CRC magic tricks

dram

2021-12-29

About me

- `@dramforever` on most random platforms
  - GitHub, Twitter, ...
- `https://dram.page`
- Call me ‘dram’

Background

Galois field $\mathrm{GF}(2)$

- Finite field
- Elements: 0, 1
- Addition is XOR
- Multiplication is AND

Notable properties:

- $2 = 0$
- $a + b = a - b$

Polynomials in GF(2)

$$p(x) = \sum_{n=0\ldots d} a_n x^n$$

- $a_n \in \mathrm{GF}(2)$ are the *coefficients*
- $\deg(p(x))$: Power of highest power term with non-zero coefficient
- x is just a symbol
  - Polynomials are not functions

Polynomial addition

- Addition: $(x^3 + x) + (x + 1)$

$$\begin{array}{cccccc} & x^3 & + & x & & \\ & & + & x & + & 1 \\ \hline = & x^3 & & & + & 1 \end{array}$$

Polynomial multiplication

- Multiplication: $(x+1)(x^3+x+1)$

$$
\begin{array}{ccccccccc}
 & x^4 & + & x^3 & & & & & \\
 & & & & & x^2 & + & x & & \\
 & & & & & & & x & + & 1 \\
\hline
= & x^4 & + & x^3 & + & x^2 & & & + & 1
\end{array}
$$

Polynomial Euclidean division

- Given $a(x)$ and $b(x)$, there is unique $(q(x), r(x))$ such that $\deg(r(x)) < \deg(b(x))$ and:

$$a(x) = b(x) \cdot q(x) + r(x)$$

Some shorthands:

- Quotient: $q(x) = a(x) \operatorname{div} b(x)$
  - (Note: Gopal et al. (2009) writes this as $\lfloor a(x)/b(x) \rfloor$)
- Remainder: $r(x) = a(x) \bmod b(x)$

Polynomial GCD

- $b(x)$ *divides* $a(x)$ iff $a(x) \bmod b(x) = 0$
- $\gcd(a(x), b(x))$ is the unique largest-degree polynomial $g(x)$ that divides both $a(x)$ and $b(x)$

Euclidean algorithm works for polynomial GCD

```
def poly_gcd(a, b):
    while b != 0:
        a, b = b, poly_mod(a, b)
    return a
```

Cyclic redundancy check

Cyclic redundancy check

- An family error *detecting* codes
- Based on polynomials in $\mathrm{GF}(2)$
- Not cryptographically secure *at all*
- Commonly called CRC-N for a CRC with an N-bit check sequence
- No single standard, parameters vary greatly
  - (For a catalogue of various CRCs see Cook (2021a))

CRC implemented as LFSRs

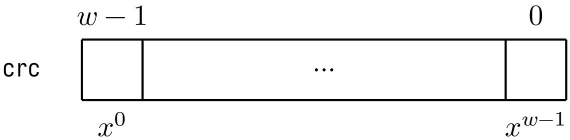
- Input message as bit-stream
  - For each byte, put LSB first[1]
  - `"\xf0\x30"` $\rightarrow$ `0000 1111 0000 1100`

```
def crc(message):
    crc = INIT

    for b in message:
        crc ^= b
        if crc & 1: crc = (crc >> 1) ^ TAP
        else:       crc = crc >> 1

    return crc ^ FINAL
```

[1] Some implementations use other bit orders.

Bit-streams and polynomials

- Least significant bit = First transmitted bit = Highest power term
- Parameters `INIT`, `TAP`, `FINAL` are *bit-streams*
- The register `crc` is a *bit-stream*



Back to the CRC code

- Suppose we're working with an N-bit CRC

```
def crc(message):
    crc = INIT

    for b in message:
        # crc <- crc + b x^(N - 1)
        crc ^= b

        # if crc has x^(N - 1) term
        if crc & 1:
            # crc <- crc x + x^N + TAP
            # (Right shift discards x^(N - 1) term instead of turning it into x^N)
            crc = (crc >> 1) ^ TAP
        else:
            # crc <- crc x
            crc = crc >> 1

    return crc ^ FINAL
```

CRC computation, but polynomials

The main loop:

- For each bit b in message:
  - $\texttt{crc} \leftarrow \texttt{crc} + bx^{N-1}$
  - If the x^{N-1} coefficient of $\texttt{crc}$ is 1, then $\texttt{crc} \leftarrow \texttt{crc} \cdot x + x^N + \texttt{tap}$
  - Else: $\texttt{crc} \leftarrow \texttt{crc} \cdot x$

Simplified:

- For each bit b in message:
  - $\texttt{crc} \leftarrow \texttt{crc} \cdot x + bx^N$
  - If the x^N coefficient of $\texttt{crc}$ is 1,
  - Then $\texttt{crc} \leftarrow \texttt{crc} + x^N + \texttt{tap}$

Why is the x^N here

- Example: `crc = 0x3`, LSB of `crc` is 1
- `crc` $= x^{N-2} + x^{N-1}$

After shifting

- `crc >> 3 = 0x1`
- `crc` $\cdot x = x^{N-1} + x^N$
- `crc` $\cdot x + x^N = x^{N-1}$

What does the algorithm do?

Without TAP:

- $\texttt{crc}^* \leftarrow \texttt{init}$
- For each bit b in message:
  - $\texttt{crc}^* \leftarrow \texttt{crc}^* \cdot x + bx^N$
  - ~~If the x^N coefficient of $\texttt{crc}^*$ is 1,~~
  - ~~Then $\texttt{crc}^* \leftarrow \texttt{crc}^* + x^N + \texttt{tap}$~~
- Return $\texttt{crc}^* + \texttt{final}$

Let m be the message, with length in bits L, then the result is:

$$\texttt{crc}^* = mx^N + \texttt{init} \cdot x^L + \texttt{final}$$

What does the algorithm do?

- Claim: $\texttt{crc} \equiv \texttt{crc}' \pmod{(\texttt{tap} + x^N)}$
- $\deg(\texttt{crc}) < N$

Therefore:

$$\begin{aligned}\texttt{crc} &= \texttt{crc}' \bmod (\texttt{tap} + x^N) \\ &= (mx^N + \texttt{init} \cdot x^L + \texttt{final}) \bmod (\texttt{tap} + x^N)\end{aligned}$$

Shorter symbols

From now on, we'll use these symbols consistently:

- L (length) is the message length in bits
- m (message) is the message bit string as a polynomial
- N is the length of the CRC
- r (remainder) is `crc`
- F is `final`
- I is `init`
- P (polynomial) is `tap` $+ x^N$

$$r = (mx^N + Ix^L + F) \bmod P$$

Efficient CRC with CLMUL

Carryless multiplication

- ‘Carryless multiplication’ operation
  - Intel: `pclmulqdq`
  - ARM: `pmull`
  - RISC-V (`Zbc`): `clmul{,h,r}`
- Much faster than software loop
- Also much faster than div/mod
- Directly corresponds to register-sized $\mathrm{GF}(2)$ polynomial multiplication

Efficient CRC by folding

- Method described in Intel whitepaper (Gopal et al., 2009)
  - *Fast CRC Computation for Generic Polynomials Using PCLMULQDQ Instruction*
- Keep intermediate result a of $2N$ bits, $a \equiv mx^N + Ix^L \pmod{P}$
- Read message in N-bit chunks, updating a if needed
- Calculate $(a \bmod P) + F$ for final result
- Using precomputed constants, avoids (dynamic) polynomial div/mod entirely.

Folding step

- Read message in N-bit chunks m_k (so $\deg(m_k) < N$), new a should be:

$$\begin{aligned} a^* &\equiv (mx^N) \cdot x^N + m_k x^N + I x^{L+N} \pmod{P} \\ &\equiv (a + m_k) x^N \pmod{P} \end{aligned}$$

- At each iteration we need to ensure $\deg(a^*) < 2N$
- Split a into 'high N terms' and 'low N terms', $a = a_H x^N + a_L$, $\deg(a_L) < N$
- $a^* = (a_L + m_k)x^N + a_H(x^{2N} \bmod P)$
- ($x^{2N} \bmod P$ can be precomputed)

Barret reduction

- We need to find $a \bmod P$
- Suppose $a = Pq + r$, where $\deg(q), \deg(r) < N$
- Let $\mu = x^{2N} \operatorname{div} P$, then $\deg(\mu) = N$
- Property: $\deg(x^{2N} + \mu P) < N$

Barret reduction:

- Let $t = (a \operatorname{div} x^N) \cdot \mu$, then

$$\begin{aligned} t &= (a \operatorname{div} x^N) \cdot \mu \\ &= (\mu P q \operatorname{div} x^N) + (\mu r \operatorname{div} x^N) \\ &= (((x^{2N} + o(x^N)) \cdot q) \operatorname{div} x^N) + o(x^N) \\ &= x^N \cdot q + o(x^N) \end{aligned}$$

- Therefore $q = t \operatorname{div} x^N$

Barret reduction

- $r = a + Pq$
- $r + F$ is our final CRC
- μ is precomputed
- (Note: $u \operatorname{div} x^N$ is not really a division, just takes 'higher half')

Slightly simplifying μ

- (Used by Wolf (2019), also in historic RISC-V Bitmanip spec (Wolf, 2021))
- μ is degree N, which does not fit in N bits
- Use $\mu \operatorname{div} x$ instead of μ. Let c_0 be the constant term of μ
- $x^{2N} + x(\mu \operatorname{div} x)P = o(N) + c_0 P = o(N+1)$
- $x^{2N-1} + (\mu \operatorname{div} x)P = o(N)$

$$\begin{aligned} t_1 &= (a \operatorname{div} x^N) \cdot (\mu \operatorname{div} x) \\ &= (((x^{2N-1} + o(x^N)) \cdot q) \operatorname{div} x^N) + o(x^{N-1}) \\ &= x^{N-1} q + o(x^{N-1}) \end{aligned}$$

- Still works: $q = t_1 \operatorname{div} x^{N-1}$
- (Noted by Kutenin (2021) that some implementations of the same CRC-32 differ in the constant term of μ)

Bit reversed CLMUL

- Store N-bit result in N-bit GPR
- Store $2N$-bit intermediate in two N-bit GPRs
- Polynomial in register is bit reversed
  - LSB = Highest power term

How do we calculate bit-reversed CLMUL?

- Two N-bit inputs, $(2N-1)$-bit result
- CLMUL is symmetric:

$$\mathrm{rev}_{2N-1}(\mathrm{clmul}(a, b)) = \mathrm{clmul}(\mathrm{rev}_N(a), \mathrm{rev}_N(b))$$

CLMUL is symmetric

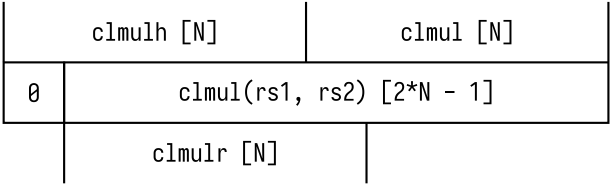
- Highest power term?

$$
\begin{array}{cccccccccc}
 & x^4 & + & x^3 & & & & & & \\
 & & & & & x^2 & + & x^1 & & \\
 & & & & & & & x^1 & + & x^0 \\
\hline
= & x^4 & + & x^3 & + & x^2 & & & + & x^0
\end{array}
$$

- Lowest power term?

$$
\begin{array}{cccccccccc}
 & x^0 & + & x^1 & & & & & & \\
 & & & & & x^2 & + & x^3 & & \\
 & & & & & & & x^3 & + & x^4 \\
\hline
= & x^0 & + & x^1 & + & x^2 & & & + & x^4
\end{array}
$$

CLMUL instructions



- In bit reversed representation:
  - $a \cdot b$, low half: `clmul rd, rs1, rs2`, high half: `clmul ; slli 1`
  - $(a \cdot b) \text{ div } x^N$ is `clmul ; slli 1`
  - $(a \cdot b) \text{ div } x^{N-1}$ is `clmul`

Further speedups

- `pclmulqdq` can help handle 128-bit chunks
- Handling multiple (e.g. 4) chunks in parallel
  - Modern processors have many CLMUL units, to keep up with AES for GCM

All these are described in the Intel whitepaper (Gopal et al., 2009).

Table-based CRC

- Without CLMUL, make a table of $T(k) = (k \cdot x^N) \bmod P$ for all $\deg(k) < C$
- Chunk size e.g. $C = 8$, each octet as a chunk
- Let $r = r_H x^{N-T} + r_L$, $\deg(r_L) < N - T$

$$\begin{aligned} r^* &= ((r_H + m_k) \cdot x^N) \bmod P + r_L \\ &= T(r_H + m_k) + r_L \end{aligned}$$

- C is usually very small, as table requires $N \cdot 2^C$ bits

Summary of computing CRC

- LFSR
- Bit-reversed representation
- Speeding up CRC with CLMUL
  - Folding
  - Barret reduction
- Using `RVZbc`

Preimage attacks

‘Security’ of CRC

- *CRC has no cryptographic security at all*
- CRC only intends to protect against inadvertent changes, especially those occuring during transmission and storage

Linear/affine properties of CRC

- For two messages with identical length:

$$
\begin{aligned}
r_1 &= (m_1 x^N + I x^L + F) \bmod P \\
r_2 &= (m_2 x^N + I x^L + F) \bmod P
\end{aligned}
$$

- Adding the two gives:

$$r_1 + r_2 = (m_1 + m_2) x^N \bmod P$$

- CRCs are *affine*

Combining messages with XOR

- If $m_1 \oplus m_2 = m_3 \oplus m_4$
- Then $\mathrm{crc}(m_1) \oplus \mathrm{crc}(m_2) = \mathrm{crc}(m_3) \oplus \mathrm{crc}(m_4)$
- In particular, let z be the all-zeros messages with same length as m_k
- Then $\mathrm{crc}(m_1 \oplus m_2) = \mathrm{crc}(m_1) \oplus \mathrm{crc}(m_2) \oplus \mathrm{crc}(z)$

Finding constrained preimages

- Example: Find integer i such that its ASCII representation hashes to (known) r:

```
crc32(str(i)) = r
```

- Application: (Does anyone know what this is?)

```
<d p="507.77900,1,25,16777215,1640704211,0,4e291766,59565938560754176,10">HDMI orz</d>
```

- Find uid (up to around 10^9) such that:

```
crc(str(uid)) = 0x4e291766
```

Finding constrained preimages

- Simple case, fixed length 9
- Meet-in-the-middle (Dot means `\0`):

```
    crc("123456789")
=   crc("12345....")
  ^ crc(".....6789")
  ^ crc(".........")
```

- Generate all 10^5 possible ‘high parts’, all 10^4 possible ‘low parts’
- Hash table of ‘high parts’ `high_table[0x770a59bd] = "12345...."`
- Array of ‘low parts’ `low_table[i] = (".....6789", 0x5af77435)`
- `crc(".........") = 0xe60914ae`

Finding constrained preimages


```
from zlib import crc32

low_table = [
    (i, crc32(b'\x00' * 5 + str(i).encode()))
    for i in range(10**4)
]

high_table = {
    crc32(str(i).encode() + b'\x00' * 4) : i
    for i in range(10**5)
}

def find_num(target):
    crc_z = crc32(b'\x00' * 9)
    for low_num, low_hash in low_table:
        expect = low_hash ^ target ^ crc_z
        if expect in high_table:
            high_num = high_table[expect]
            return high_num * (10**4) + low_num
```

Preimage exercise

- (Python: `zlib.crc32`)
- Find integer i such that `zlib.crc32(str(i)) == 0x4e291766`
- `0 < i < 10**9`

Bit-flipping messages

- Several ‘free positions’ in a message where we can change the bits to anything
- Pick values for these bits such that the message has desired CRC

Use cases:

- Modifying a file so it hashes to interesting values
- Tamper with file without CRC changing

Bit-flipping messages

- For each 'free position'
- Flipping bit $k \leftrightarrow$ CRC gets bitwise-xor by a_k
- We have N positions we can flip a bit
- Let $d_k = 1$ if flip position k, 0 if no flip

$$\texttt{orig} \oplus \texttt{target} = \bigoplus_{k=0\ldots N-1} d_k \cdot a_k$$

- Find linear combination of a_k that gives $\texttt{orig} \oplus \texttt{target}$
- Solvable with Gaussian elimination

Bit-flipping messages

- Notable features
  - If all we need is where to flip bits, the original message is *not required*
- In general, for CRC-N, N free positions are needed
- We're going to use this $\mathrm{GF}(2)$ polynomial linear equation solver later

Summary of preimage attacks

- Constrained charset: Meet-in-the-middle
- Bit flips: Gaussian elimination

Reverse engineering

CRC without unknown parameters

- Worse case: we only know it's a CRC with width N
- We have a few known (m_k, r_k) pairs
- Rocksoft Model Algorithm parameters (Williams, 1993)
  - N is `width`
  - P is `poly`
  - I is `init`
  - F is `xorout`
  - `refin` and `refout`
    - Bit order of input/output
    - Both assumed to be `true` here
    - Otherwise, only takes 4 tries to test all combinations
  - `name` and `check` are irrelevant
- Used in the CRC catalogue (Cook, 2021a)

How do we even know it's a CRC?

- Use the affine property
- If $m_1 \oplus m_2 = m_3 \oplus m_4$
- Then $\mathrm{crc}(m_1) \oplus \mathrm{crc}(m_2) = \mathrm{crc}(m_3) \oplus \mathrm{crc}(m_4)$

  Check for all pairs you can find
- Consecutive numbers or related strings often have linear relationships

  Can also help uncover incorrect (m, r) pairs

Messages with CRC appended

- Most often, the CRC comes right after the message
- In terms of polynomials:
  - Message: m
  - Append N zeros: mx^N
  - Set to CRC value: $mx^N + r$

Known message and CRC

$$r \equiv mx^N + Ix^L + F \pmod{P}$$

- Known m and r:
- Moving knowns to one side:

$$mx^N + r \equiv Ix^L + F \pmod{P}$$

Comparing equal length messages

- If we have two known $m_k x^N + r_k$ with m_1 and m_2 having equal length L

$$m_1 x^N + r_1 \equiv I x^L + F \pmod{P}$$
$$m_2 x^N + r_2 \equiv I x^L + F \pmod{P}$$

- Adding the two gives:

$$(m_1 x^N + r_1) + (m_2 x^N + r_2) \equiv 0 \pmod{P}$$

- Shorthand: $v_k = m_k x^N + r_k$

‘Difference messages’

- If $L_j = L_k$ then P divides $v_j + v_k$
- Finding all linear independent pairs of (j, k):
  - Sort all messages by length
  - Find all adjacent equal-length pairs

  We have several $v_j + v_k$ that are ‘multiples’ of P
- Next task: Find degree N polynomial P
- ... such that P divides all $v_j + v_k$ where $L_j = L_k$

CRC RevEng

- CRC RevEng (Cook, 2021b) can find parameters for CRCs
- Algorithm for finding P: Brute force
  - Search through all polynomials with degree N, and with constant-term
  - Check if divides all differences
- Optimization:
  - If a certain $\deg(v_j + v_k) < 2N$, search for the smaller factor of $v_j + v_k$ instead

An alternative...?

- P divides all $v_j + v_k$ where $L_j = L_k$
- $\Leftrightarrow$ P divides the GCD of all such $v_j + v_k$

Taking a few GCDs may quickly isolate P:

- The GCD turns out to have degree N, then it is P
- The GCD has a degree slightly larger than N, use 'find smaller factor' method
- The GCD still has high degree... Try with more samples

(Even if we don't know N, we can guess that it's the degree of the GCD.)

An example

- Real data captured from a bus on an adjustable desk:

```
m1, r1 = AAFF 0040 2EEC
m2, r2 = AAFF 0060 2964
m3, r3 = AAFF 0050 2B08
```

$$(m_1 x^{16} + r_1) + (m_2 x^{16} + r_2) = x^{18} + x^{15} + x^{14} + x^{13} + x^4 + x^0$$
$$(m_1 x^{16} + r_1) + (m_3 x^{16} + r_3) = x^{19} + x^{15} + x^{13} + x^5 + x^2 + x^1 + x^0$$

- Calculating the GCD gives:

$$\gcd(\dots, \dots) = x^{16} + x^{14} + x^{13} + x^2 + x^0$$

- Conclusion: Probably CRC-16 with this P

Comparing differing-length messages

- If we have two known $m_k x^N + r_k$ with m_1 and m_2 having different lengths L_1 and L_2 respectively

$$m_1 x^N + r_1 \equiv I x^{L_1} + F \pmod{P}$$
$$m_2 x^N + r_2 \equiv I x^{L_2} + F \pmod{P}$$

- Adding the two gives:

$$(m_1 x^N + r_1) + (m_2 x^N + r_2) \equiv I(x^{L_1} + x^{L_2}) \pmod{P}$$

Solving for initial value

$$I(x^{L_1} + x^{L_2}) \bmod P = ((m_1 x^N + r_1) + (m_2 x^N + r_2)) \bmod P$$

- Suppose:

$$I = \sum_{k=0 \ldots N-1} a_k x^k$$

Then:

$$\sum_{k=0 \ldots N-1} a_k (x^k (x^{L_1} + x^{L_2}) \bmod P)$$
$$= ((m_1 x^N + r_1) + (m_2 x^N + r_2)) \bmod P$$

Solving for init

- Finding a linear combination of $(x^k(x^{L_1} + x^{L_2})) \bmod P$ summing to $((m_1 x^N + r_1) + (m_2 x^N + r_2)) \bmod P$
- It's Gaussian elimination again

Example:

```
m1, r1 = AAFF 0040 2EEC
m2, r2 = AAFF 040E02 0450
```

- $L_1 = 32$, $L_2 = 40$
- Solving gives `init = 0xffff`

Non-unique init values

- There might not be a unique solutions for I, as noted by Ewing (2010)
- Some CRC polynomials may have multiple equivalent (I, F) pairs
- These polynomials are reducible, i.e. have non-trivial factors

Firsly, two notable properties of $x + 1$

- $x^M + 1 = (x + 1)(x^{M-1} + x^{M-2} + \dots + x + 1)$
- If M is a power of 2, then $x^M + 1 = (x + 1)^M$

Non-unique init values

- If I_1 and I_2 for, then for *all* pairs of natural numbers (L_1, L_2),

$$I_1(x^{L_1} + x^{L_2}) \equiv I_2(x^{L_1} + x^{L_2}) \pmod{P}$$

Or equivalently:

$$(I_1 + I_2)(x^{L_1} + x^{L_2}) \equiv 0 \pmod{P}$$

Let $I^* = I_1 + I_2$

Conditions for non-unique init values

- For *all* pairs of natural numbers (L_1, L_2), without loss of generality assuming $L_1 > L_2$,

$$I^*(x^{L_1} + x^{L_2}) \equiv 0 \pmod{P}$$

- Bezout's theorem: for all a and b, there exists u and v such that

$$au + bv = \gcd(a, b)$$

- If $I^*a \equiv 0 \pmod{P}$ and $I^*b \equiv 0 \pmod{P}$
- Then $I^*(au + bv) \equiv 0 \pmod{P}$
- Therefore $I^* \gcd(a, b) \equiv 0 \pmod{P}$

What is the GCD then?

- We need to find I^* such that

$$I^* g \equiv 0 \pmod{P}$$

- Where g is the GCD of all polynomials of the form $x^{L_1} + x^{L_2}$

$$\begin{aligned} x^{L_1} + x^{L_2} &= (x^{L_1 - L_2} + 1)x^{L_2} \\ &= (x+1)(x^{L_1 - L_2 - 1} + \dots + 1)x^{L_2} \end{aligned}$$

- If $L_1 - L_2 = 1$ then $x^{L_1 - L_2 - 1} + \dots + 1 = 1$
- If $L_2 = 0$ then $x^{L_2} = 1$
- $g = x + 1$

If lengths are constrained

- If all lengths L are multiples of some power-of-two 'byte width', say $Z = 2^w$
  - $Z = 8$ for octets
  - $Z = 1$ for bit streams

$$
\begin{aligned}
x^{L_1} + x^{L_2} &= (x^{L_1 - L_2} + 1)x^{L_2} \\
&= (x^{(L_1 - L_2)/Z} + 1)^Z x^{L_2} \\
&= (x + 1)^Z (x^{(L_1 - L_2)/Z - 1} + \cdots + 1)^Z x^{L_2}
\end{aligned}
$$

- Similarly, we have $g = (x + 1)^Z$

Reducible poly values

- If P has a factor $(x+1)^f$, then $P/(x+1)^{\min\{f,Z\}}$ is a valid I^*
- Given valid I, any other $I + uI^*$ is also a valid initial value

Examples of reducible poly

ModBus CRC-16, $P = x^{16} + x^{15} + x^2 + 1$

- $I^* = x^{15} + x^1 + x^0$ is a multiple of $(x + 1)$
- Valid `init` values: `0xffff` (standard), `0x3ffe`
- Difference is: `0xc001`, same as noted by Ewing (2010)

Examples of reducible poly

Go `crc64` package[2], `ECMA` polynominal

- (Wrong bit order for ECMA 182 CRC-64)
- Polynomial is multiple of $(x+1)^2$
- Valid `init` values: `0xffffffffffffffff` (standard), `0x0b8fb9ee4606a6fd`, `0x71b79ae69afa0a7c`, `0x85c7dcf72303537e`

In general, there are $2^{\min\{f,Z\}}$ valid (I,F) pairs, because smallest I^* has degree $N-\min\{f,Z\}$, so $\deg u < \min\{f,Z\}$

[2] https://pkg.go.dev/hash/crc64

Solving for xorout

- Going back to any (m, r)

$$r \equiv mx^N + Ix^L + F \pmod{P}$$

- Solving for F is pretty easy now, given that we know everything else:

$$F \equiv mx^N + r + Ix^L \pmod{P}$$

- (It turns out $F = 0$ for the adjustable table)

Summary of

- Not guaranteed, but we can find CRC parameters from (m, r) pairs
  - `refin` and `refout`, only 4 possibilities
  - `width` is guessed or based on degree of GCD
  - `poly` or P determined by taking GCD, and possibly factoring
  - `init` or I solved with Gaussian elimination
    - May not be unique, can have equivalent (I, F) pairs
  - `xorout` or F computed from other parameters

Conclusion

Conclusion

- CRCs are simple
- Simple stuff can have deep theory behind it
- $\mathrm{GF}(2)$ and $\mathrm{GF}(2)[x]$ sit at the intersection of computer science, ring theory, and linear algebra
  - Useful for checking for transmission/storage errors
  - Useful for cryptography too (AES-GCM)

This talk has been literally everything I know about CRCs...

Things I still don't know

- Types of errors CRC can detect
- Picking CRC polynomials with good error detection properties...

Thanks

- Check out my blog post for some other details
- Blog: `https://dram.page/p/crc-tricks`
- Slides: `https://dram.page/p/crc-tricks/crc-tricks.pdf`

References

Cook, G. (2021a). Catalogue of parametrised CRC algorithms. https://reveng.sourceforge.io/crc-catalogue/.

Cook, G. (2021b). CRC RevEng: arbitrary-precision CRC calculator and algorithm finder. https://reveng.sourceforge.io.

Ewing, G. (2010). Reverse-engineering a CRC algorithm. https://www.cosc.canterbury.ac.nz/greg.ewing/essays/CRC-Reverse-Engineering.html.

Gopal, V., Ozturk, E., Guilford, J., Wolrich, G., Feghali, W., Dixon, M., and Karakoyunlu, D. (2009). Fast CRC computation for generic polynomials using pclmulqdq instruction. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/fast-crc-computation-generic-polynomials-pclmulqdq-paper.pdf. Intel White Paper.

References (Cont'd)

Kutenin, D. (2021). How a bug(?) in the linux CRC-32 checksum turned out not to be a bug. https://danlark.org/2021/03/08/how-a-bug-in-the-linux-crc-32-checksum-turned-out-not-to-be-a-bug/.

Williams, R. N. (1993). A painless guide to CRC error detection algorithms. https://zlib.net/crc_v3.txt.

Wolf, C. (2019). Reference implementations of various CRCs using carry-less multiply. http://svn.clairexen.net/handicraft/2018/clmulcrc/.

Wolf, C. (2021). RISC-V Bitmanip extension, document version 0.93. Technical report.