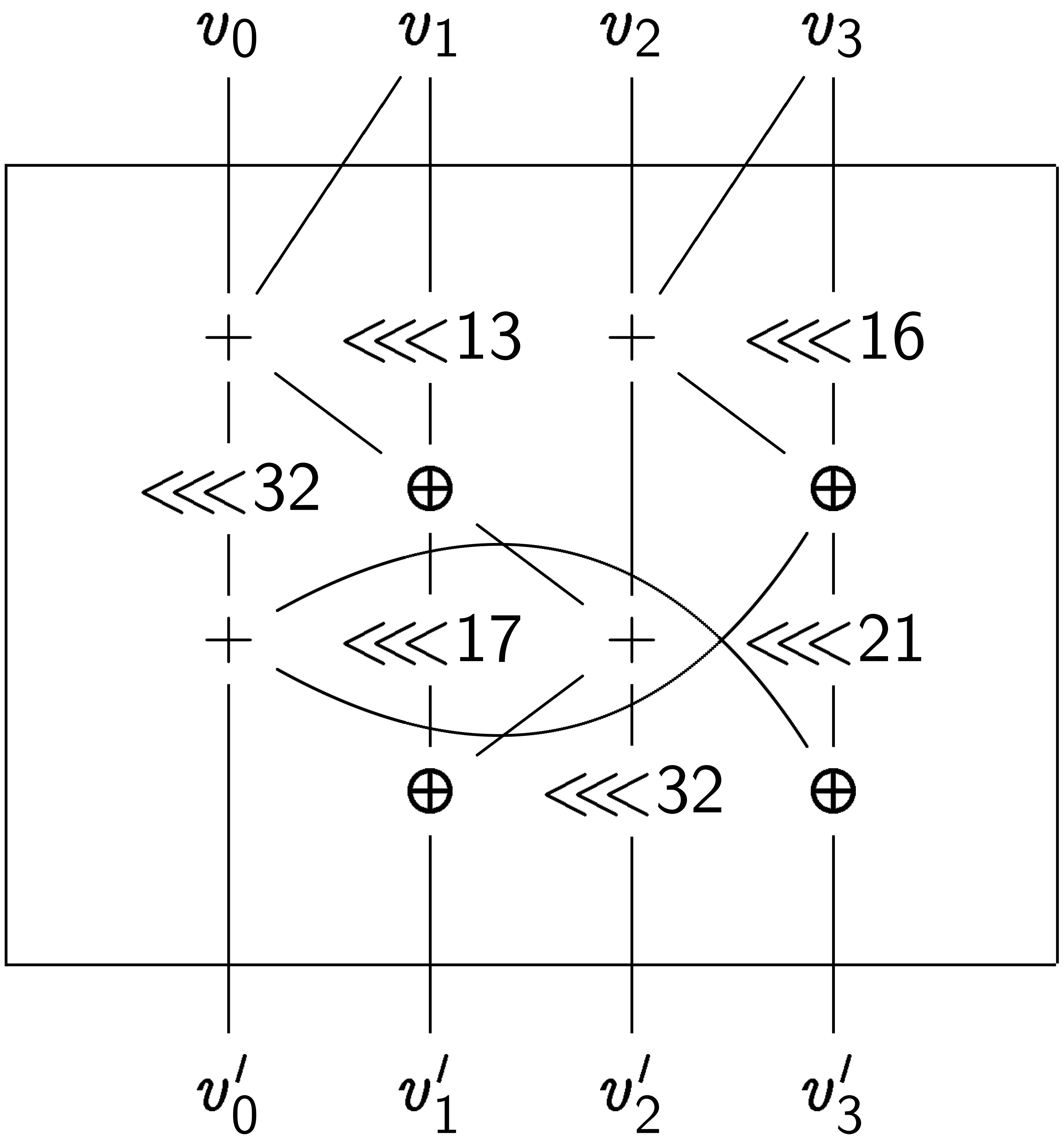
The importance of overhead

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

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

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

SipRound and SipHash



This is SipRound. Next page:
SipHash-2-4 applied to 16 bytes.

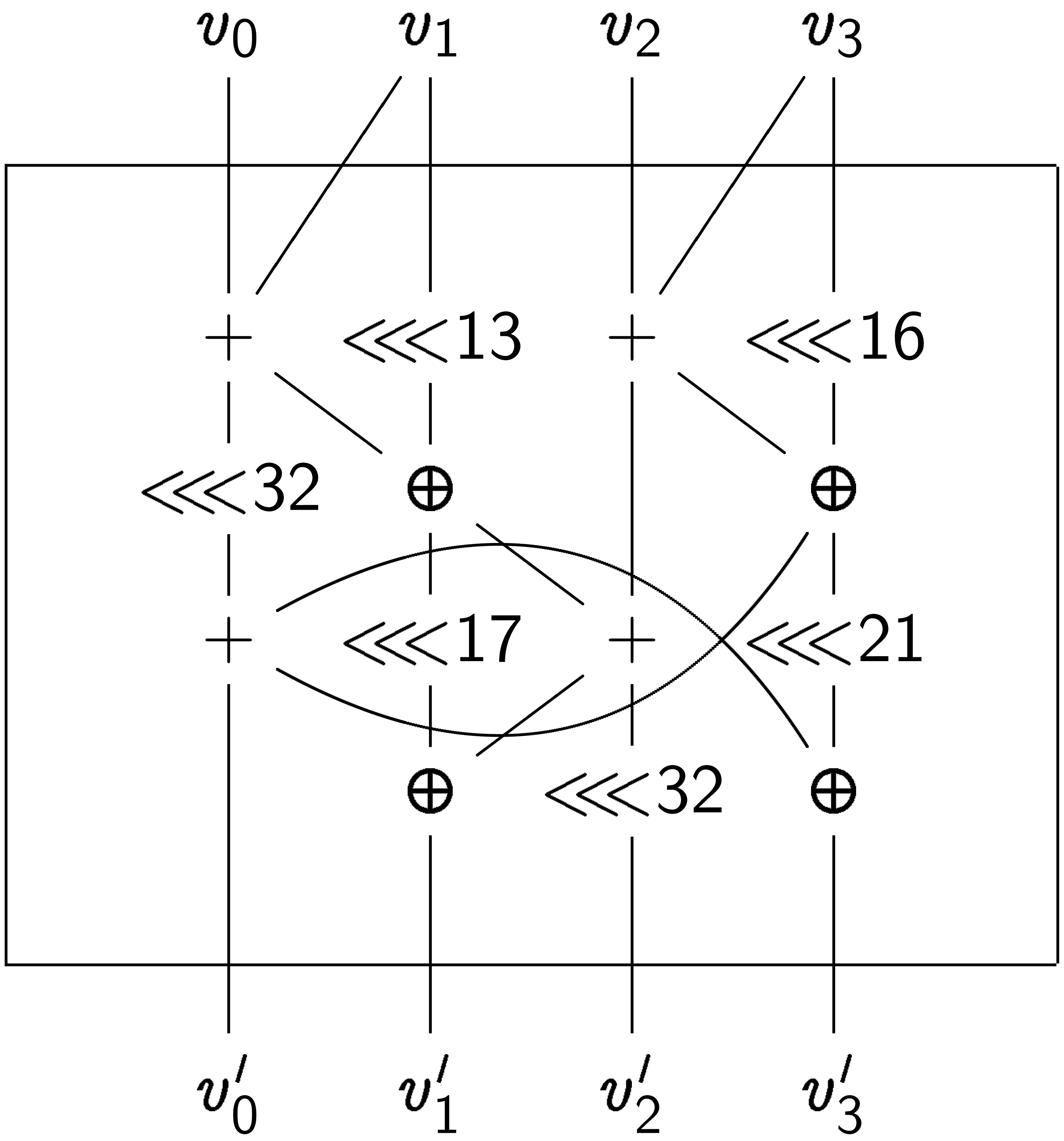
portance of overhead

design, 1990s:
D5 is really fast;
out 5 cycles/byte.
e HMAC-MD5 as a PRF.

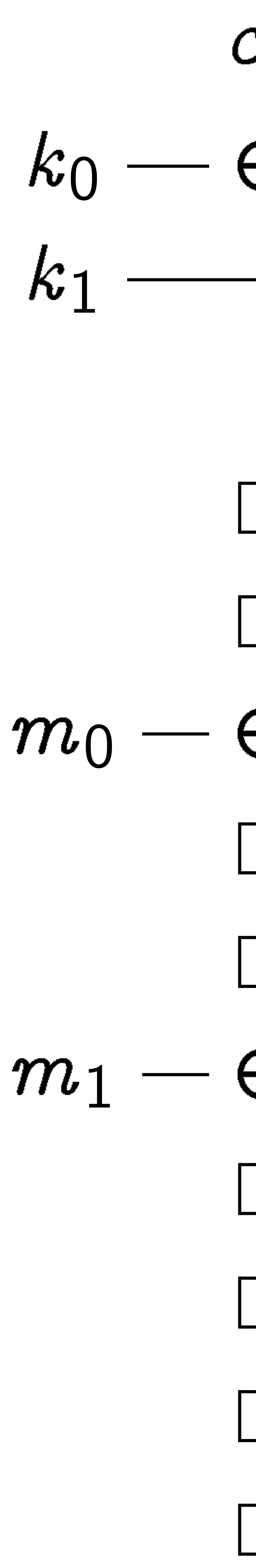
design, 2000s:
ers are even faster;
ch 1 or 2 cycles/byte.
5-AES, UMAC-AES, et al.

h-table perspective:
peed advertisements
for long inputs,
huge overheads!

SipRound and SipHash



This is SipRound. Next page:
SipHash-2-4 applied to 16 bytes.



f overhead

90s:

ly fast;

es/byte.

MD5 as a PRF.

00s:

en faster;

ycles/byte.

MAC-AES, et al.

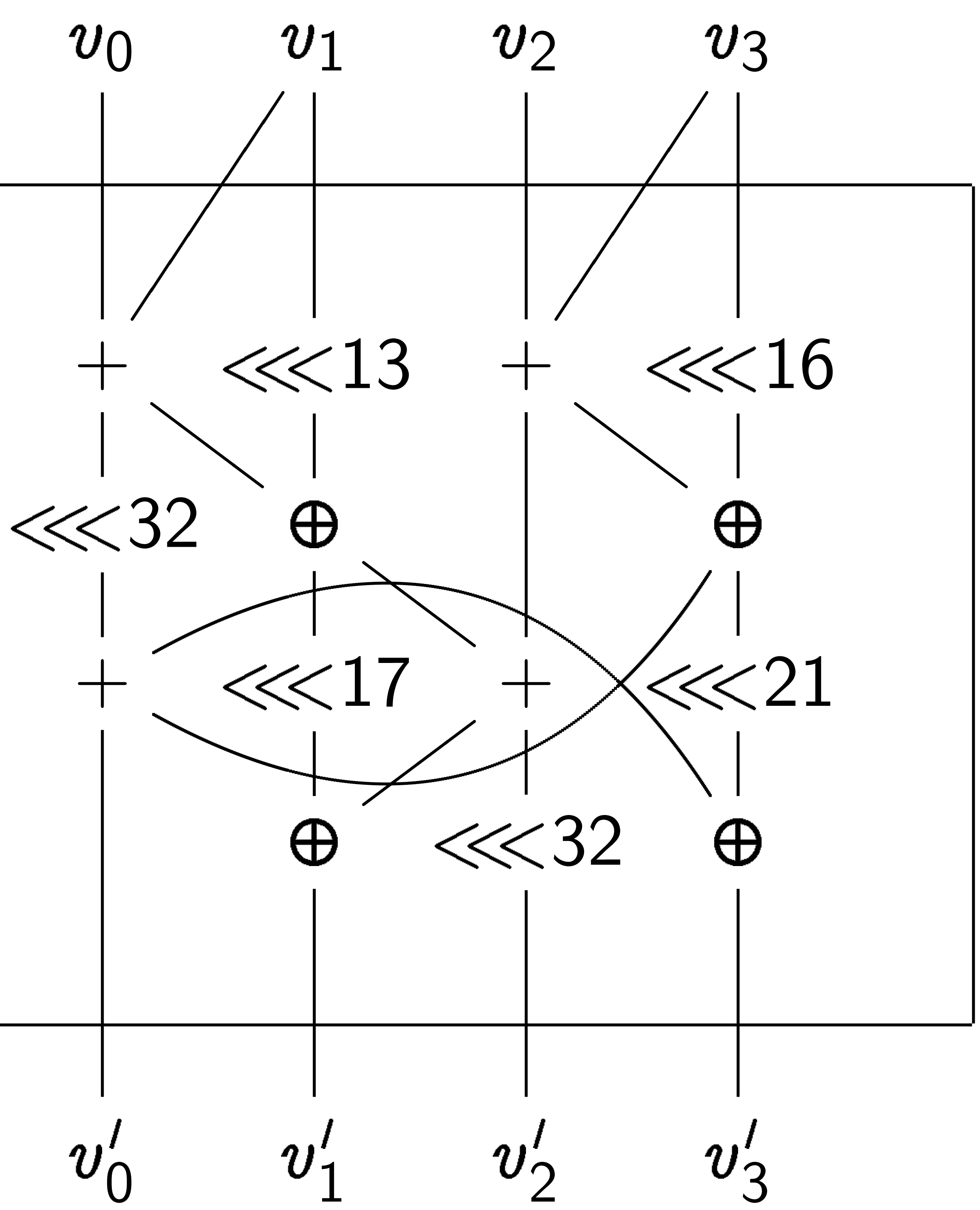
rspective:

rtisements

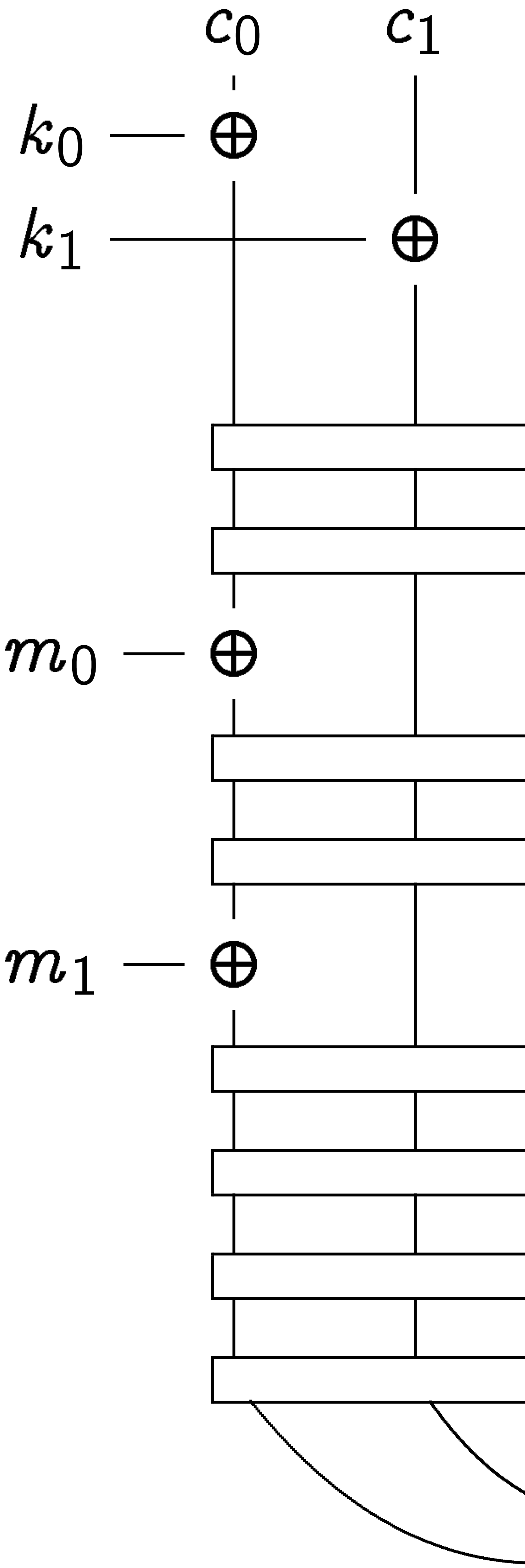
nputs,

rheads!

SipRound and SipHash



This is SipRound. Next page: SipHash-2-4 applied to 16 bytes.

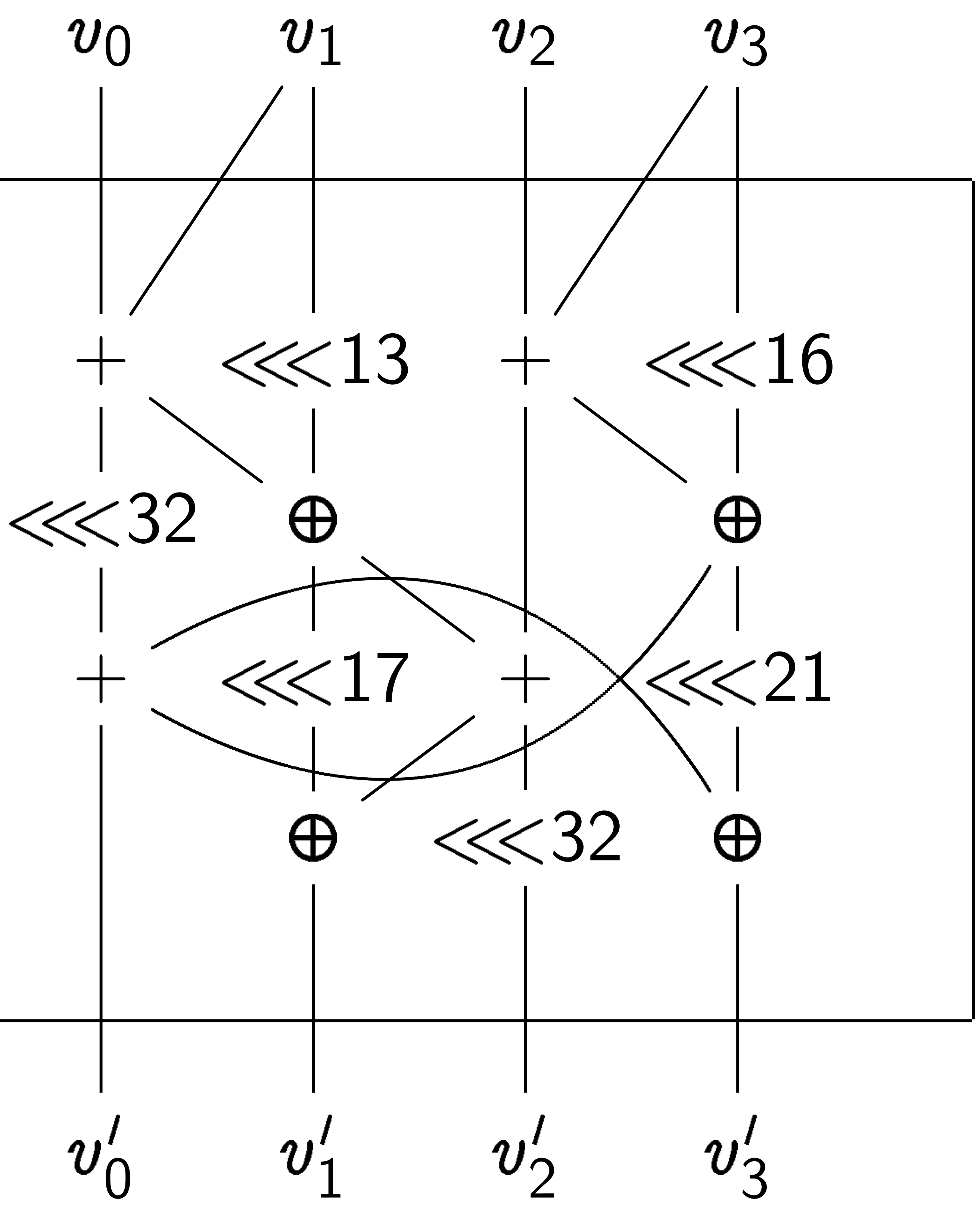


PRF.

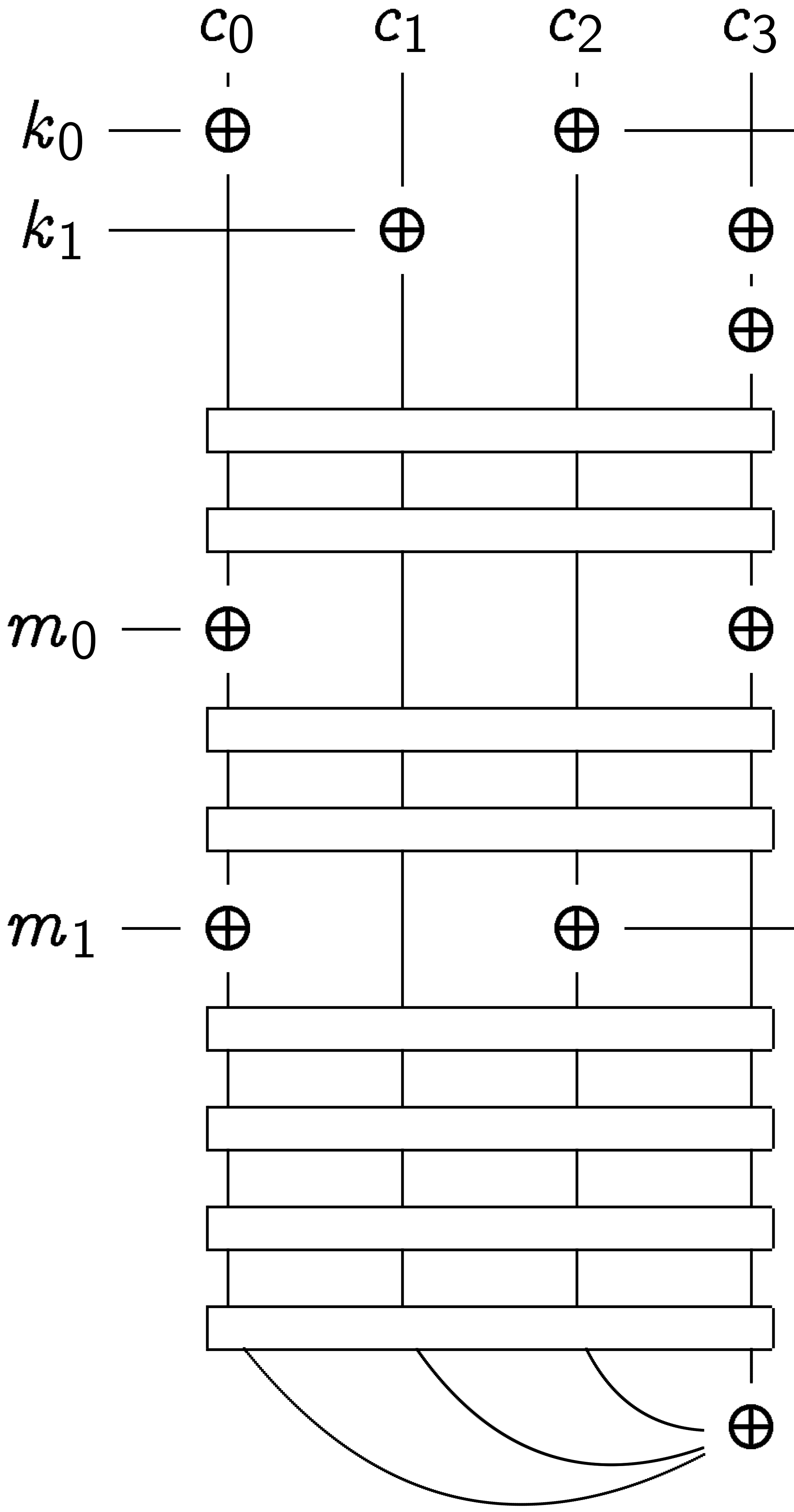
e.

. et al.

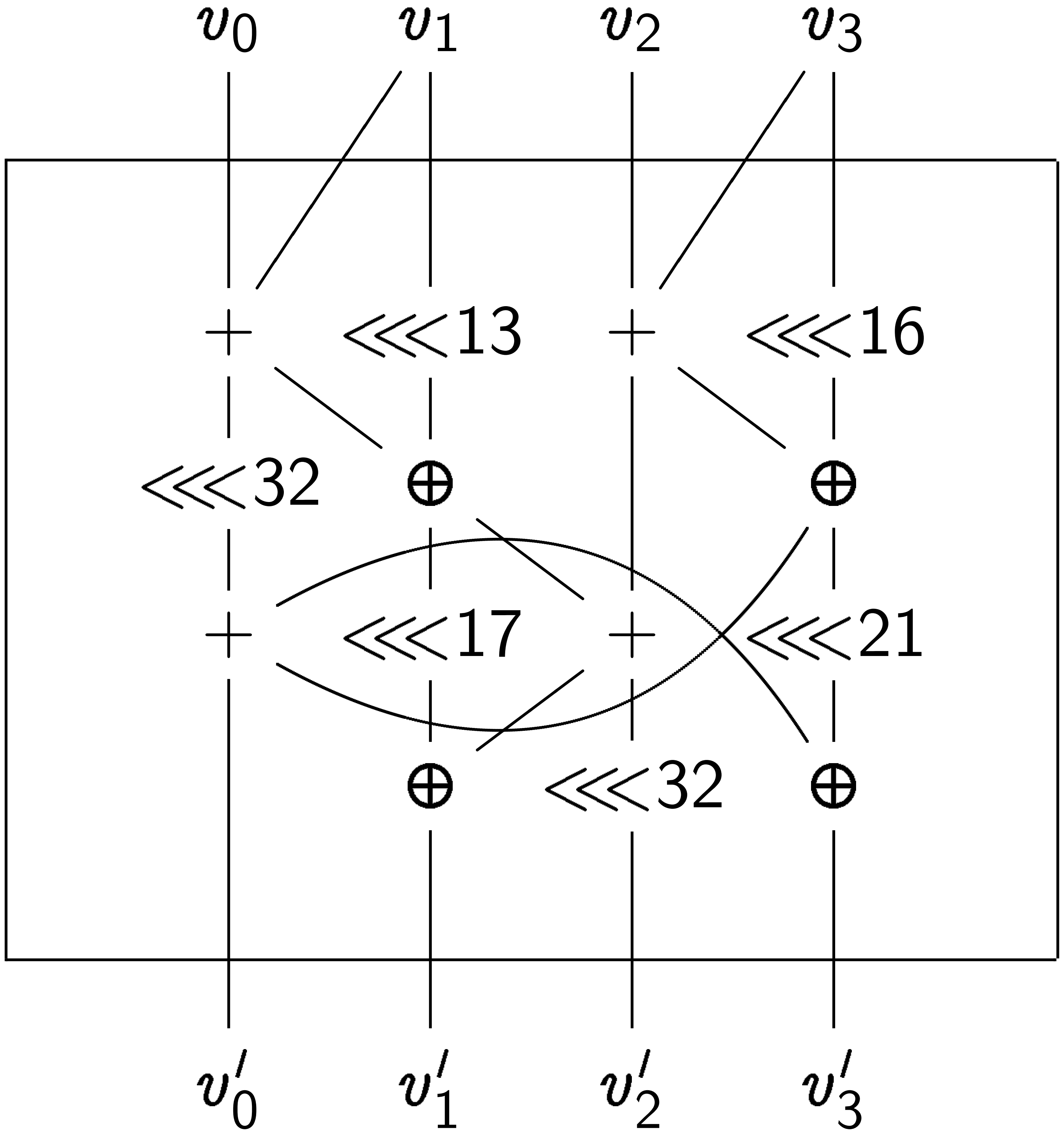
SipRound and SipHash



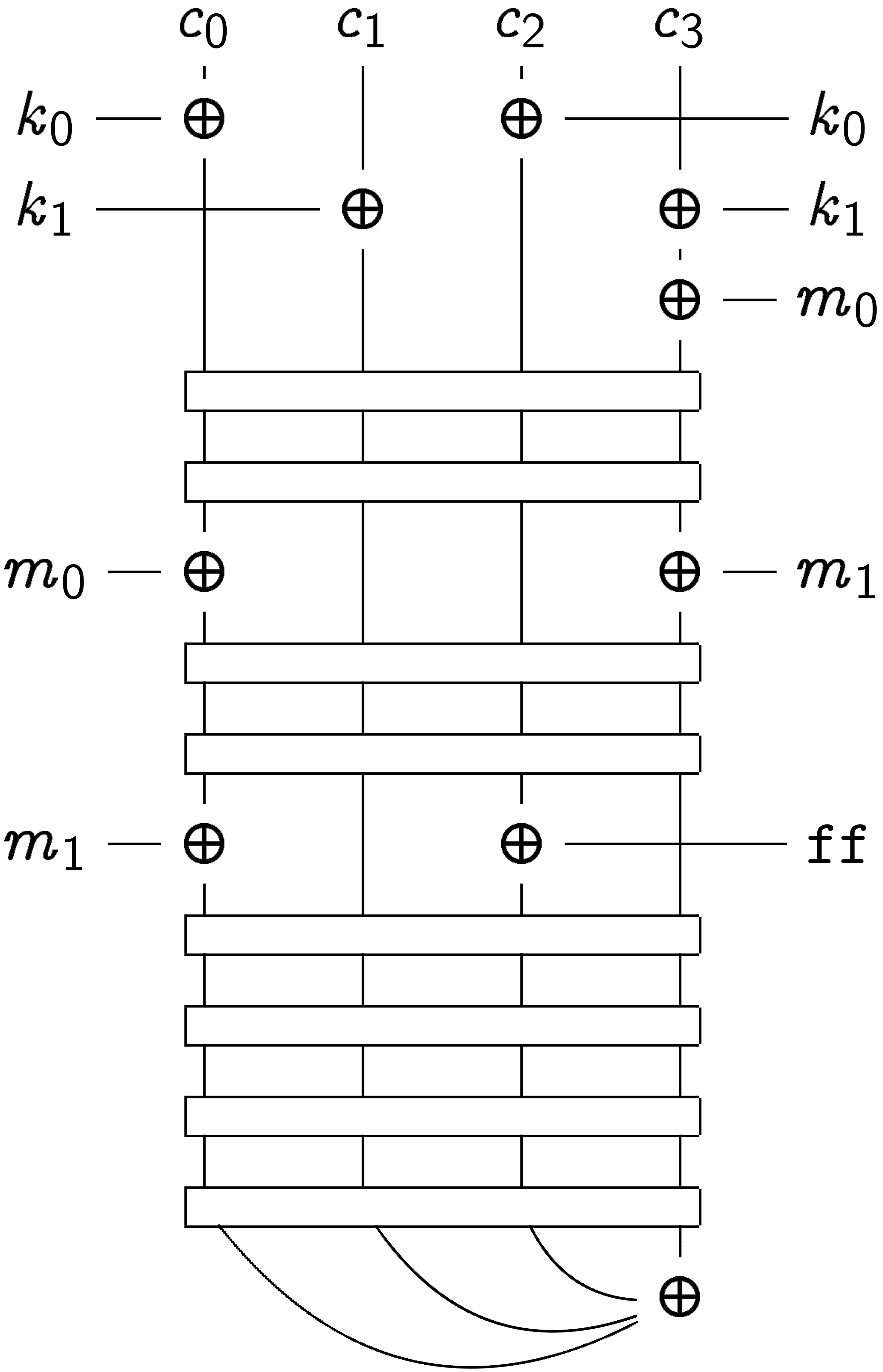
This is SipRound. Next page: SipHash-2-4 applied to 16 bytes.



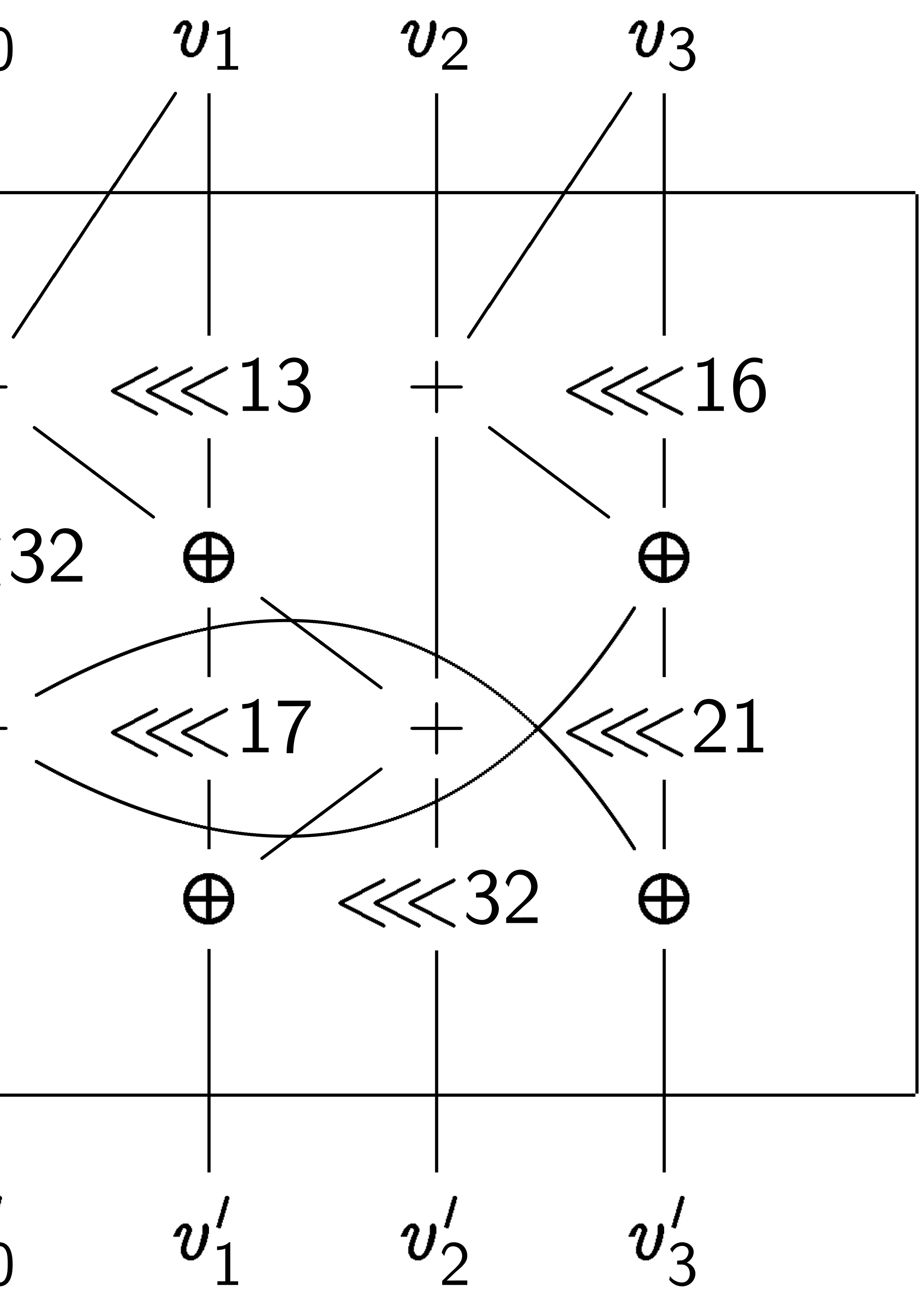
SipRound and SipHash



This is SipRound. Next page:
SipHash-2-4 applied to 16 bytes.

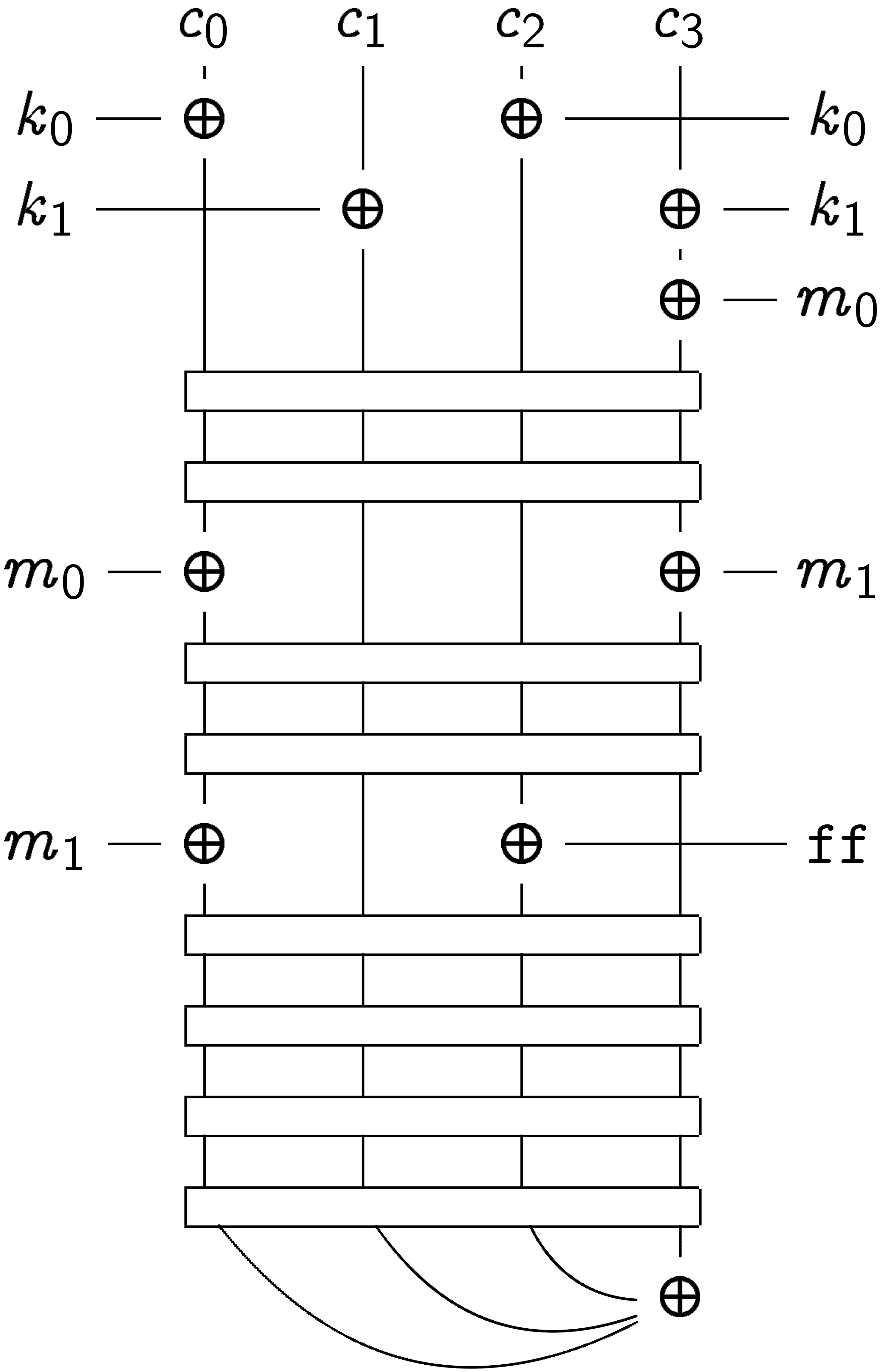


d and SipHash



SipRound. Next page:

-2-4 applied to 16 bytes.

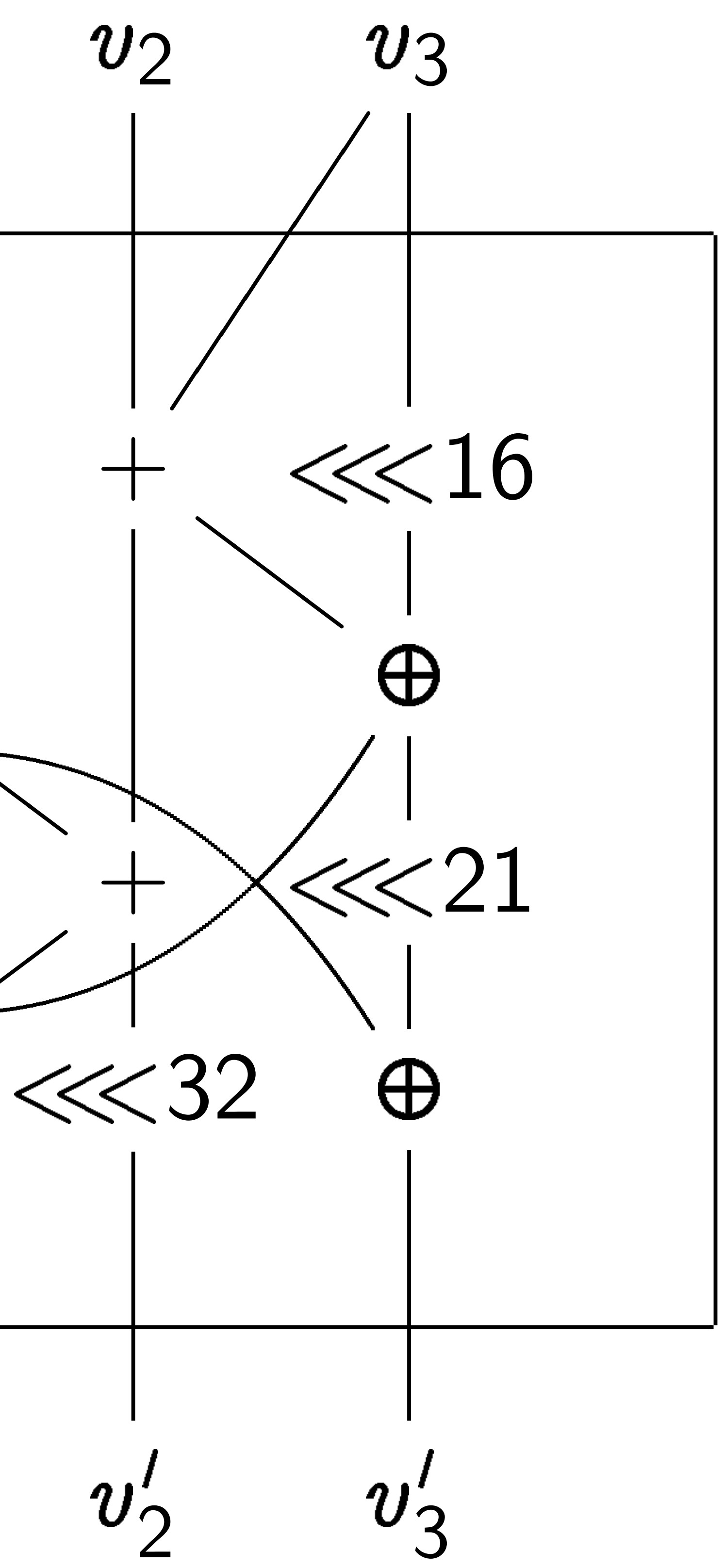


Much m

- Specif
- Discus
- Staten
- Design
- Prelim
- Bench
  1.65 c

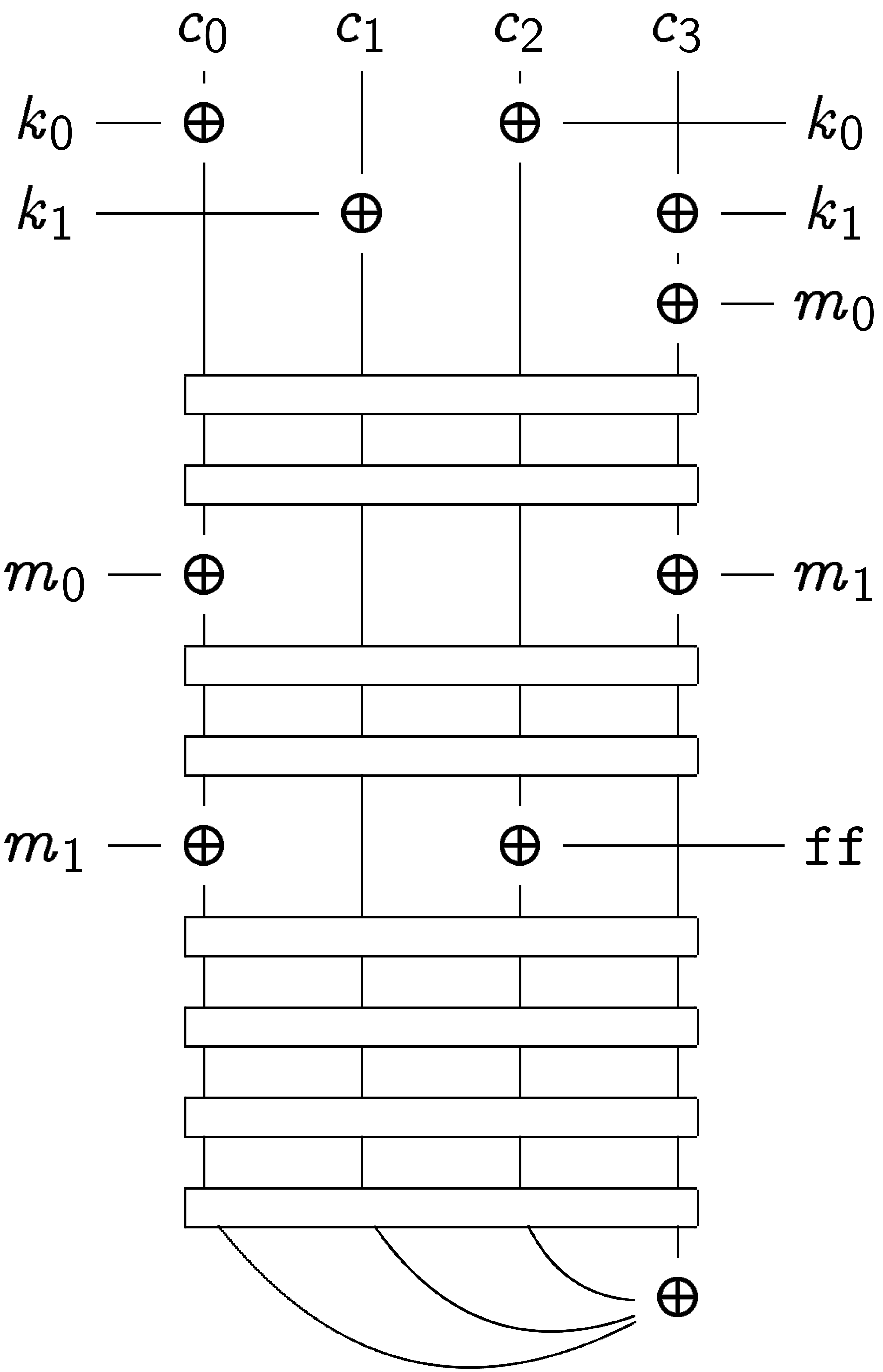
Positive
third-par
now usec
Redis, R

Hash



Next page:

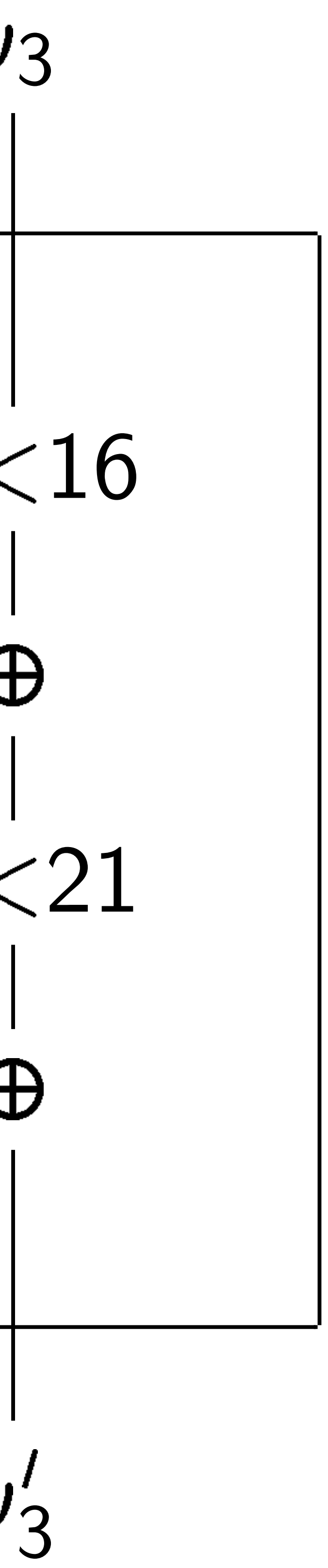
ed to 16 bytes.



Much more in pap

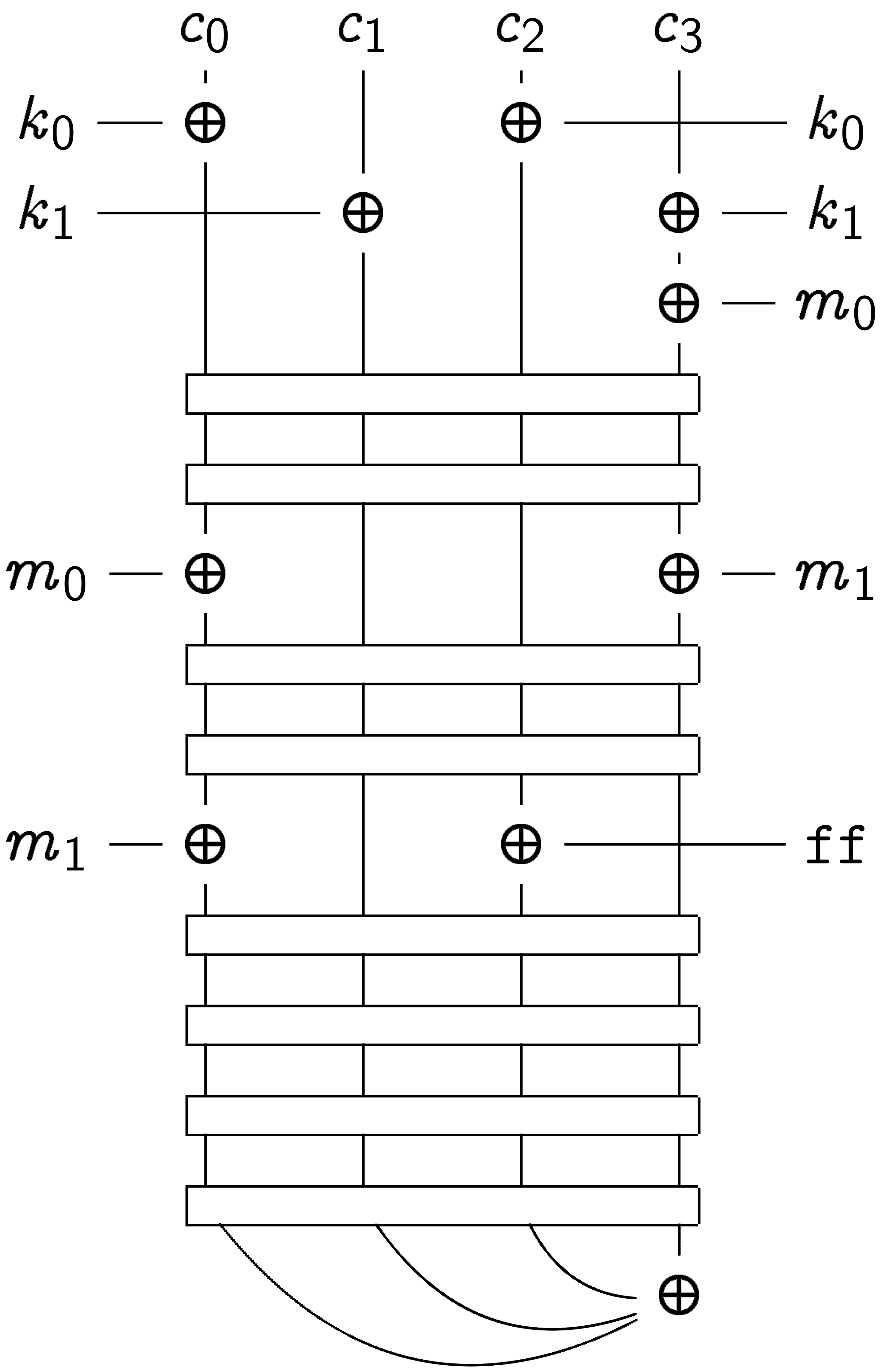
- Specification: pa
- Discussion of fea
- Statement of sec
- Design rationale
- Preliminary cryp
- Benchmarks. e.g
  1.65 cycles/byte

Positive SipHash r
third-party implem
now used for hash
Redis, Rust, Open



e:

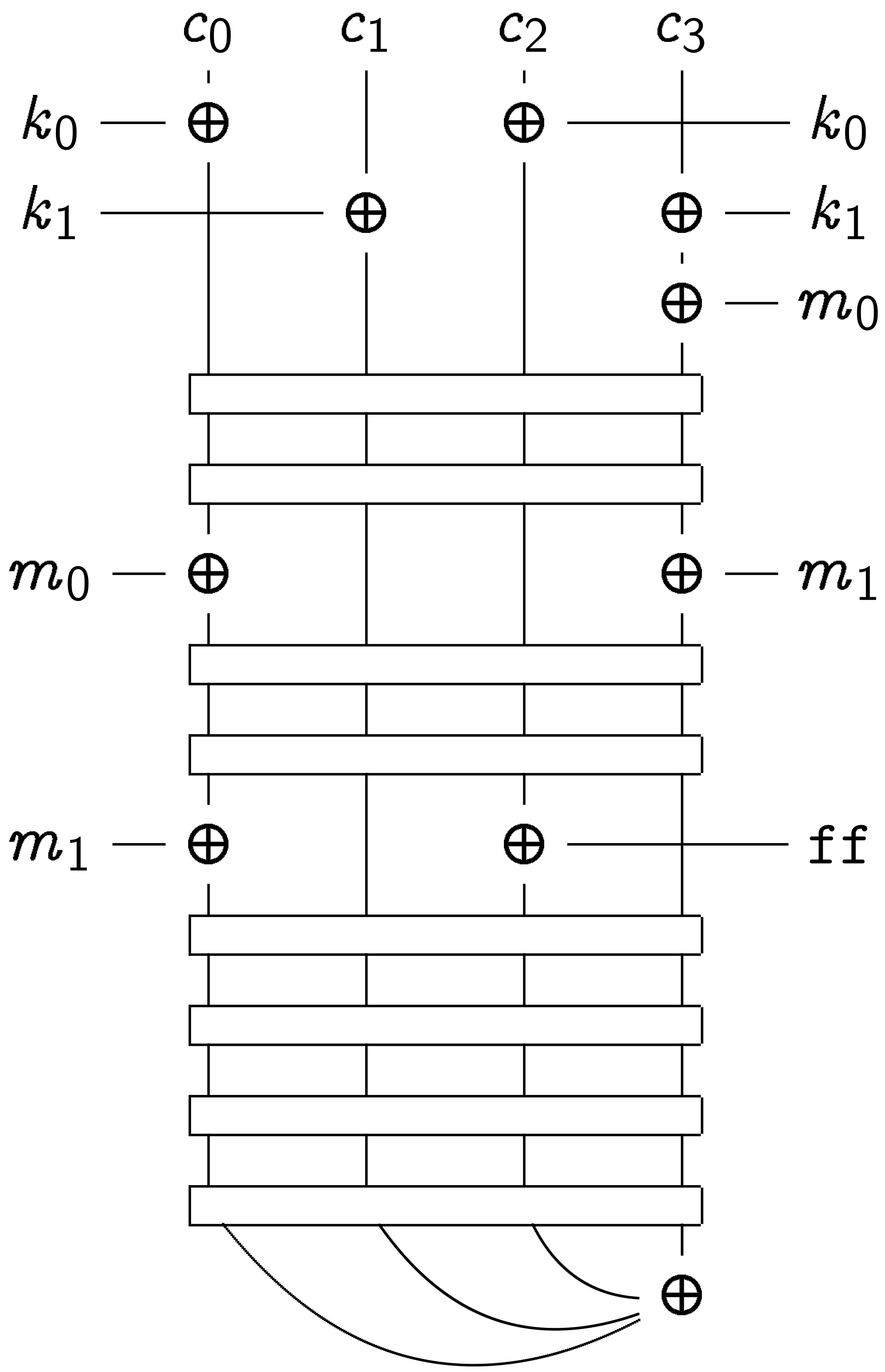
ytes.



Much more in paper:

- Specification: padding etc
- Discussion of features.
- Statement of security goal
- Design rationale and credi
- Preliminary cryptanalysis.
- Benchmarks. e.g. Ivy Brid
  1.65 cycles/byte + 27 cyc

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



Much more in paper:

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

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