PRODUCT SECURITY INITIATIVE

QUALCOMM

Boole: a hash/MAC/stream cipher primitive

Greg Rose

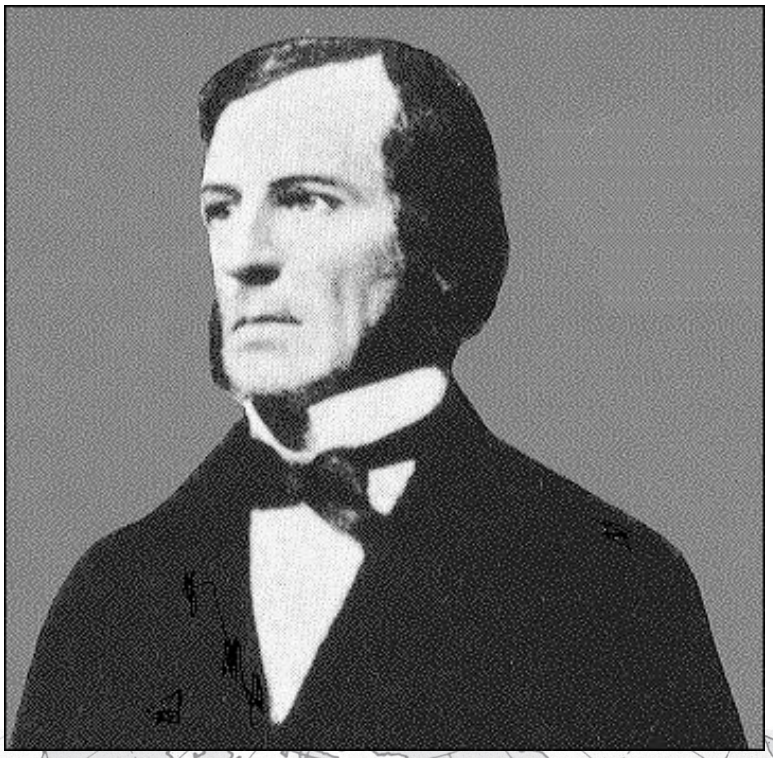
ggr@qualcomm.com

Outline

- Introducing Boole
- Design influences
- Structure
- Diagram
- Golomb Rulers
- The nonlinear functions
- MAC functionality
- Performance

Introducing Boole

- Named in memory of George Boole (1815-1864, founder of mathematical logic and Boolean algebra)



Introducing Boole

- Fast (and hopefully secure) family of primitives for hash, MAC and stream cipher
- Suitable for hardware and software
- Uses almost entirely Boolean operations
- Nonlinear Shift Register

- Open Source implementation
- Submission for Advanced Hash Standard

Design influences

- Member of the SOBER family
  - word-oriented shift register design
- Input and output modes, like PANAMA (Sponge)
  - Not both at once
  - Implementation does both in two separate states
- Keep the nonlinearity in the state, like SCREAM
- Piling up small non-linearity like Trivium
- Hash/MAC -- Reintroduce differences later, like SHA-256

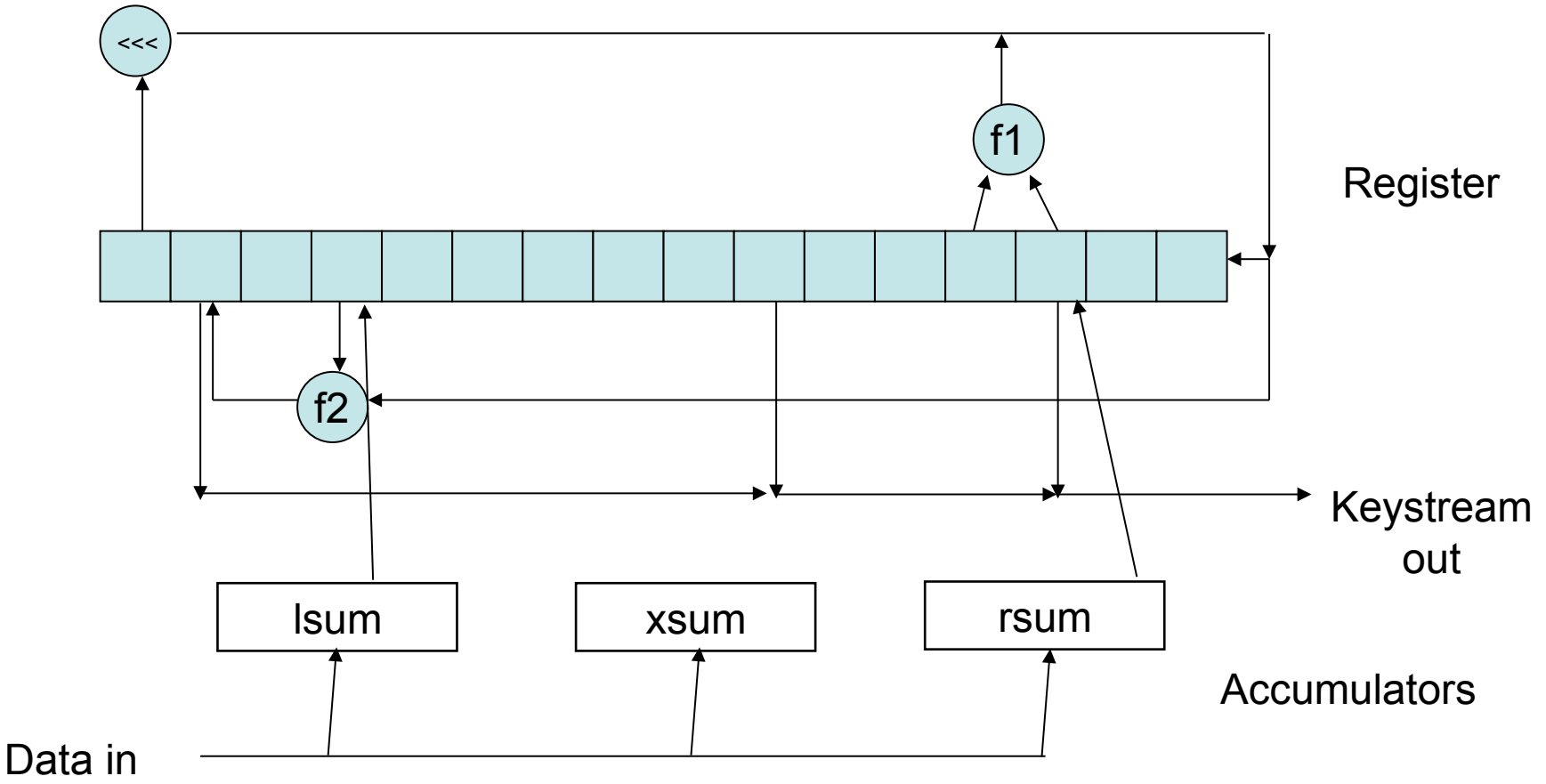
Structure

- 16-word shift register
  - both nonlinear feedback and “feed forward”
  - linear output filter function
  - Accepts/delivers one word per register cycle
- Three additional words used during input phase for keying, hashing and MACing
- Two related but different nonlinear functions
  - entirely computed (no tables)
  - use only rotation, NOT, XOR, OR

Family of functions

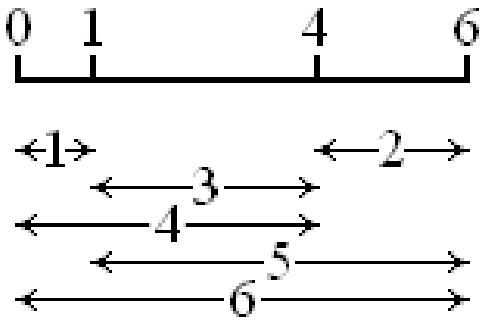
- (Almost) identical source code for word sizes
  - 16/32/64 bit words
  - Different nonlinear functions for word sizes
  - Word-to-byte glue
- Hash/MAC outputs up to 128/256/512 bits
- Symmetric key security goals of 64/128/256 bits
- Boole64 meets all submission criteria
- Boole32 can be considered an alternate, lower resource version for 224- and 256-bit outputs
- Boole16 useful as an MD5 replacement or MAC for constrained environments, also easier to study and attack

Diagram


<<<
f1
Register
f2
Keystream
out
lsum
xsum
rsum
Accumulators
Data in

Golomb Rulers

- AKA Full Positive Difference Set



- Need 3:
  - Feedback (0, 12, 13, 16)
  - "feed forward" (0, 2, 8, 12, 15)
  - nonlinear functions (0, 1, 5, 7, 19, 22)
    - Special "modular" FPDS
    - (1, 9, 10, 13, 15) and (1, 3, 9, 10, 14) (16-bit)
    - (0, 1, 5, 7, 19, 22) (32 bit, two arrangements, from Shannon)
    - (0, 1, 3, 20, 34, 42, 55, 60) and (0, 1, 5, 27, 35, 46, 52, 55) (64 bit)

Nonlinear functions – 16-bit

```
w ^= ROTL(w, A) | ROTL(w, B);
w ^= ~ROTL(w, C) | ROTL(w, D);
return w;
```

- f1: (A, B, C, D) = (9, 13, 10, 15)
- f2: (A, B, C, D) = (3, 14, 9, 10)

Nonlinear functions – 32-bit

- Same as Shannon stream cipher

```
w ^= ROTL(w, A) | ROTL(w, B);
w ^= ROTL(w, C) | ROTL(w, D);
return w;
```

- f1: (A, B, C, D) = (5, 7, 19, 22)
- f2: (A, B, C, D) = (7, 22, 5, 19)

Nonlinear functions –64-bit

```
w ^= 0x6996c53a;
w ^= ROTL(w, A) | ROTL(w, B);
w ^= ROTL(w, C) | ROTL(w, D);
w ^= (w << E) | ROTL(w, F);
return w;
```

- f1: (A, B, C, D, E, F) = (34, 42, 20, 55, 3, 60)
- f2: (A, B, C, D, E, F) = (35, 46, 27, 52, 5, 55)
- f1 rotates left (shown), f2 rotates and shifts *right*
- Used cycle-finding to test for "random function like" behavior of hundreds of variants

Phases of operation – input phase

- Input phase (key, nonce, hash/MAC data)
  - *lsum* = *f*1(*lsum*)
  - Input XORed to *lsum/xsum*
  - XOR *lsum* into *rsum*
  - Rotate *lsum* left and *rsum* right one bit
  - XOR *lsum* to R[13], *rsum* to R[3]
  - Cycle register

Phases of operation – mixing and output

- "*mixing*" phase (between input and output)
  - Length in bits of input XORed into R[0]..R[3]
  - Output bit length XORed into R[4]
  - XOR *lsum* into R[4, 7, 10, 13], XOR *xsum* into R[5, 8, 11, 14], XOR *rsum* into R[6, 9, 12, 15]
  - Then *diffuse* (cycle 16 times)

- Output phase (stream, hash/MAC)
  - Cycle, then XOR R[0, 8, 12] to output buffer (cleared in advance for hash/MAC)

Kinds of operation

- Hash function
  - Input data
  - Finalize (mixing, then mixing again!)
  - Output hash
- MAC/Stream
  - Our implementation ties these together
    - Two separate sets of state variables for hash/MAC and stream
    - Requires use of nonce
  - Key: Input key, *finish*, save register
  - Nonce: restore register, Input nonce (twice), *finish*
  - Macdata: Input plaintext only, or Stream: XOR stream
  - Encrypt: (stream then MAC) or Decrypt: (MAC then stream)
  - Mac: *mixing*, then output MAC

Performance (64-bit fast version)

(2.4GHz Macbook Pro, Intel Core 2 Duo)
1600-byte blocks, 160-bit MACs, 512-bit hashes
Cycles per ... based on 2.4 GHz assumption.
646.058116 Mbyte per second stream encryption, 3.71 cycles per byte
500.000000 Mbyte per second encrypt blocks, 4.80 cycles per byte
425.531915 Mbyte per second MAC blocks, 5.64 cycles per byte
253.164557 Mbyte per second encrypt/MAC blocks, 9.48 cycles per byte
253.164557 Mbyte per second MAC/decrypt blocks, 9.48 cycles per byte
2.272727 million 128-bit keys per second, 1056.00 cycles per key
1.560062 million 128-bit nonces per second, 1538.40 cycles per nonce
606.903079 Mbyte per second large buffer hash, 3.95 cycles per byte
444.444444 Mbyte per second block hash, 5.40 cycles per byte

Hardware implementation

- 19 words worth of flipflops
- Nonlinear function requires only OR gates
- Rotations are essentially free
- 64-bit adder needed for accumulating length, but can be slow and simple.

Reference code interface

```
HashReturn ble_key(ble_ctx *c, const UCHAR key[], int keylen, int maclen);
HashReturn ble_nonce(ble_ctx *c, const UCHAR nonce[], int nlen)
HashReturn ble_stream(ble_ctx *c, UCHAR *buf, int nbits);
HashReturn ble_macdata(ble_ctx *c, UCHAR *buf, int nbits);
HashReturn ble_encrypt(ble_ctx *c, UCHAR *buf, int nbits);
HashReturn ble_decrypt(ble_ctx *c, UCHAR *buf, int nbits);
HashReturn ble_mac(ble_ctx *c, BitSequence *hashval);

HashReturn Init(hashState *state, int hashbitlen);
HashReturn Update(hashState *state, const BitSequence *data,
    DataLength databitlen);
HashReturn Final(hashState *state, BitSequence *hashval);
HashReturn Hash(int hashbitlen, const BitSequence *data,
    DataLength databitlen, BitSequence *hashval);
```