D. J. Bernstein

University of Illinois at Chicago &
Technische Universiteit Eindhoven

1. New CAESAR project:
Competition for
Authenticated
Encryption:
Security,
Applicability,
Robustness

http://competitions.cr.yp.to

NIST has generously provided
some funding, 2013–2017.

2. Feistel modes redivivus:
wide-block online ciphers

Joint work with:
Mridul Nandi and Palash Sarkar,
Indian Statistical Institute,
Kolkata

---

3. SipHash:
a fast short-input PRF

Joint work with:
Jean-Philippe Aumasson,
Kudelski Security (NAGRA)

https://131002.net/siphash/

Feistel motivation:

Tor is an Internet service
that tries to provide
low-latency anonymity for
human-rights activists,
police investigators,
Chinese Internet users,
corporate browsing, etc.

Currently 3000 relays,
more than 2 GB/second,
estimated $>$500000 users/day.

Part of Nick Mathewson's talk "Cryptographic challenges in and around Tor" five days ago at Workshop on Real-World Cryptography in Stanford:

"We should replace our old relay cell protocol . . ." which currently uses AES-CTR to encrypt 509-byte blocks, something horrific to authenticate. Easy "tagging" attacks.

"A chained wide-block cipher seems like a much better idea!"

Many wide-block SPRP papers; some chaining-SPRP papers. Mathewson lists LIONESS, CMC, XCB, HCTR, XTS, XEX, HCH, TET for wide-block SPRP.

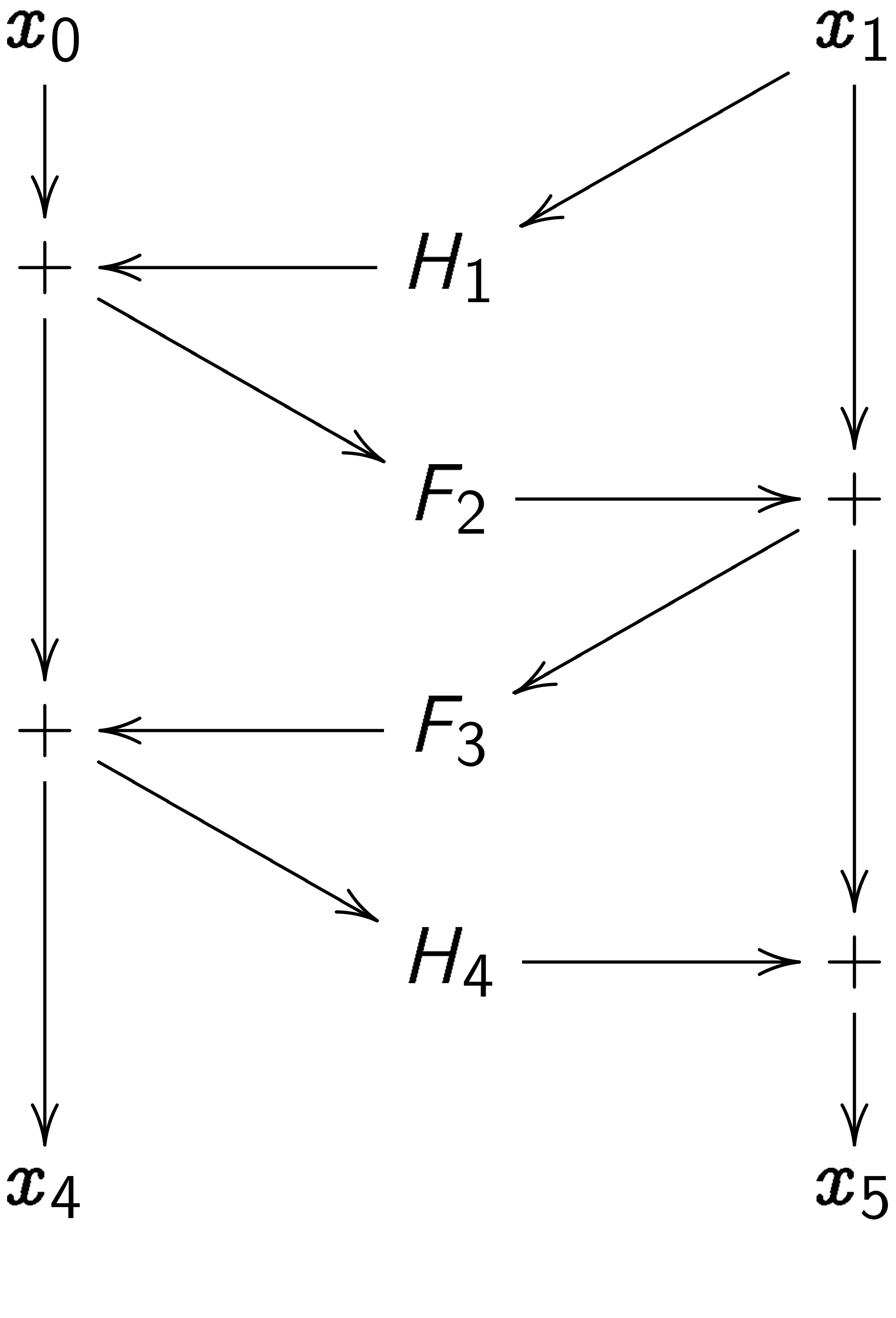
e.g. STOC 1997/JCrypt 1999 Naor–Reingold "LR revisited":

Encrypt $2n$-bit blocks using n-bit PRF in 4 Feistel rounds.

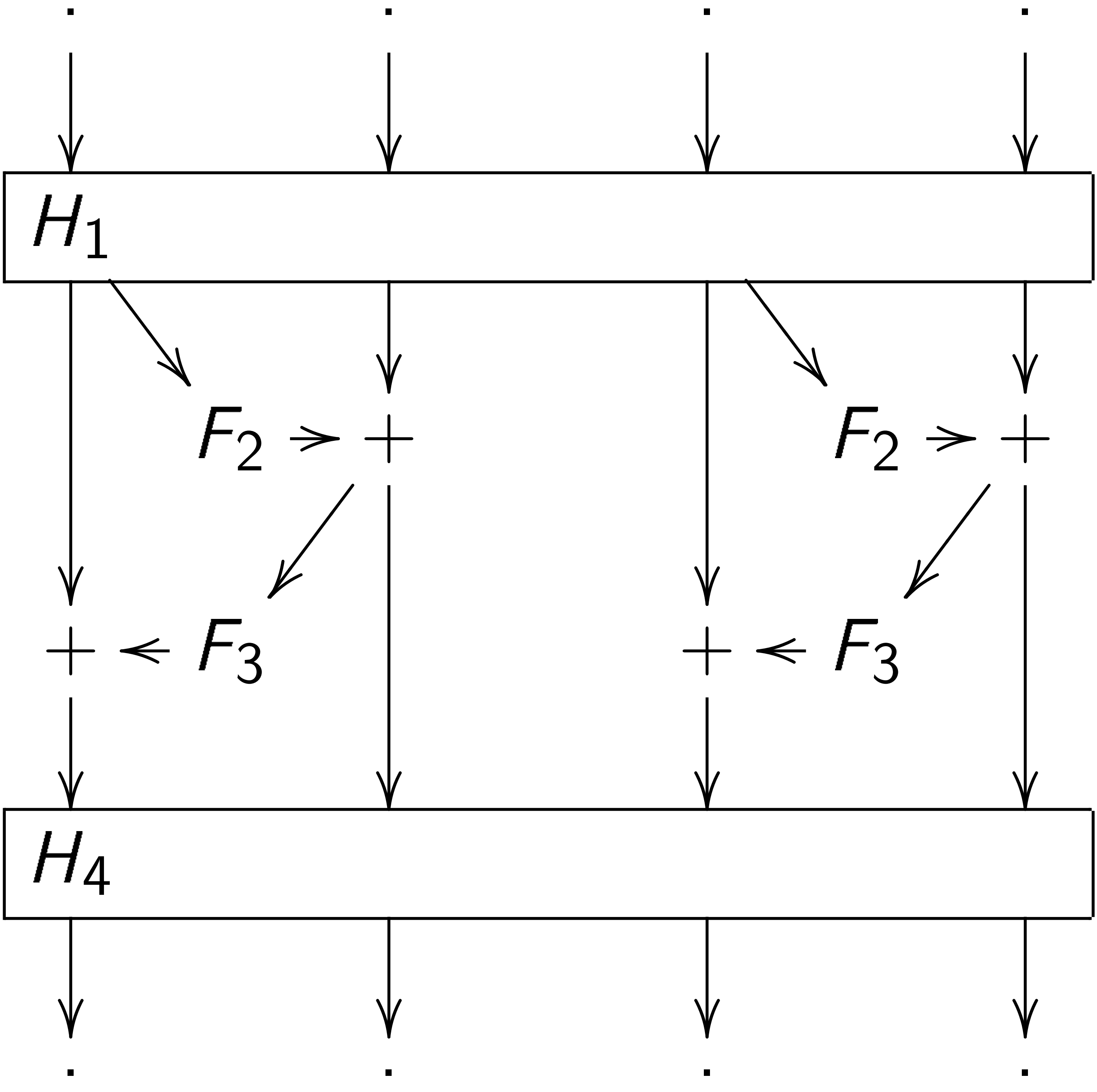
Replace PRF with AXU in first round (1996 Lucks), last round.

Section 7, "SPPE on many blocks": $2nb$-bit hash; $b\times$ECB on $2n$ bits; $2nb$-bit hash.

4-round HFFH Feistel:



Naor–Reingold for $b = 2$:



Our wide-block construction:

Our wide-block construction:

$$\begin{tikzcd}
x_0 \arrow[d] & & x_1 \arrow[dl] \arrow[dd] \\
+ \arrow[dr] \arrow[dd] & H_1 \arrow[l] & \\
& F_2 \arrow[r] & + \arrow[dl] \arrow[dd] \\
+ \arrow[dr] \arrow[dd] & F_3 \arrow[l] & \\
& H_4 \arrow[r] & + \arrow[d] \\
x_4 & & x_5
\end{tikzcd}$$

Free to choose sizes
of left and right sides.
Probably want both sides big:
fast F (e.g. Salsa20)
is happiest with big output.

Free to choose sizes
of left and right sides.
Probably want both sides big:
fast F (e.g. Salsa20)
is happiest with big output.

How to build large-output H
using small-output h?

Free to choose sizes
of left and right sides.
Probably want both sides big:
fast F (e.g. Salsa20)
is happiest with big output.

How to build large-output H
using small-output h?
Define $H(x) = (h(x), 0, \ldots, 0)$.

Free to choose sizes
of left and right sides.
Probably want both sides big:
fast F (e.g. Salsa20)
is happiest with big output.

How to build large-output H
using small-output h?
Define $H(x) = (h(x), 0, \ldots, 0)$.

How to build large-input F
using small-input f?

Free to choose sizes
of left and right sides.
Probably want both sides big:
fast F (e.g. Salsa20)
is happiest with big output.

How to build large-output H
using small-output h?
Define $H(x) = (h(x), 0, \ldots, 0)$.

How to build large-input F
using small-input f?
Define $F(x) = f(h(x))$.

Free to choose sizes
of left and right sides.
Probably want both sides big:
fast F (e.g. Salsa20)
is happiest with big output.

How to build large-output H
using small-output h?
Define $H(x) = (h(x), 0, \ldots, 0)$.

How to build large-input F
using small-input f?
Define $F(x) = f(h(x))$.

Chaining? Almost free.

Several SipHash 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.

Focus: hash flooding

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

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

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

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

December 1999, Bernstein, `dnscache` software:

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

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

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

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

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

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

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

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

Defending against hash flooding

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

Defending against hash flooding

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

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

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

Store string s in list $\#i$
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.

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.

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!

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
$H(m_1, m_2, \ldots, m_{12}) =$
$m_1 k_1 + m_2 k_2 + \cdots + m_{12} k_{12}$.
$k_1, k_2, \ldots, k_{12}$: random, 20-bit.
This is “provably secure”!

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

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 $H(s)$.

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.

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.

The importance of overhead

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

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

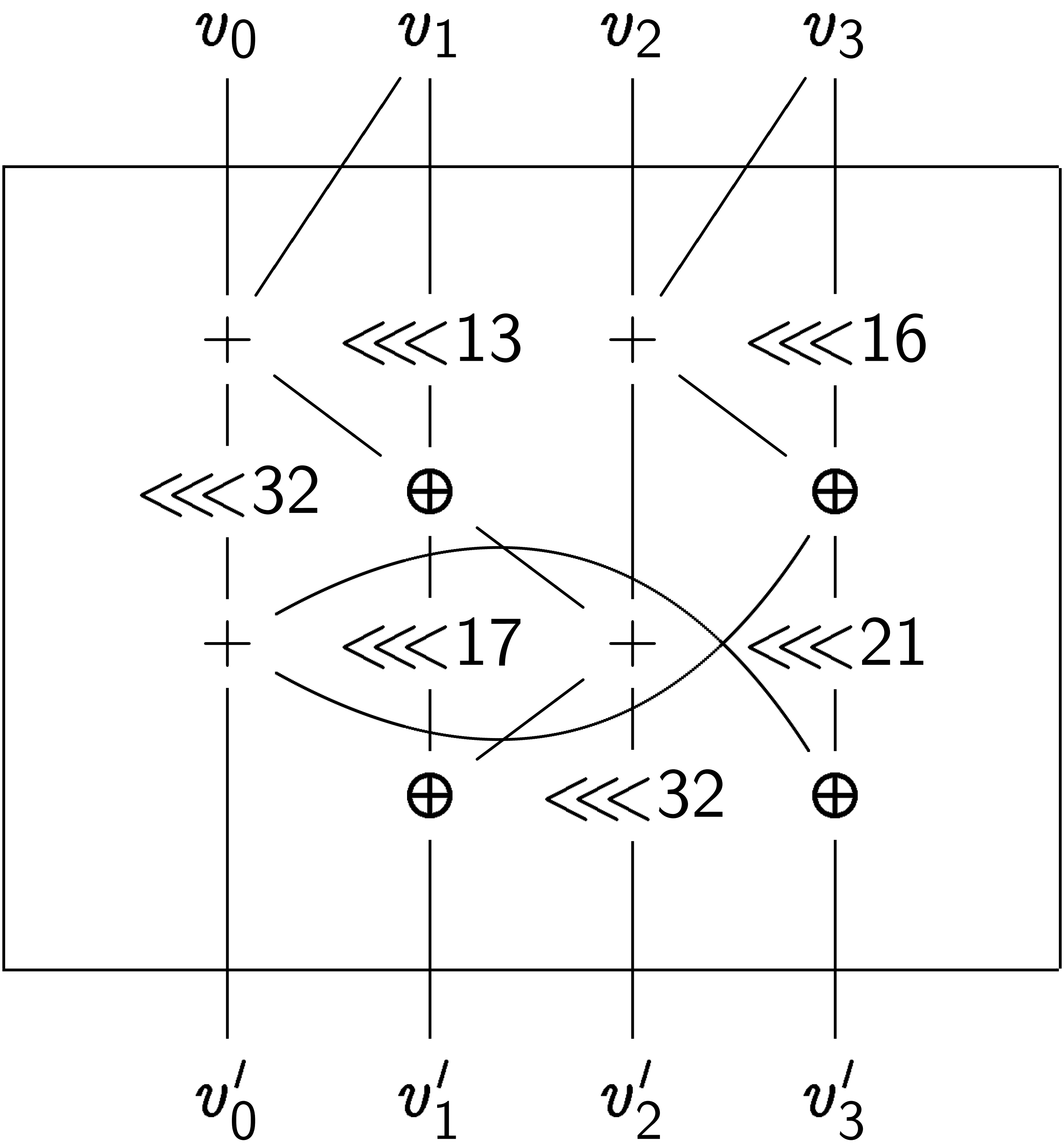
The importance of overhead

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

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

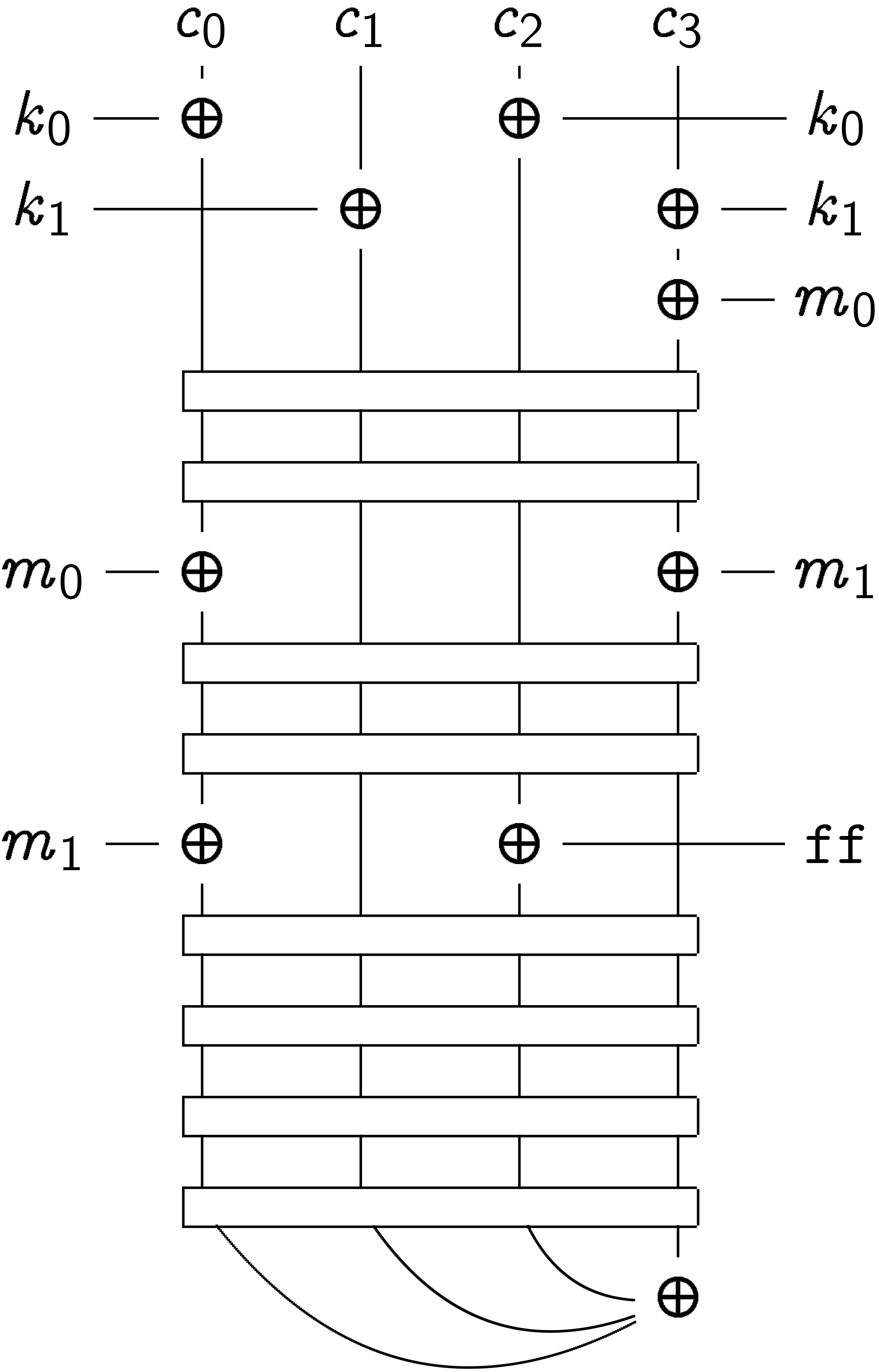
The hash-table perspective:
These speed advertisements
are only for long inputs,
ignoring huge overheads!

SipRound and SipHash



This is SipRound. Next page:

SipHash-2-4 applied to 16 bytes.


c_0
c_1
c_2
c_3
k_0
k_0
k_1
k_1
m_0
m_0
m_1
m_1
ff

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.