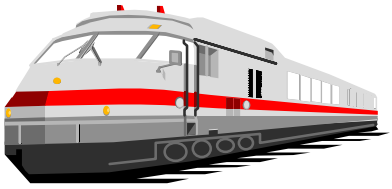
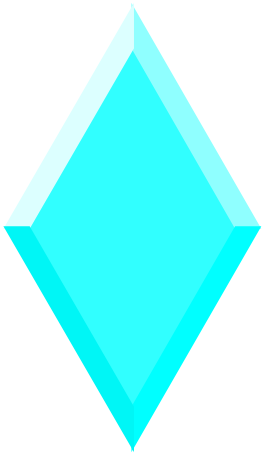
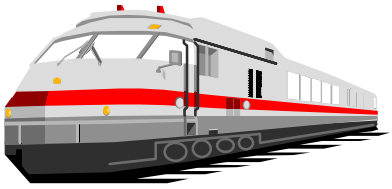


MMH (Multilinear Modular Hashing)
Software Universal-Hashing at the Gbit/Second Rates

Shai Halevi Hugo Krawczyk





Universal-Hashing [CW79]

"Small set" of functions $H = \{ h : \Sigma^B \rightarrow \Sigma^s \}$

- E.g., hashing 1 Gbit buffer to 64 bits

- H is Universal if for any $x \neq y \in \Sigma^B$

$$\Pr_{h \in H}[h(x) = h(y)] = 2^{-s}$$

ε-Almost-Universal (ε-AU) if

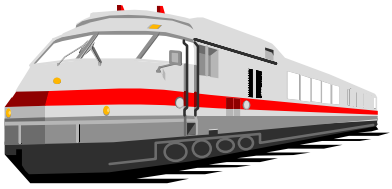
$$\Pr_{h \in H}[h(x) = h(y)] < \varepsilon$$

- H is XOR-Universal if for any $x \neq y \in \Sigma^B$, any $d \in \Sigma^s$

$$\Pr[h(x) \oplus h(y) = d] = 2^{-s}$$

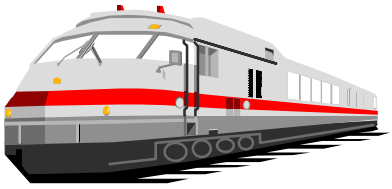
ε-Almost-XOR-Universal (ε-AXU) if

$$\Pr[h(x) \oplus h(y) = d] < \varepsilon$$



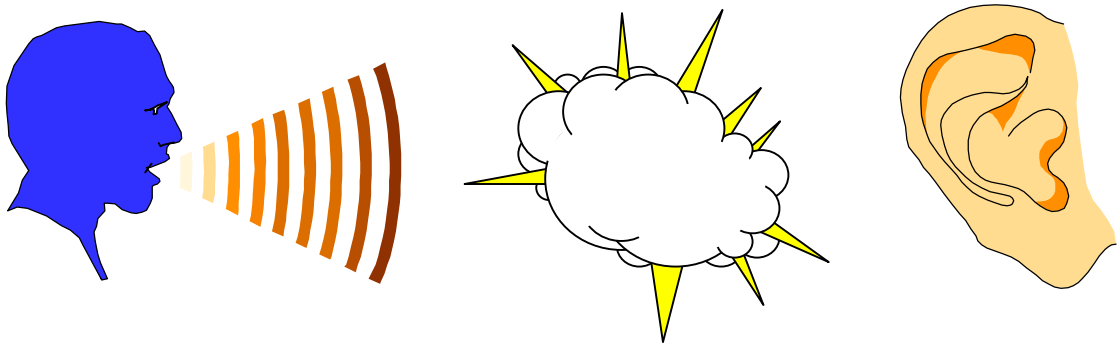
Applications

- Databases
- Search Engines
- Compilers
- Parallel Architectures
- Message authentication [WC81]
- Complexity theory
- etc...

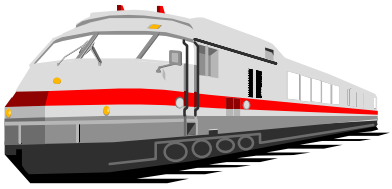


Message Authentication

- Two parties want to exchange messages over an unauthenticated channel.



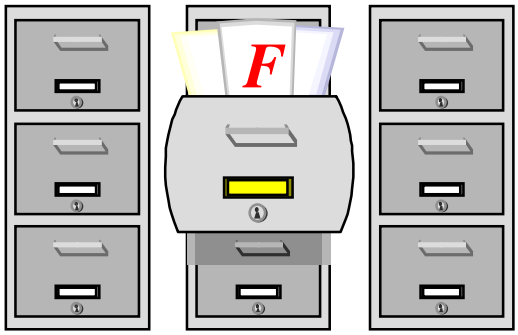
- The parties share a secret key.
- The sender attaches a *tag* to every message, which is computed from the message and the secret key.
- The receiver checks the *tag* before accepting the message as authentic



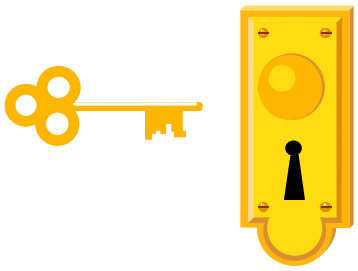
Using Universal Hashing

[WC81, St91, Kr94, Ro95 . . .]

Easy example: protecting a "public file"

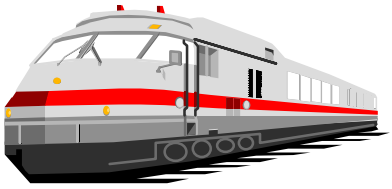


Store file *F* in public-domain



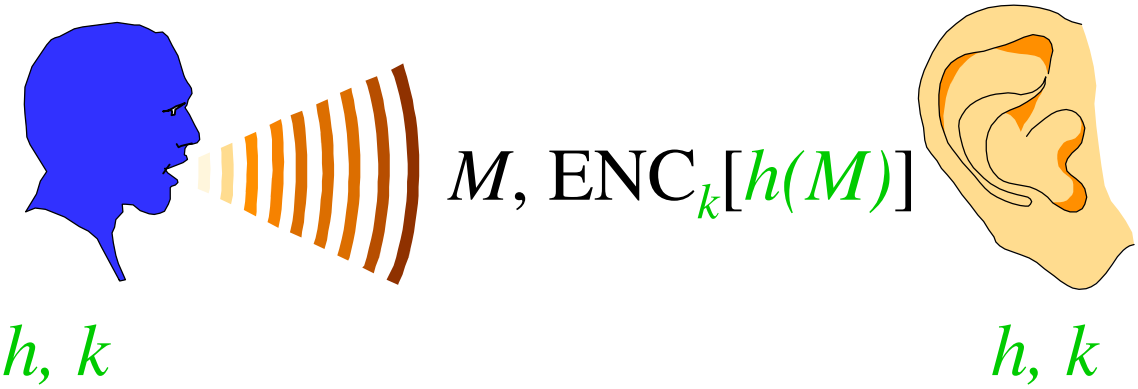
Keep function *h* and value *h(F)* in protected memory

- An attacker cannot see or modify *h*, *h(F)*
- Attackers bets bet - pick *F'* to maximize Pr[*h(F)* = *h(F')*]
- If *h* is ε-AU, the attacker succeeds with probability < ε



Using Universal Hashing (2)

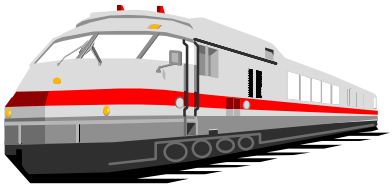
- Sending a message M over the network
  - Keep *h* protected as before
  - Send *h(F)* over the network -- encrypted



- *h(M)* is kept secret by the encryption, but we also worry about its integrity
  - The function *h* must have additional properties

Theorem: ε-AXU +"perfect encryption" (OTP) => Attacker succeeds with probability < ε

  - Provable security, no cryptographic assumptions
  - In practice, use block-cipher/pseudorandomness



Goal: FAST Hashing in Software

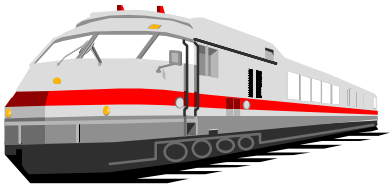
- Current implementations: ≈ 100 Mbit/sec on "yesterday's machines" (e.g. Pentium)

Getting **> 1 Gbit/sec** on today's machines

  - Needed even on 100 Mbit/sec links !

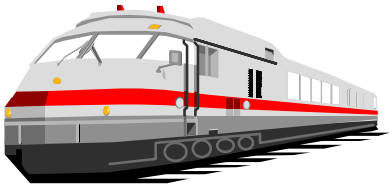
- Take advantage of modern CPU's
  - Parallelism
  - Support for DSP / Multi-media applications
    - In particular, support for inner-product. (e.g., SUN's VIS, Intel's MMX, ...)
  - 64-bit Architectures

- Low instruction-count
  - 1.5 instructions/byte on 32-bit machines
  - Less on 64-bit machines



MMH: Overview

- Start from a well-known construction
  - Multi-linear functions over a finite field
  - Standard techniques for reducing collision-probability and dealing with long messages
- Get a "reasonable" implementation
- Eliminate costly operations
  - Obtain a very fast "buggy implementation"
- Prove that the resultant construction is "almost as good" as the original



A Well-known Construction

Multi-linear functions over Z_p (p - prime)

$H = \{ g_{\boldsymbol{x}} : Z_p^{\,k} \rightarrow Z_p \}$

- $\boldsymbol{x} = \langle x_1 \ldots x_k \rangle$ and $\boldsymbol{m} = \langle m_1 \ldots m_k \rangle$

$$g_{\boldsymbol{x}}(\boldsymbol{m}) = \boldsymbol{x} \bullet \boldsymbol{m} \bmod p = \Sigma\, x_i m_i \bmod p$$

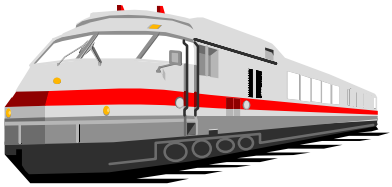
- $x_i, m_i \in \{0 \ldots p-1\}$

$\Rightarrow H$ is ε-AXU for ε=1/p

Our parameters

- $p = 2^{32}+15,\ k = 32,\ x_i, m_i \in \{0 \ldots 2^{32}-1\}$

$\Rightarrow H$ is ε-AXU for $\varepsilon=2^{-32}$



Modifications

Problem 1: Reduction mod p is expensive

- Carry inner-product over the integers. Reduce mod p at the end.

Problem 2: We get integers of > 64 bits during the inner-product operation

$$\sum \underbrace{x_i \, m_i}_{64}$$

$$\underbrace{\qquad\qquad}_{> 64}$$

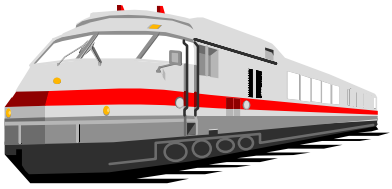
- Ignore carry-bits beyond the 64'th bit-location
- Same as reducing mod 2^{64}

Problem 3: Output is in $\{0 \ldots 2^{32}+14\}$, requires 33 bits

- Ignore most-significant-bit in output
- Same as reducing mod 2^{32}

$$h_{\boldsymbol{x}}(m) = [[[\boldsymbol{x} \bullet m] \bmod 2^{64}] \bmod p] \bmod 2^{32}$$

Theorem: ε-AXU with $\varepsilon \approx 2^{-30}$

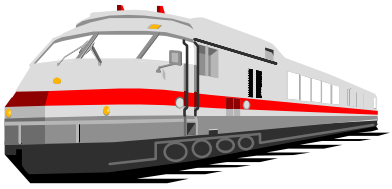


Inner-product (32-bit)

- 32x32 integer multiplication
- Addition with carry

```
sumHi:sumLow = 0
for i=1 to 32 do                          1  ins.
begin
  load m[i]                               1  ins.
  load x[i]                               1  ins.
  tmpHi:tmpLow = m[i]*x[i]                2  ins.
  sumLow += tmpLow                        1  ins.
  sumHi += tmpHi + carry                  1  ins.
end                                       -----
                                          7 ins.
```

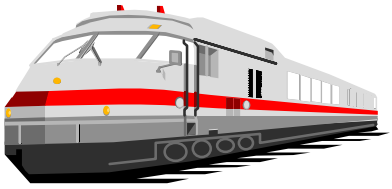
- With optimizations, can get below 6 instructions/word (1.5 instructions/byte)



Modular-reduction

- Computing $[sum \bmod (2^{32}+15)] \bmod 2^{32}$
- $2^{32}a+b \equiv 2^{32}a + b - a\,(2^{32}+15)$
  $= b - 15a \pmod{2^{32}+15}$
- $sum = 2^{32}a+b$ $\quad a, b \in \{0 \ldots 2^{32}-1\}$

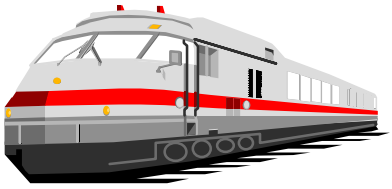
  1. c:d = b–15a $\quad c \in \{-15 \ldots 0\}$
  2. e:f = d–15c $\quad e \in \{0, 1\}$
  3. if e = 1 and f > 14
     output f – 15
     else
     output f
- About 6–10 instructions



Other points

- Code Optimizations
  - Loop unrolling
  - Processing 2–4 message blocks at a time (less loads)
  - Re-ordering instructions
  - Pre-fetching message words to cache

- Problems
  - Collision probability too high
  - Can only handle 32-word messages



Reducing Collision-probability

Naive solution: Use 2 independent keys

$\boldsymbol{x} = \langle x_1 \ldots x_k \rangle,\ \boldsymbol{x}' = \langle x_1' \ldots x_k' \rangle$

$g_{\boldsymbol{x}\,\boldsymbol{x}'}(m) = g_{\boldsymbol{x}}(m) \,\|\, g_{\boldsymbol{x}'}(m)$

- ε-AXU with $\varepsilon = 2^{-64}$
- keys are twice as long

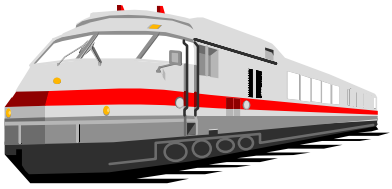
Better solution: Use "Toeplitz matrices"

$\boldsymbol{x} = \langle x_1 \ldots x_k \rangle,\ \boldsymbol{x}' = \langle x_2 \ldots x_{k+1} \rangle$

$g_{\boldsymbol{x}\,\boldsymbol{x}'}(m) = g_{\boldsymbol{x}}(m) \,\|\, g_{\boldsymbol{x}'}(m)$

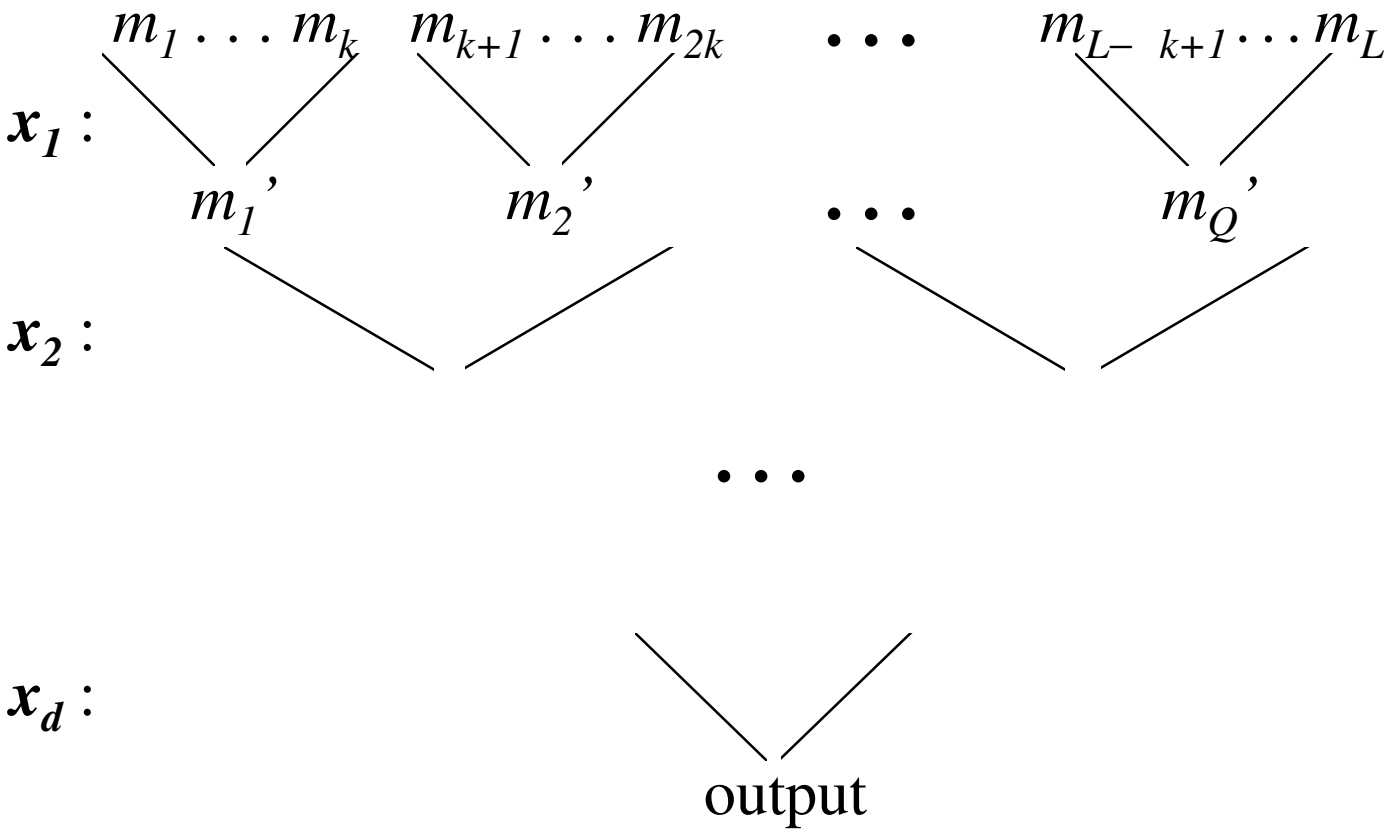
- Also ε-AXU with $\varepsilon = 2^{-64}$
- keys are only slightly longer

- It is possible to get the collision-probability as low as needed using the same idea.



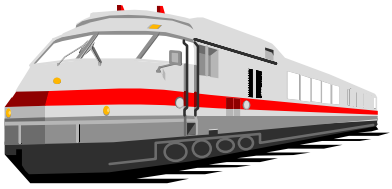
Hashing Long Messages

Tree-hashing



- Whole thing is $d\varepsilon$-AXU
- keys are d times larger
- Lower keys are almost never used

"Hybrid solution": Use only top 2 levels, hash the rest using a different hashing scheme.

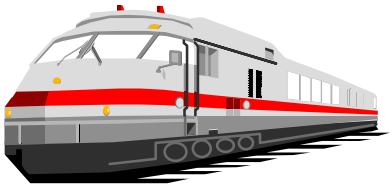


Performance

- Implementations for
  - 150 MHz PowerPC 604, and
  - 200 MHz Pentium-Pro

Mbit/sec	2^{-60}-AXU	2^{-30}-AXU
message in memory	390 / 380	600 / 645
message in cache	420 / 500	820 /1080

These numbers are conservative



Comparison with other methods

CBC-MAC - as fast/slow as block-cipher

- DES: 5-15 Mbps (Pentium 90)
- RC5 (12 rounds): 26 Mbps (Pentium 90)

HMAC - as fast/slow as message-digest

- MD5: 114 Mbps (Pentium 90) [BGV96]
- SHA: 60 Mbps (Pentium 90)
- RIPEMD 160: 40 Mbps (Pentium 90)

Universal Hashing

- CRC (64-bit): 85-100 Mbps (Pentium 90) [Sh96]
- Bucket Hashing ("32 bit"): 260-330 Mbps (MIPS 4400) [Ro95]
- MMH, 64 bit output: 500 Mbps (P6 200)
- MMH, 32 bit output: 1080 Mbps (P6 200)