OPAQUE is **NOT** Magic

Steve "Sc00bz" Thomas

What is a PAKE?

PAKEs

- Password authentication
- Encrypted tunnels
- Sending files
  - https://github.com/magic-wormhole
- Fighting phone spoofing
  - https://commsrisk.com/?p=35506

How PAKEs Work

```
a = random()
A = a*G
```

Hide the Ephemeral Keys

```
a = random()
A = a*G+P
```

Hide the Generator

```
a = random()
A = a*P
```

Myth #1

- “Zero knowledge” means the server doesn't have a password hash

Myth #1

- “Zero knowledge” means the server doesn't have a password hash
- “Augmented PAKE for authentication: we recommend the usage of OPAQUE to avoid targeted dictionary attacks on user passwords by [the company].”

Myth #2

- OPAQUE is an augmented PAKE

Types of PAKEs

- Balanced
  - Peer-to-Peer
- Augmented (aPAKE)
  - Client-Server

Don’t call these symmetric/asymmetric

Types of PAKEs

- Balanced
  - Peer-to-Peer
- Augmented (aPAKE)
  - Client-Server
- Doubly Augmented
  - Client-Server/Device-Server
- Identity
  - IoT

Myth #3

- OPAQUE should be used for TLS because other PAKEs need to send the user name

OPRF

```
C:     P = hashToCurve(pw, id, …)
C:     r = random()
C:     R = r*P
C->S: id, R
   S: salt = dbLookup(id)
   S: R' = salt*R
C<-S: R'
C:     BlindSalt = (1/r)*R'
BlindSalt == (1/r)*r*salt*P == salt*P
```

Myth #4

- OPAQUE is the only PAKE that can prevent precomputation attacks

Myth #5

- Adding an OPRF to other PAKEs makes them much slower than OPAQUE

Costs

```
OPAQUE-3DH
C: fHI**ii ff***ix
S: f*i     ff***x
```

```
BS-SPEKE
C: fHI**ii f***xiiH
S: f*i     ff***i
```

*: Scalar point multiply

x: Scalar base point multiply

H: Hash to curve

i: Field invert

I: Scalar invert

f: From bytes

Costs

```
OPAQUE-3DH
C: fHI**ii ff***ix
S: f*i     ff***x
```

```
(strong) AuCPace
C: fHI**ii ff***iiH
S: f*i     ff***xiiH
```

*: Scalar point multiply

x: Scalar base point multiply

H: Hash to curve

i: Field invert

I: Scalar invert

f: From bytes

Costs

```
OPAQUE-3DH
C: fHI**ii ff***ix
S: f*i     ff***x
```

```
Double BS-SPEKE
C: fHI**ii f***xii*H
S: f*i     ff****i
```

*: Scalar point multiply

x: Scalar base point multiply

H: Hash to curve

i: Field invert

I: Scalar invert

f: From bytes

Costs

OPAQUE-3DH

```
C:  fHI**ii  ff***ix
S:  f*i      ff***x
```

OPAQUE-Noise-KN-No-AEAD

```
C:  fHI**ii  f**ixx
S:  f*i      ff**x
```

*: Scalar point multiply

x: Scalar base point multiply

H: Hash to curve

i: Field invert

I: Scalar invert

f: From bytes

PAKE Properties

- Fragile
- Quantum Annoying

Quantum Annoying

- "It is noted in [BM92] that if we assume that a discrete log pre-computation has been made for the modulus, a password attack must also compute the specific log for each entry in the password dictionary (until a match is found)."
  - SPEKE paper 1996
- "With EKE, the password *P* is used to superencrypt such values; it is not possible to essay a discrete logarithm calculation except for all possible guesses of *P*."
  - EKE paper 1992

Myth #6

- OPAQUE is the only one that can be made post quantum

Myth #7

- If you have an HSM that does Curve25519 but not Ristretto255, then you can't use Ristretto255

Myth #8

- You can’t inverse a clamped scalar while preserving the clamp

Clamp

ClampedScalar =

data % 2**254

+ 2**254

- data % 8

```
min = 0x4000000000000000000000000000000000000000000000000000000000000000
max = 0x7ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff8
```

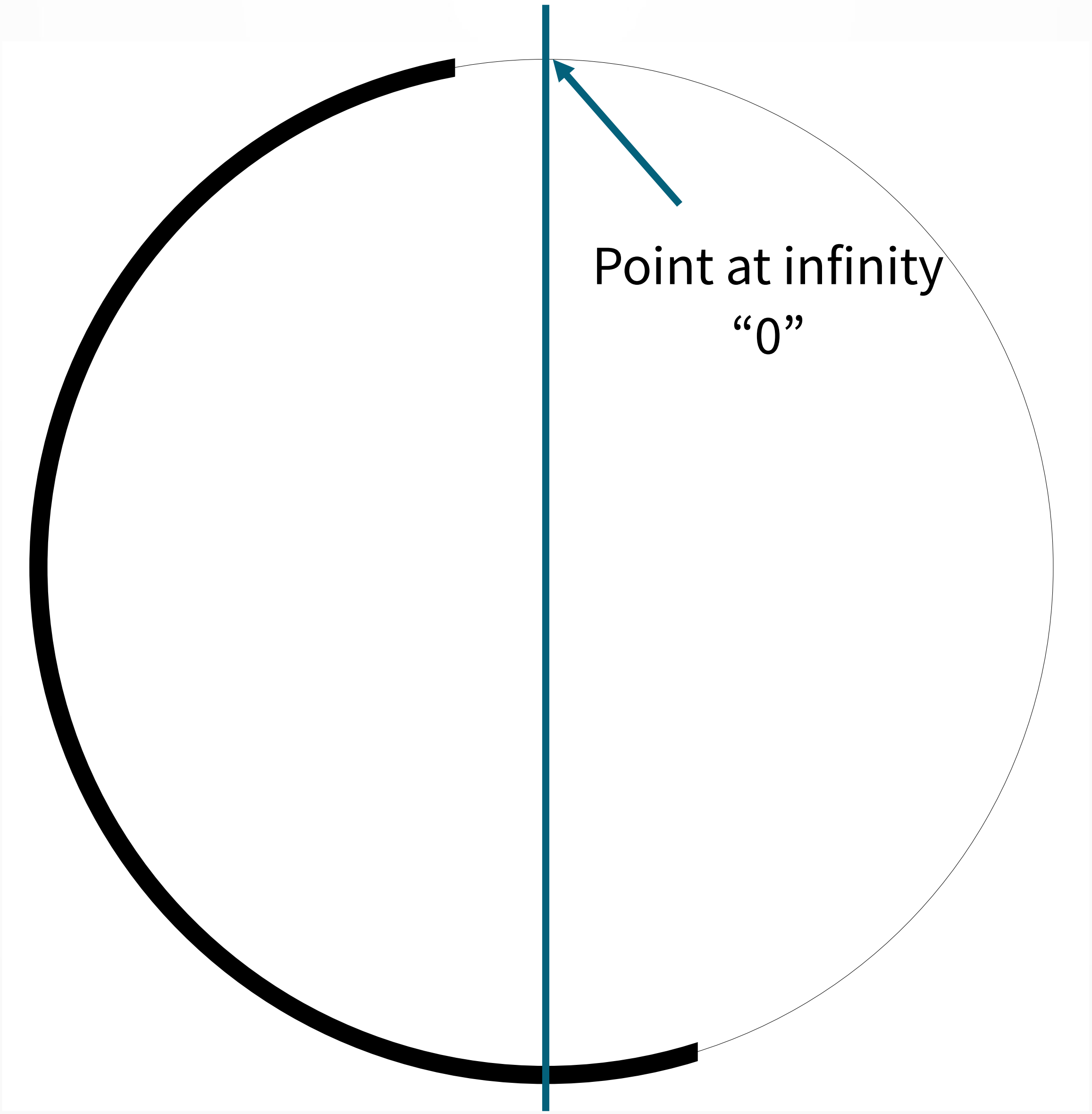
Clamped Scalar Inverse (Curve25519)

- Prime sub group
  - L = 2**252 + 0x14def9dea2f79cd65812631a5cf5d3ed
- Normal scalar inverse
  - ScalarInverse = power_mod(Scalar, L - 2, L)

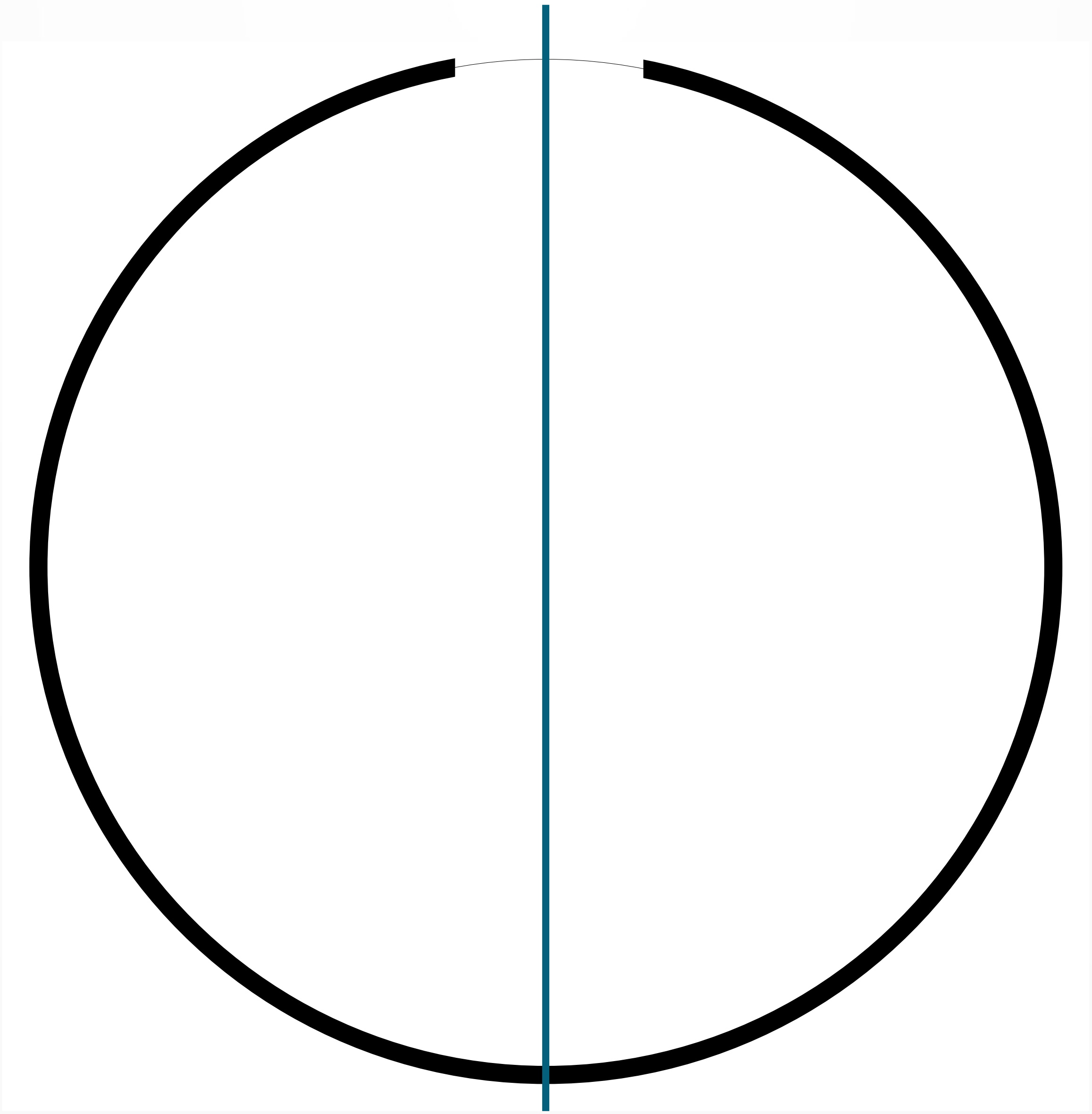
Clamped Scalar Inverse (Curve25519)

- Prime sub group
  - L = 2**252 + 0x14def9dea2f79cd65812631a5cf5d3ed
- ScalarInverse1 = power_mod(Scalar, L - 2, 8 * L)
- ScalarInverse2 = 8 * L – ScalarInverse1
- checkBit(ScalarInverse1, 254) != 0
  - ScalarInverse1
  - Otherwise ScalarInverse2

Clamped Scalar Inverse Failure



Clamped Scalar Inverse Failure



Clear the Cofactor (Curve25519)

- Prime sub group
  - L = 2**252 + 0x14def9dea2f79cd65812631a5cf5d3ed
- Multiply scalar by “inverse of 8 multiplied 8”
  - That’s “8/8” which is “1”
- Clear = power_mod(8, L - 2, L) * 8
- NewScalar = Clear * Scalar (mod 8*L)

“Myth #9”

- OPAQUE's Footgun

“Myth #9”

- I found one in the wild
- They now wrap ChaChaPoly1305 with HMAC-SHA512

Cheat Sheet

- Balanced
  - CPace
- Augmented
  - BS-SPEKE
- Doubly Augmented
  - Double BS-SPEKE
- Identity
  - CHIP

- Balanced PAKEs don't need key stretching
- bscrypt (minimums)
  - m=256 (256 KiB), t=8, p=1
  - m=256 (256 KiB), t=4, p=2
  - m=256 (256 KiB), t=3, p=3
  - General
    - m=highest per core cache level in KiB
    - t≥max(2, 1900000/1024/m/p)
    - p≤cores

What is bscrypt?

- See BSidesLV 2022 (PasswordsCon track)
  - “bscrypt – A Cache Hard Password Hash”

Minimum Password Settings

https://tobtu.com/minimum-password-settings/

Questions?

- Twitter: @Sc00bzT
- Github: Sc00bz
- steve at tobtu.com

References

[1] Send files https://github.com/magic-wormhole

[2] Phone spoofing https://commsrisk.com/?p=35506

[3] SPEKE https://jablon.org/jab96.pdf / https://jablon.org/jab97.pdf

[4] Doubly Augmented https://moderncrypto.org/mail-archive/curves/2015/000424.html

[5] (strong) AuCPace https://ia.cr/2018/286

[6] OPAQUE https://ia.cr/2018/163

[7] CHIP, CRISP https://ia.cr/2020/529

[8] EKE https://www.cs.columbia.edu/~smb/papers/neke.pdf

[9] OPAQUE's footgun https://ia.cr/2020/1491

Bonus Slides

- bscrypt
- BS-SPEKE secure registration

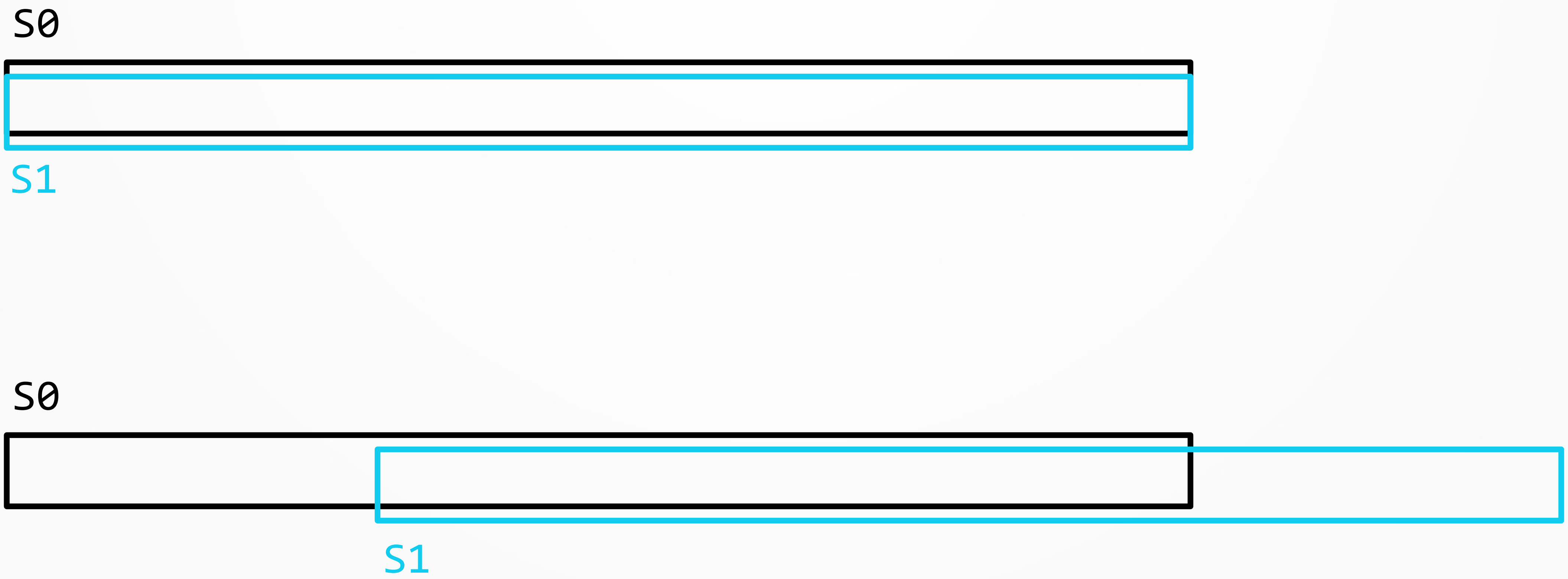
bscrypt

- The fun slides from my BSidesLV 2022 talk
- But first one info slide

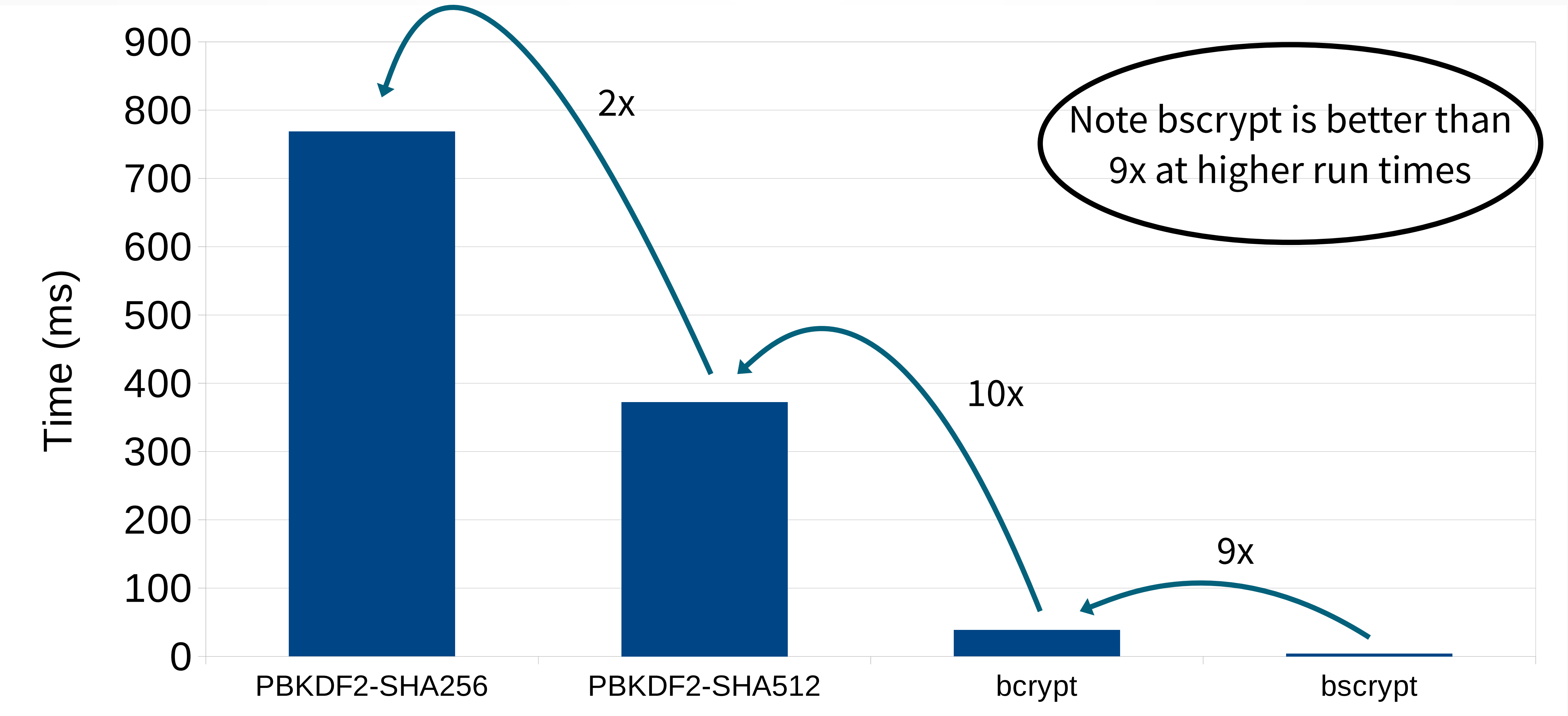
Accumulators

```
R ^= sbox0[L >> 32 & mask];
R += sbox1[L       & mask];
L ^= sbox0[R >> 32 & mask];
L += sbox1[R       & mask];
...
```

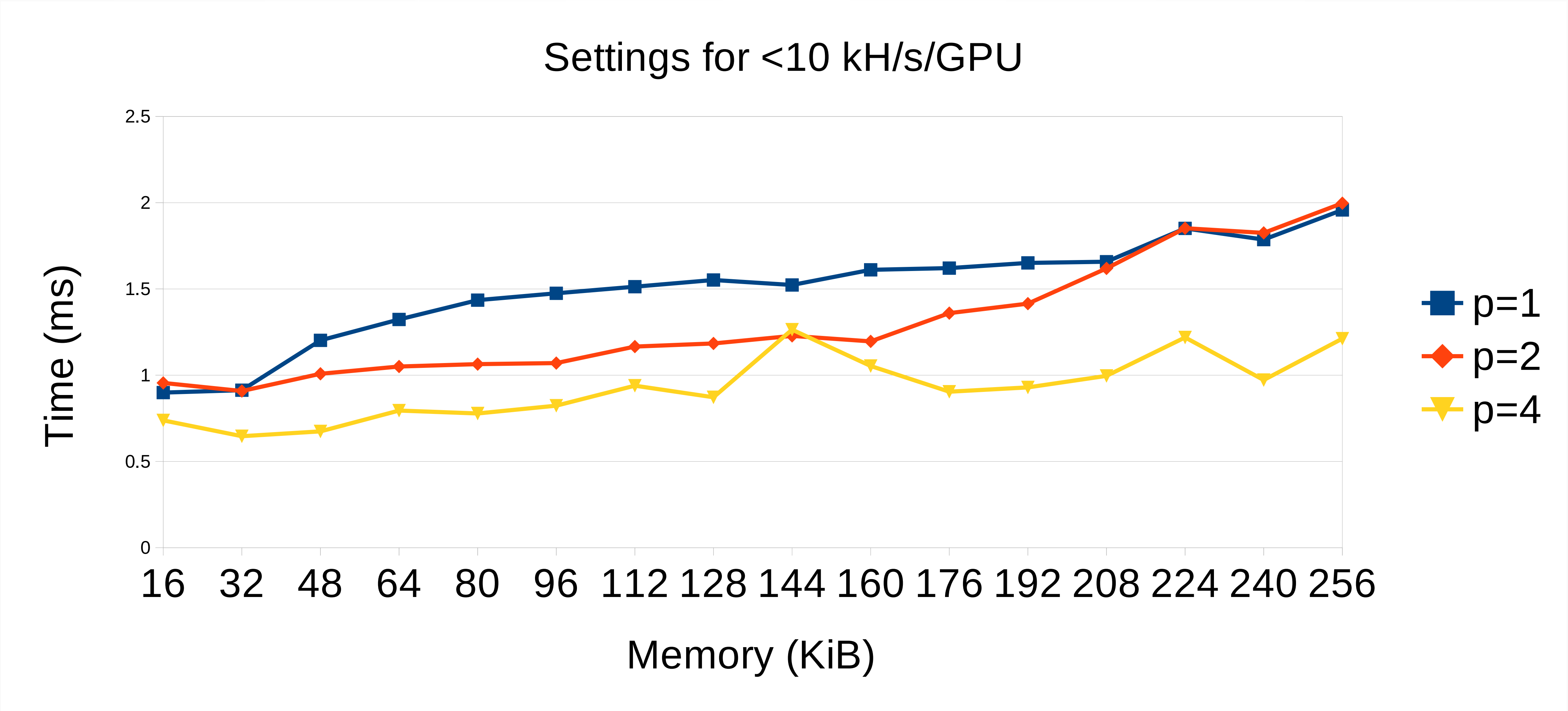
Overlapping S-boxes



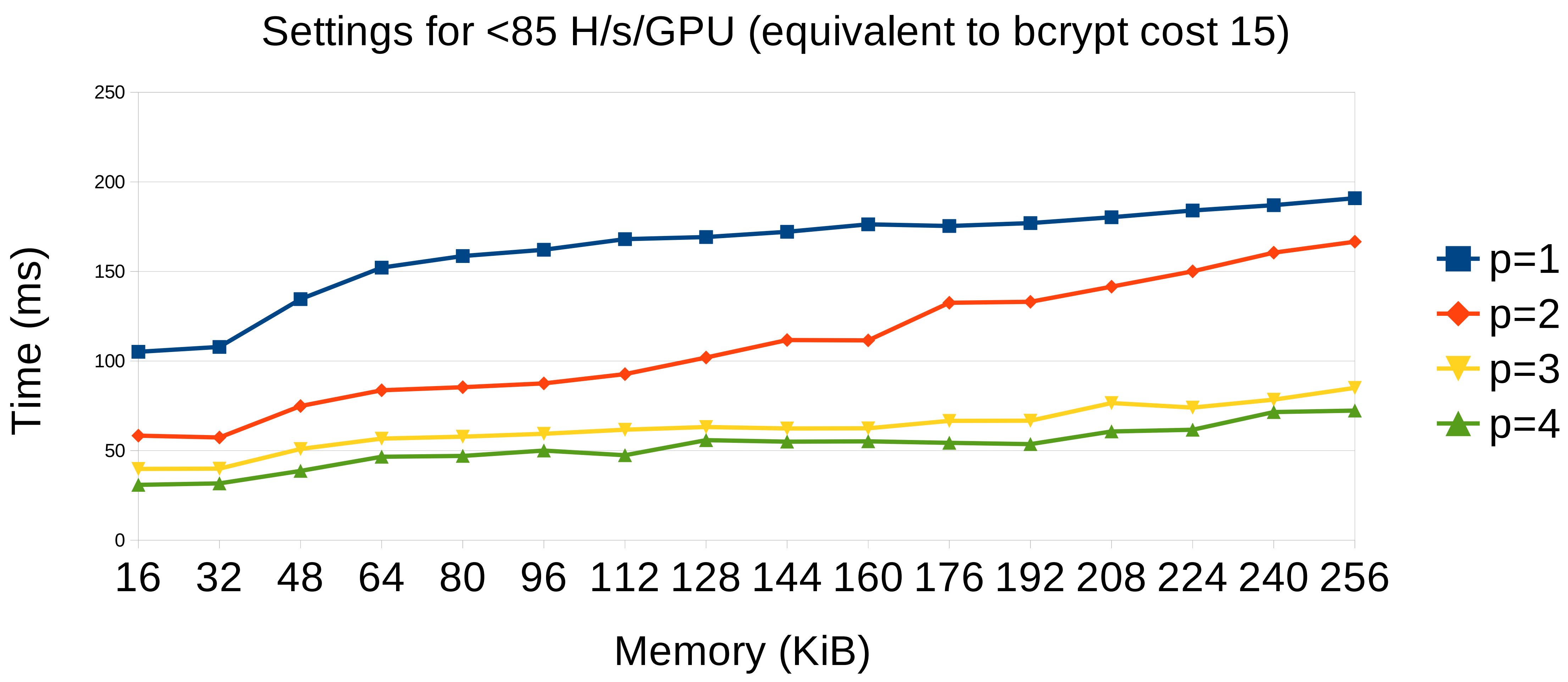
i5-6200U: Settings for ~5300 KH/s/GPU



i5-6500: 32 KiB L1, 256 KiB L2, 6 MiB L3



i5-6500: 32 KiB L1, 256 KiB L2, 6 MiB L3



BS-SPEKE Secure Registration

```
    S: Check client verifier
    S: verifierS  = H(...)
    S: sessionKey = H(...)
    S: encReg = aead_encrypt(sessionKey, reg || regMac)
 C<-S: verifierS, encReg
 C:    Check server verifier
 C:    sessionKey = H(...)
 C:    reg || regMac = aead_encrypt(sessionKey, encReg)
 C:    Checks regMac == MAC(macKey, reg * G)
```