Cryptography Vulnerabilities in the Wild

Swiss Crypto Day – ETHZ – September 8th 2023

Paul Bottinelli [paul.bottinelli@nccgroup.com] - Cryptography Services

nccgroup

Cryptography reviews: where do we find vulns?

- Randomness
- Algorithm Confusion
- Key Management
- Cryptographic Primitives
- Protocols
- Side-channel Attacks

Cryptography reviews: where do we find vulns?

Randomness

Algorithm Confusion

Key Management

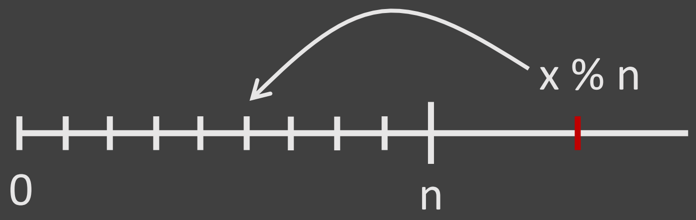
Cryptographic Primitives

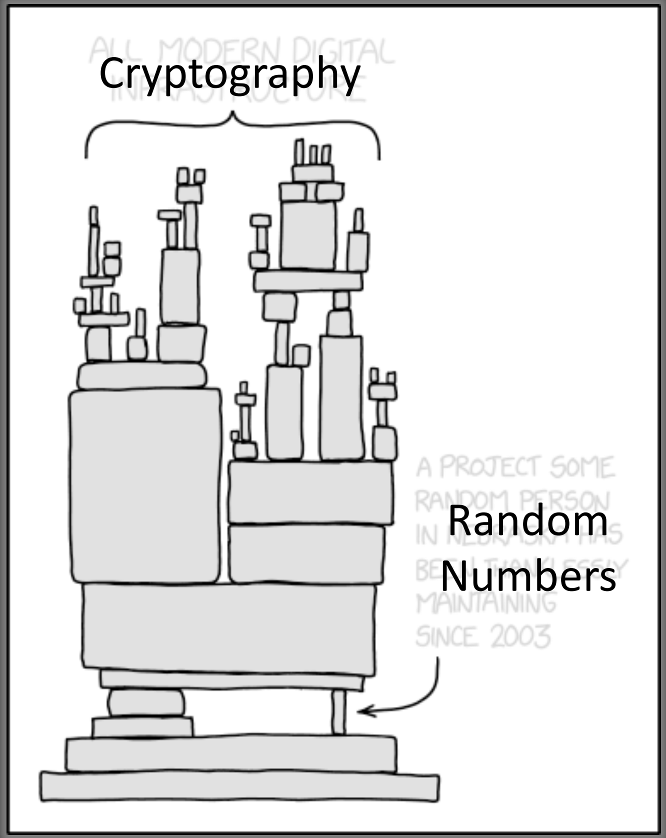
Protocols

Side-channel Attacks

Randomness

- Incorrect usage or seeding of PRNG
- Usage of bad random numbers for other purposes than private keys
- Modulo bias issues

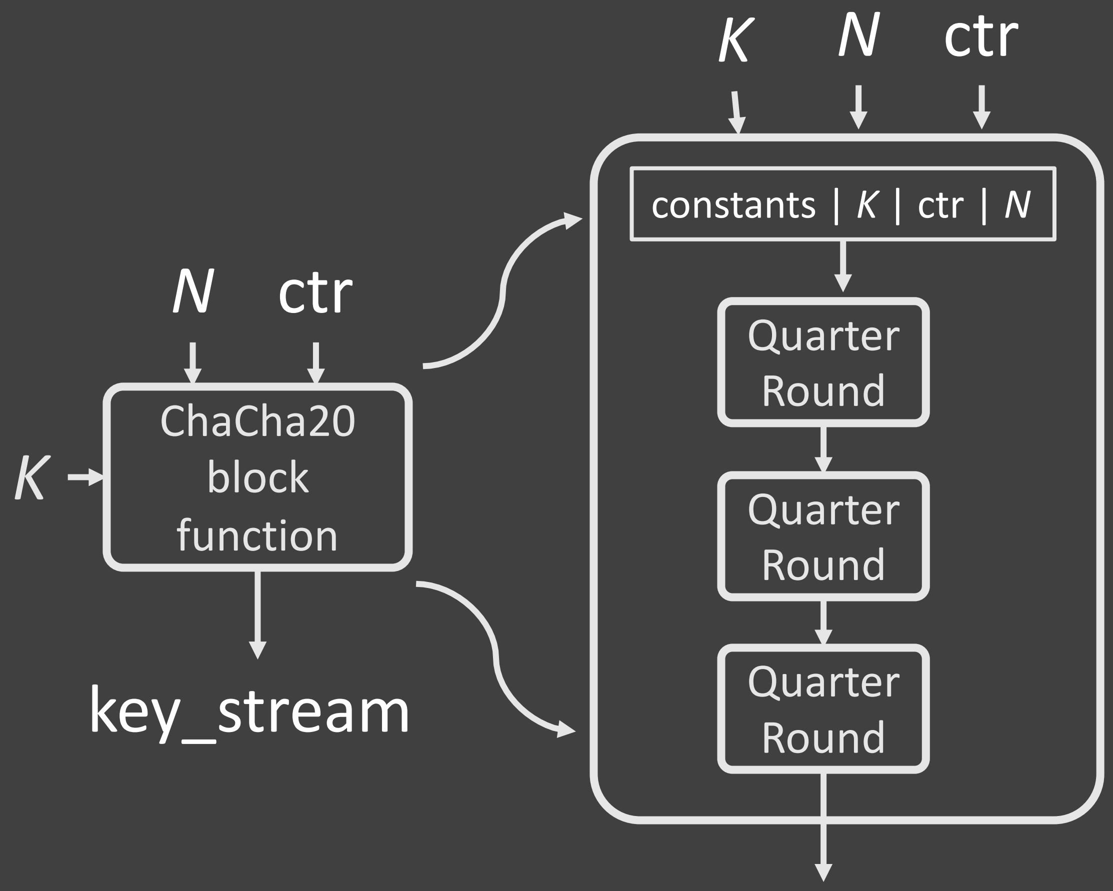




Adapted from https://xkcd.com/2347/

Spotlight – Strong Biases in ChaCha20 PRNG

- ChaCha20 is a stream cipher
- Block function generates a key stream; encryption works by XOR-ing key stream and plaintext
- The ChaCha block function transforms a state by performing a sequence of *quarter rounds*
- The block function is often used as a PRF



Spotlight – Strong Biases in ChaCha20 PRNG

Review of a Java implementation

→ bytes are in [-128, 127]

```java
@Test
public void outputExpected() throws Exception {
    byte[] seed = {
          3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
    };
    byte[] expected1 = {
           -23, 10, 77, -86, 22, 44, -100, 103, -85, -32, 33,
    };
    byte[] expected2 = {
           7,-78,14,-84,-100,53,35,-110,85,10,79,49,-78,-21,-3
    };
```

```
2.1.  The ChaCha Quarter Round

   1.  a += b; d ^= a; d <<<= 16;
   2.  c += d; b ^= c; b <<<= 12;
   3.  a += b; d ^= a; d <<<= 8;
   4.  c += d; b ^= c; b <<<= 7;
```

Source: https://www.rfc-editor.org/rfc/rfc7539

Spotlight – Strong Biases in ChaCha20 PRNG

Review of a Java implementation

→ bytes are in [-128, 127]

2.1. The ChaCha Quarter Round

```
@Test
public void outputExpected() throws Exception {
    byte[] seed = {
            3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3
    };
    byte[] expected1 = {
             -23, 10, 77, -86, 22, 44, -100, 103, -85, -32, 33, -109, -98, -21, -48, -50, -74, -1, -1, -1, -85, -1, -1, -1,
    };
    byte[] expected2 = {
             7,-78,14,-84,-100,53,35,-110,85,10,79,49,-78,-21,-38,28,96,99,-59,16,-26,-1,-1,-1,-100,-1,-1,-1,-73,-1,-1,-1
    };
```

Spotlight – Strong Biases in ChaCha20 PRNG

```java
/**
 * @param x The value to be rotated
 * @param k The number of bits to rotate left
 */
private static int rotateLeft32(int x, int k) {
    final int n = 32;

    int s = k & (n - 1);
    return x << s | x >>(n - s);
}
```

Signed right shift!!

- Java does not have primitive types for unsigned integers, values are *interpreted in two’s complement*
- the “>>” operator performs a *signed right shift.* The sign bit is maintained and not unconditionally replaced by a zero
- 0b1000 >>> 1 = 0b0100 (unsigned right shift)
- 0b1000 >> 1 = 0b1100 (signed right shift)

Spotlight – Strong Biases in ChaCha20 PRNG

Example:

```
rotateLeft32(0x80000000, 16)
 = 0b100…000 << 16 | 0b100…000 >> 16
 = 0b000…000 | 0b110…000 >> 15
 = …
 = 0b0…0 | 0b11…1100…00
 = 0xFFFF8000 = {-1, -1, -128, 0}
```

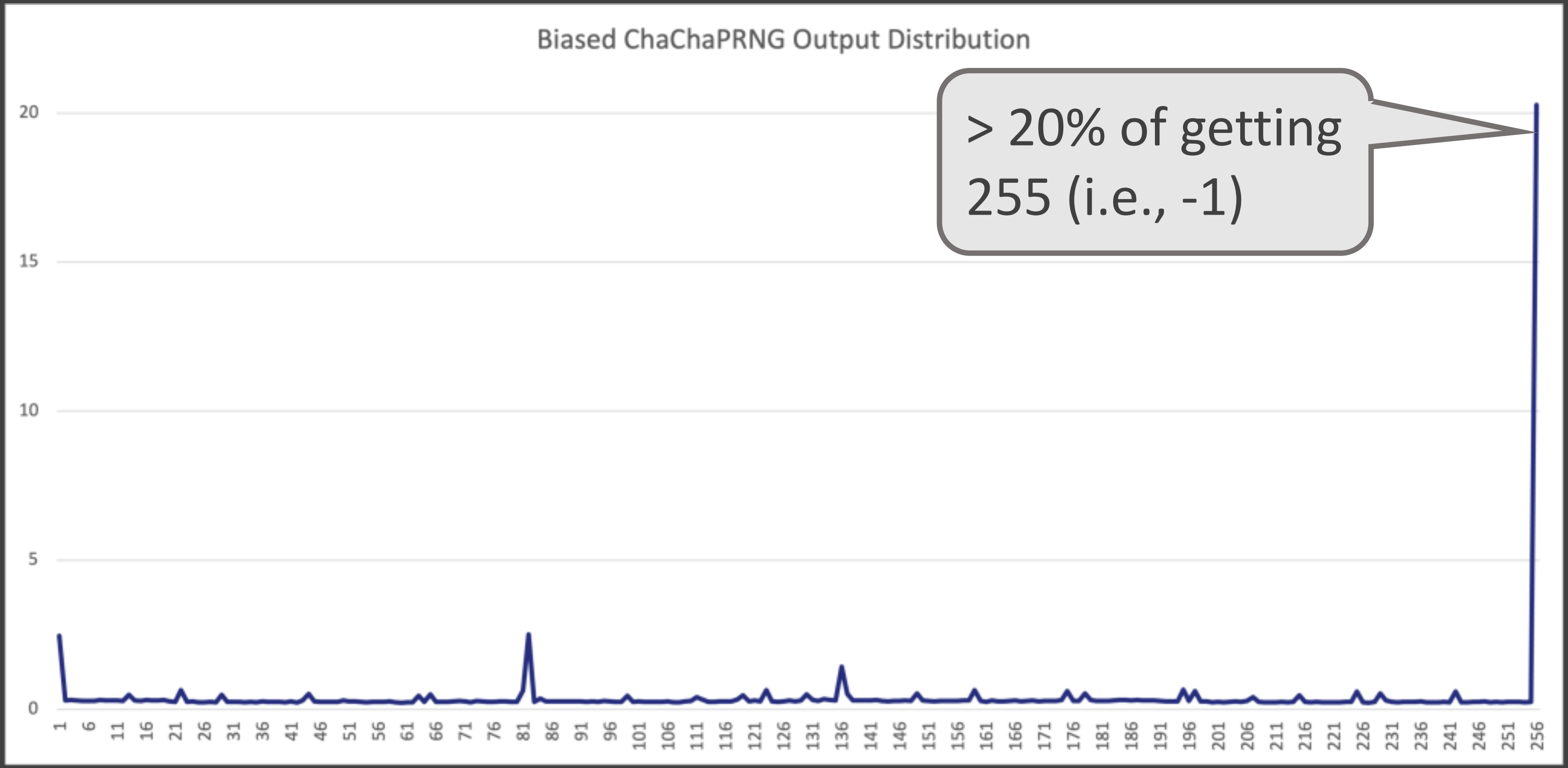
From 1 “one-bit” to 17!

```java
private static int rotateLeft32(int x, int k) {
    final int n = 32;

    int s = k & (n - 1);
    return x << s | x >>(n - s);
}
```

nccgroup

Spotlight – Strong Biases in ChaCha20 PRNG

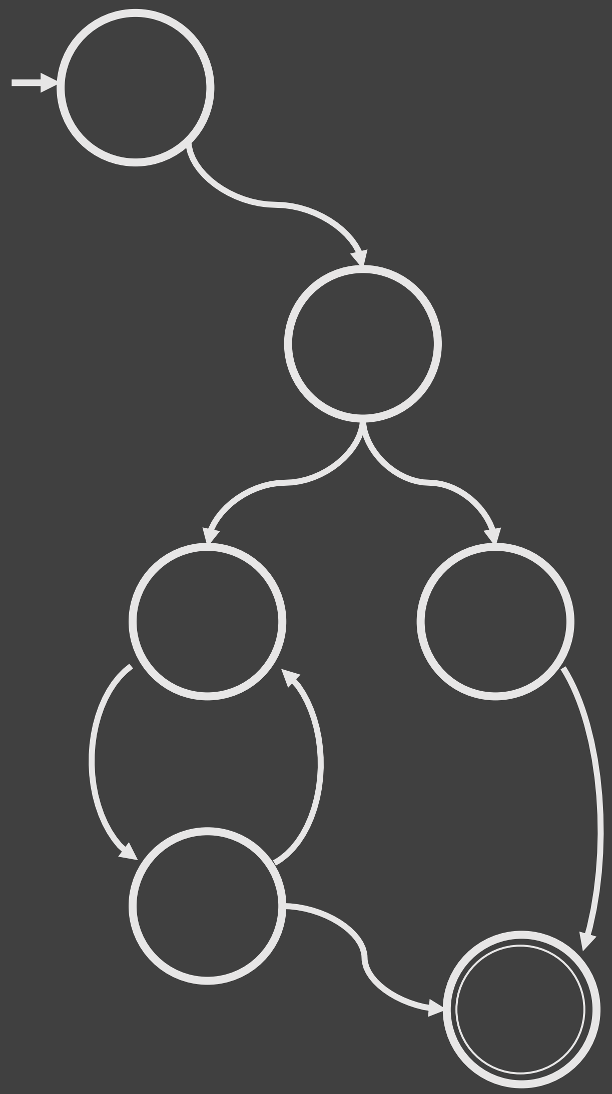


Lessons:

1. Language translation can be tricky
2. Test vectors are important

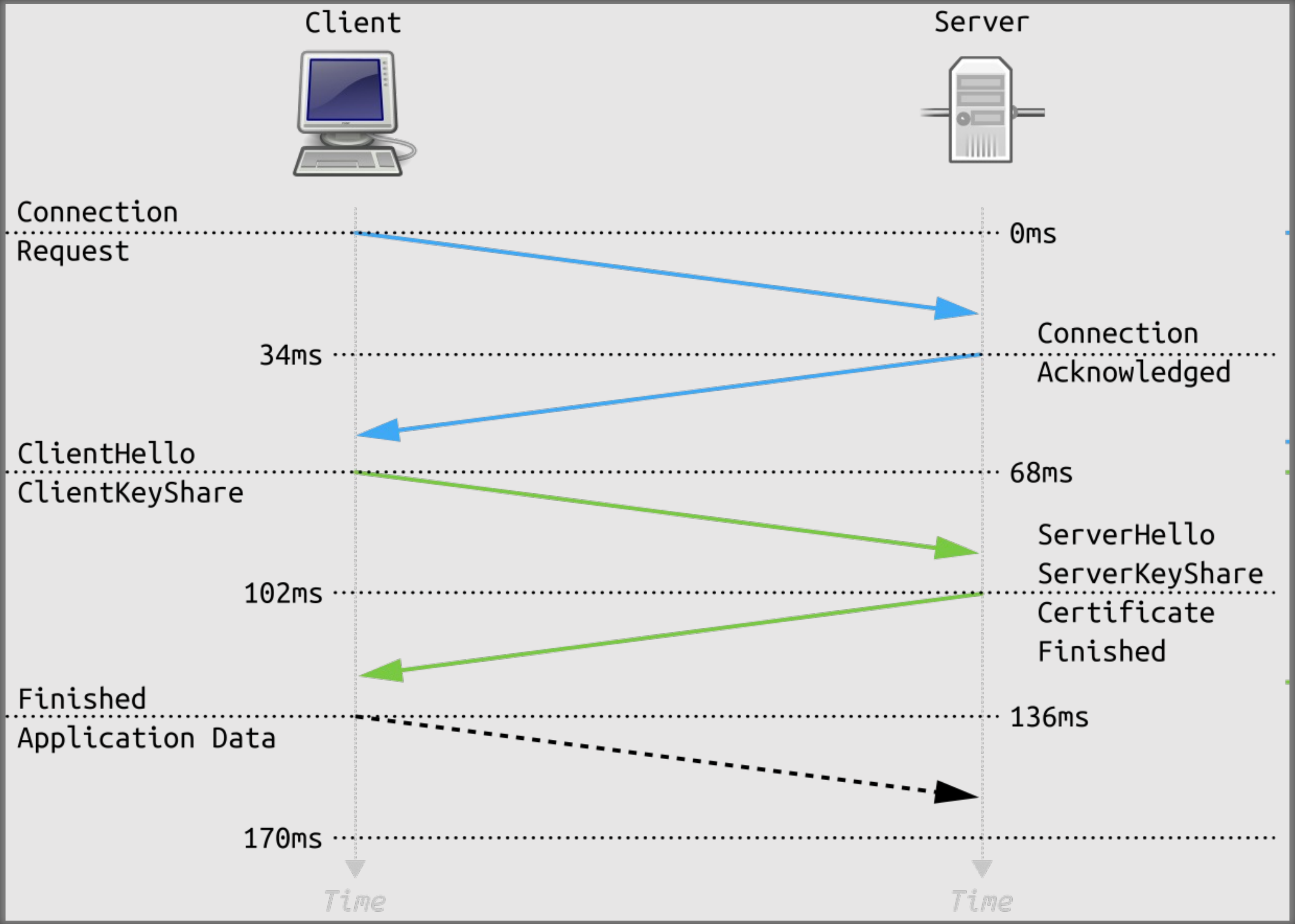
Protocols

- Protocols are complex state machines
- Message flow tampering (dropped, injected, re-ordered, repeated, or modified messages)
- Backwards compatibility and downgrade attacks
- Interoperability with other implementations
- Missing input validation!



Spotlight – TLS 1.3 Client MitM

- TLS 1.3 is specified in RFC 8446
- Used to establish a secure channel between Client and Server
- Starts with a handshake (parameter negotiation), where Client authenticates Server with a Certificate

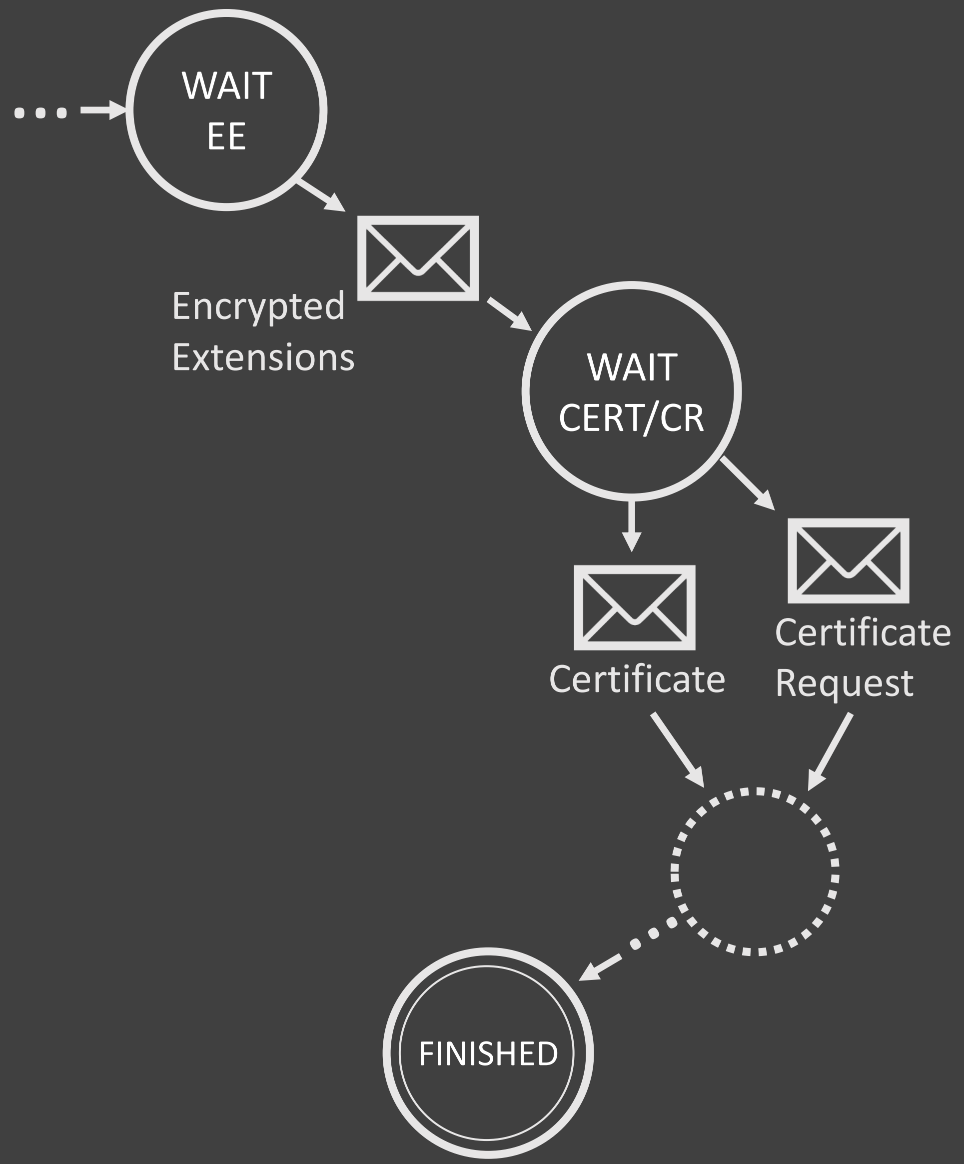


Source: https://commons.wikimedia.org/wiki/File:Full_TLS_1.3_Handshake.svg

Spotlight – TLS 1.3 Client MitM

RFC 8446 defines following handshake messages

```
select (Handshake.msg_type) {
    case client_hello:           ClientHello;
    case server_hello:           ServerHello;
    case end_of_early_data:      EndOfEarlyData;
    case encrypted_extensions:   EncryptedExtensions;
    case certificate_request:    CertificateRequest;
    case certificate:            Certificate;
    case certificate_verify:     CertificateVerify;
    case finished:               Finished;
    case new_session_ticket:     NewSessionTicket;
    case key_update:             KeyUpdate;
```

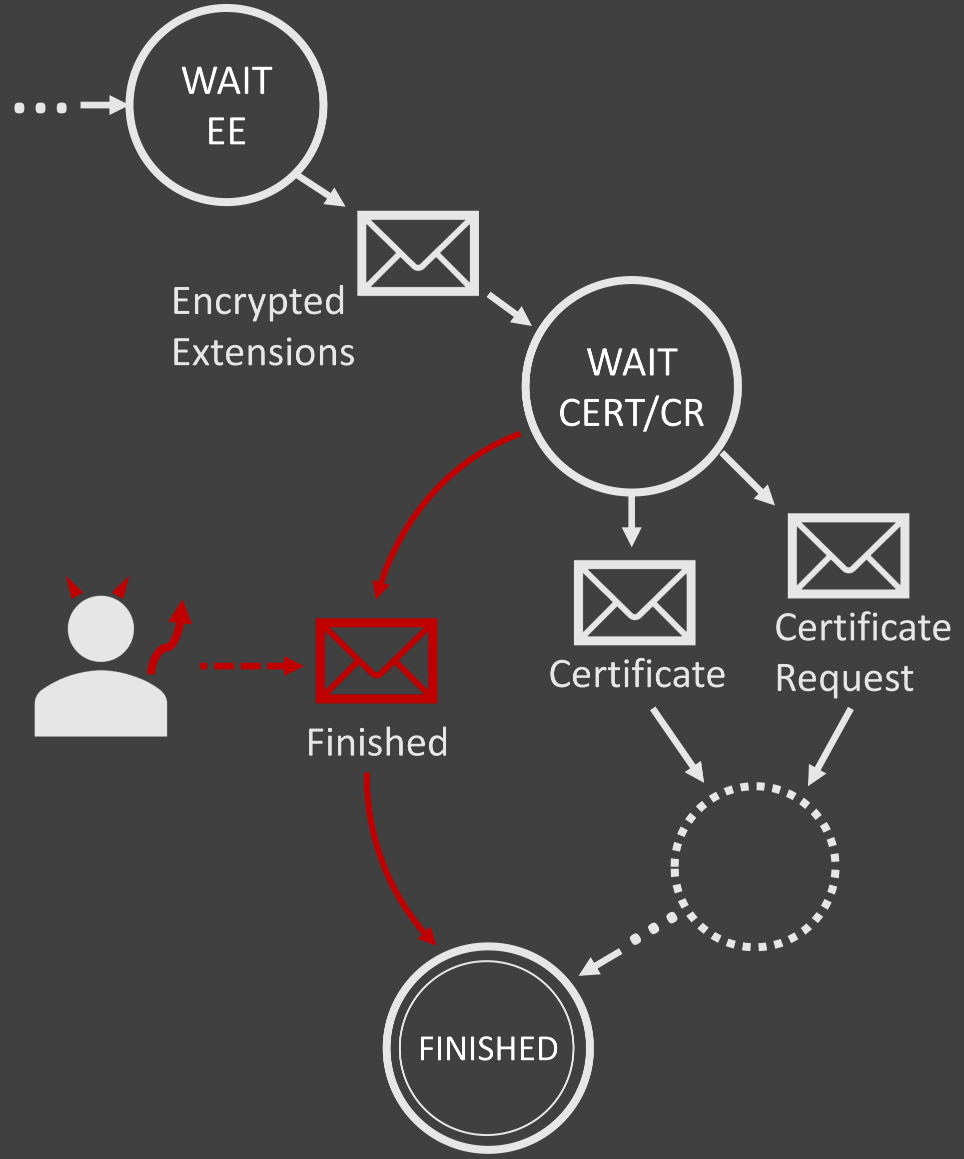


Spotlight – TLS 1.3 Client MitM

RFC 8446 defines following handshake messages

```
select (Handshake.msg_type) {
    case client_hello:          ClientHello;
    case server_hello:          ServerHello;
    case end_of_early_data:     EndOfEarlyData;
    case encrypted_extensions:  EncryptedExtensions;
    case certificate_request:   CertificateRequest;
    case certificate:           Certificate;
    case certificate_verify:    CertificateVerify;
    case finished:              Finished;
```

- Only “CertificateRequest” or “Certificate” messages are valid from WAIT_CERT_CR state
- Bug in TLS 1.3 client state machine:
  - Implementation accepted “Finished” message!
- Attacker can impersonate any TLS 1.3 Server



Cryptographic Primitives

Symmetric Cryptography

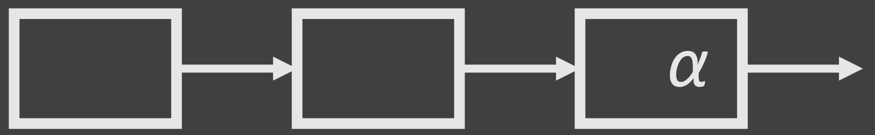
- Use of non-authenticated modes and ciphers
- Use of custom “encrypt + MAC” constructions instead of AEAD modes (e.g., GCM)
- IV/Nonce considerations
- Key usage limits and IV wrap-around

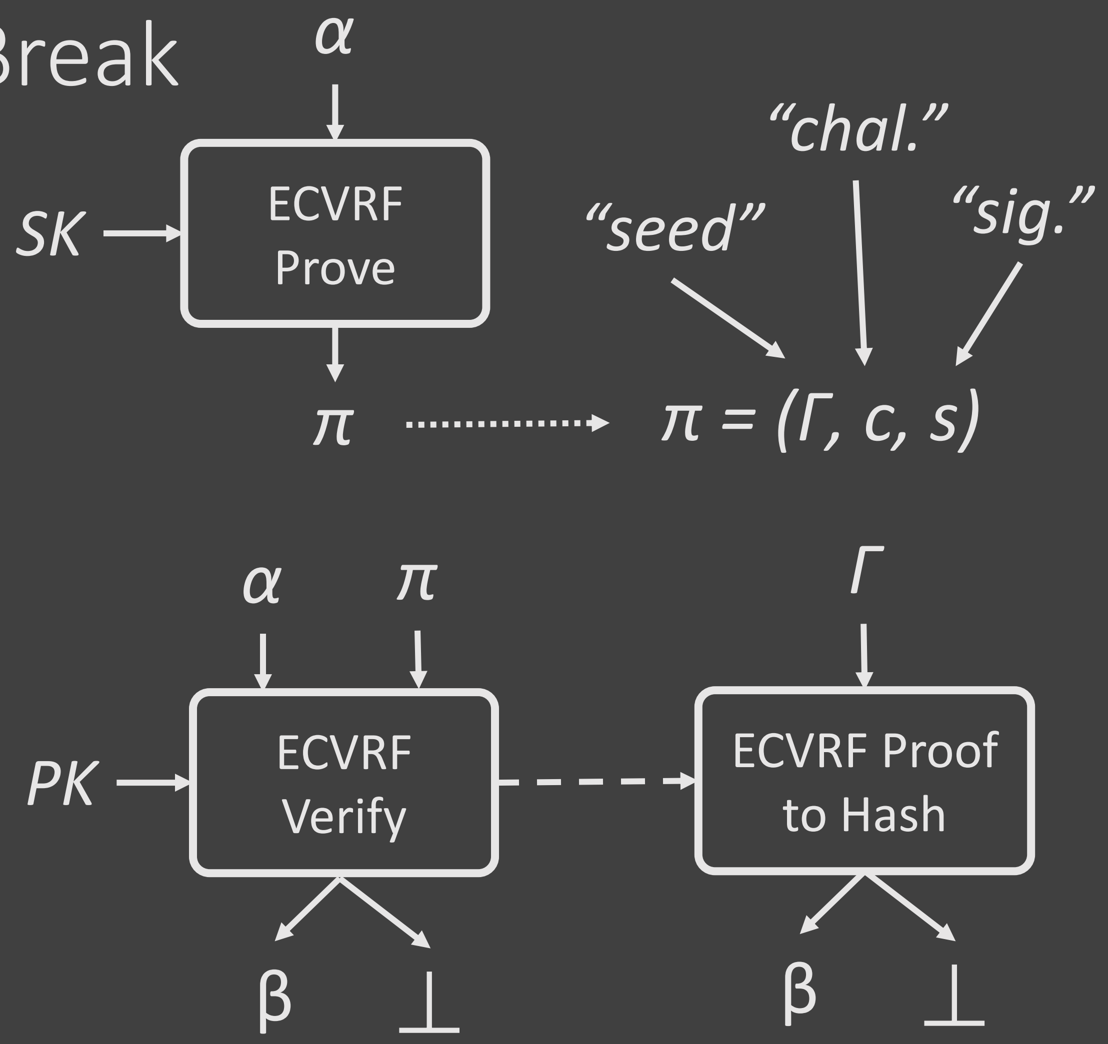
Asymmetric Cryptography

- Vulnerable use of RSA
- Missing elliptic curve public key validation
- ECDSA-specific issues
- Modern primitives (VRFs, ZK Proofs, etc)

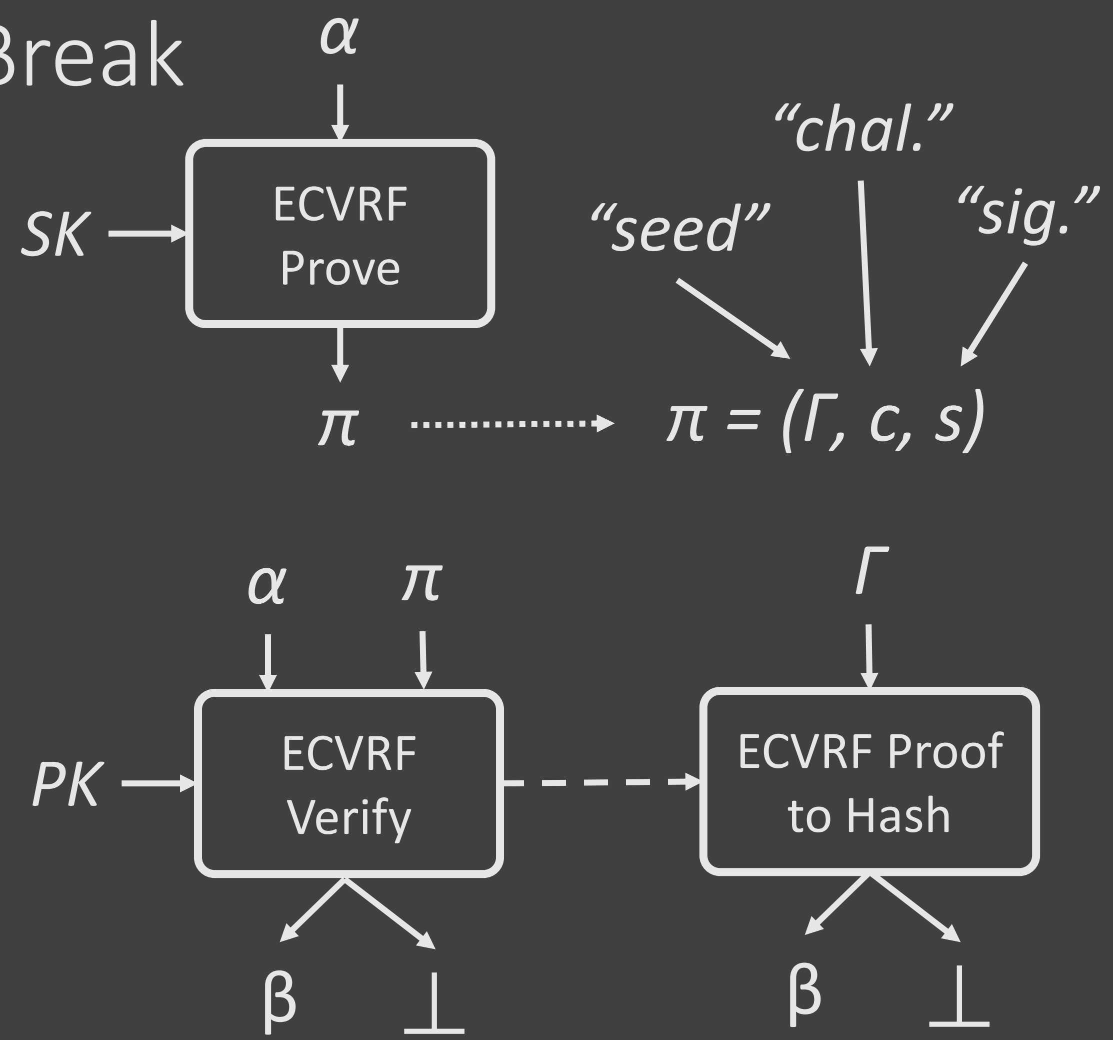
Spotlight – VRF Uniqueness Break

- Verifiable Random Functions (VRF) are the public key version of a keyed cryptographic hash
  https://datatracker.ietf.org/doc/rfc9381/
- 3 main API functions + a few other auxiliary functions
- Used in PoS blockchains, e.g., for leader election. Validators compute a VRF output and the lowest is the leader.





Spotlight – VRF Uniqueness Break

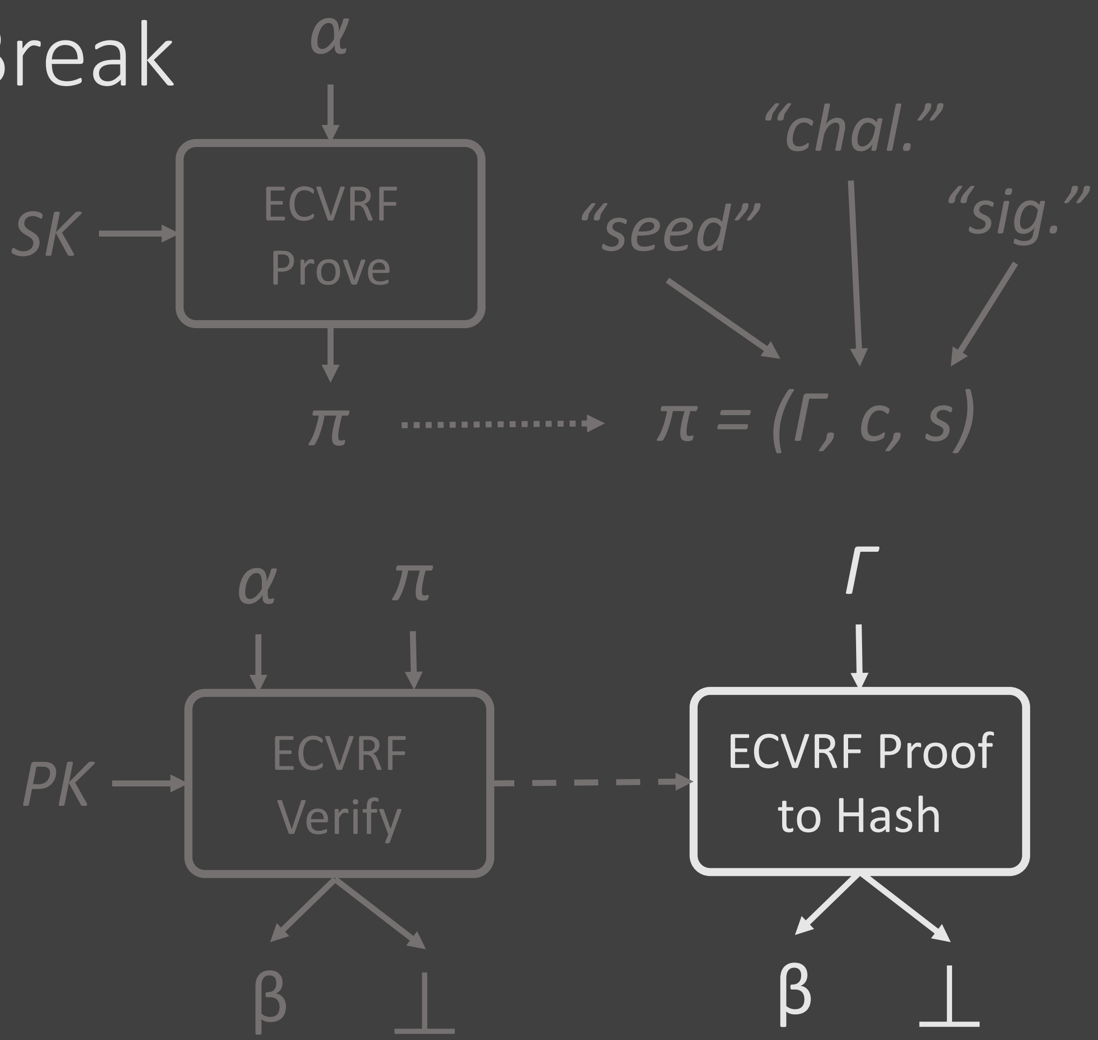


Security Properties

- **Uniqueness:** for any fixed VRF public key and for any input α, it is infeasible to find proofs for more than one VRF output β.
- **Collision resistance:** it is infeasible to find two different inputs α_1 and α_2 with the same output β.
- **Pseudorandomness:** ensures that when someone who does not know *SK* sees a hash output β (without its corresponding proof π, β is indistinguishable from a random value.

Spotlight – VRF Uniqueness Break

- Note that the final randomness exclusively depends on Γ; $\beta \approx H(\Gamma)$
- Creating a different Γ' (Gamma) for which (c', s') forms a valid proof breaks uniqueness



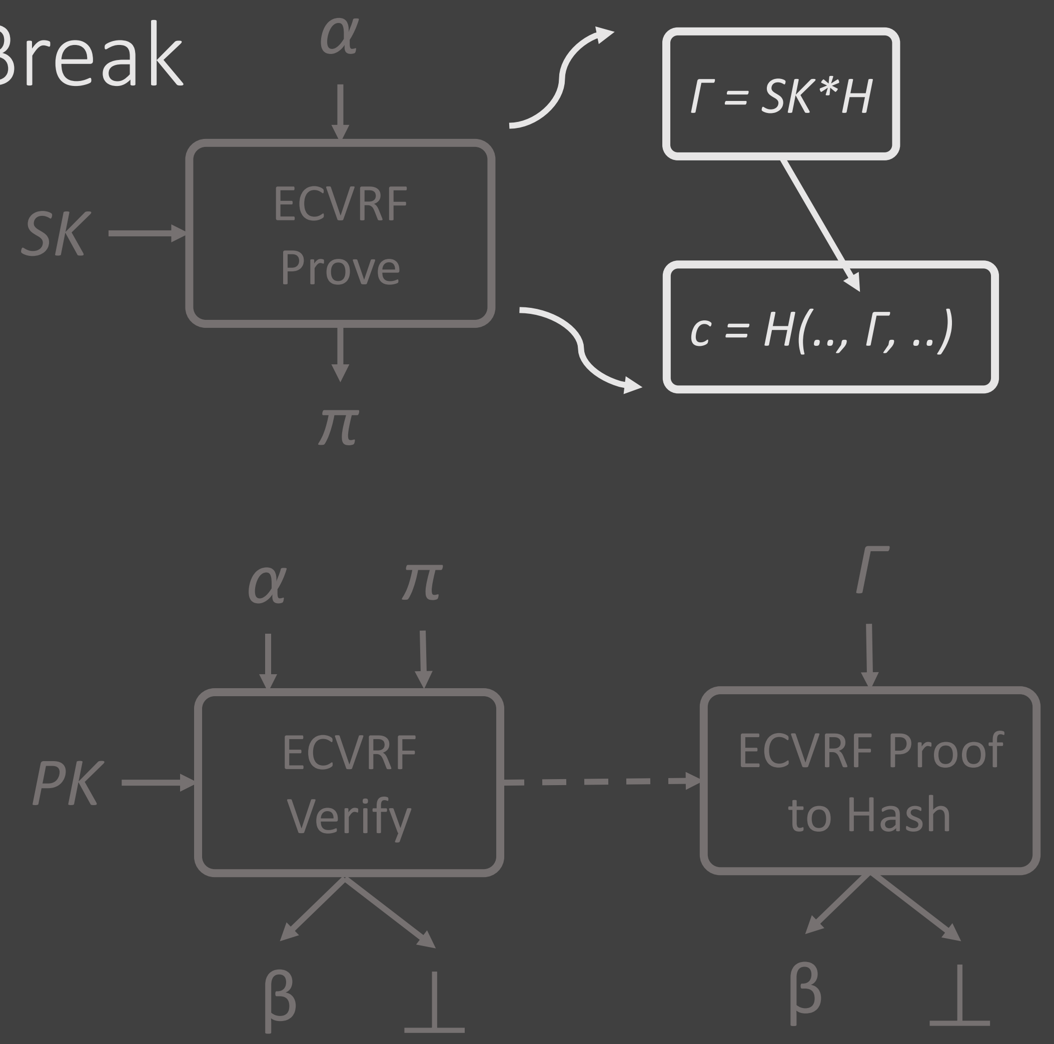
nccgroup

Spotlight – VRF Uniqueness Break

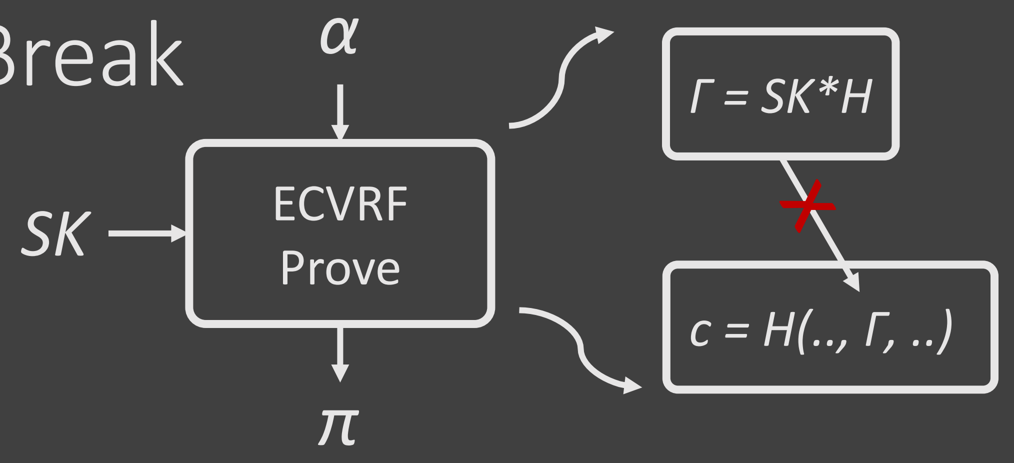
- Note that the final randomness exclusively depends on *Γ*; $\beta \approx H(\Gamma)$
- Creating a different *Γ’* (Gamma) for which *(c’, s’)* forms a valid proof breaks uniqueness
- ECVRF Prove:

  …
  3. H = encode(α)
  *4. Gamma = SK*H*
  5. Generate nonce k
  *6. c = H(PK, H, Gamma, k*G, k*H)*
  *7. s = (k + c*SK) mod q*
  …

c depends on Gamma



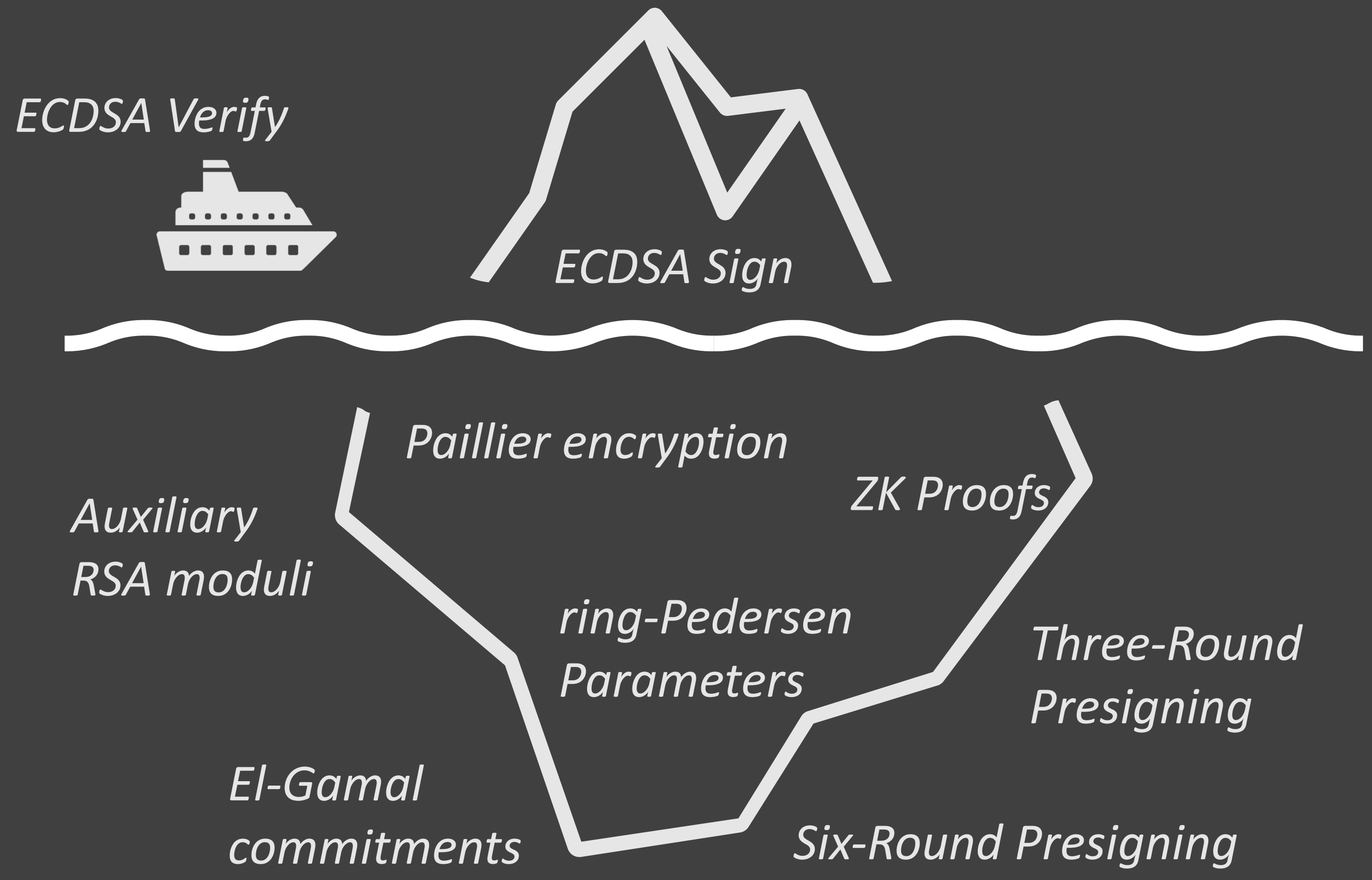
Spotlight – VRF Uniqueness Break



- We reviewed an implementation that omitted *Gamma* from the challenge computation (on purpose)
  - $c = H(PK, H, k*G, k*H)$
- *c* no longer depends on *Gamma*
- A malicious prover can generate arbitrary proofs that will correctly verify under their public key
- For an arbitrary *k*, this is a valid proof
- Malicious validator can rig the vote

$$\text{this} \left\{ \begin{array}{l} c = H(PK, H, (k + 1)*G, k*H) \\ \Gamma = (SK + c^{-1})*H \\ s = k + (SK + c^{-1})*c \end{array} \right.$$

Spotlight – ECDSA Signature Verification Bypass



- Bypass of ECDSA verification if standard not implemented properly
- Common in embedded systems with custom implementations in firmware
- Resurgence in Threshold ECDSA implementations

Spotlight – ECDSA Signature Verification

```python
def ecdsa_verify(self, e=hash, signature):
    G = self.generator
    n = G.order()
    r = signature.r
    s = signature.s
    w = inverse_mod(s, n)
    u1 = (e * w) % n
    u2 = (r * w) % n
    X = u1 * G + u2 * self.public_key
    v = X.x() % n

    return v == r
```

ECDSA SIGNATURE VERIFICATION. To verify A's signature (r, s) on m, B obtains an authentic copy of A's domain parameters $D = (q, \mathrm{FR}, a, b, G, n, h)$ and associated public key Q. It is recommended that B also validates D and Q (see §5.4 and §6.2). B then does the following:

1. Verify that r and s are integers in the interval $[1, n-1]$.
2. Compute SHA-1(m) and convert this bit string to an integer e.
3. Compute $w = s^{-1} \bmod n$.
4. Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
5. Compute $X = u_1 G + u_2 Q$.
6. If $X = \mathcal{O}$, then reject the signature. Otherwise, convert the x-coordinate x_1 of X to an integer $\overline{x}_1$, and compute $v = \overline{x}_1 \bmod n$.
7. Accept the signature if and only if $v = r$.

ANSI X9.62 ECDSA Signature Verification

Spotlight – ECDSA Signature Verification

```python
def ecdsa_verify(self, e=hash, signature):
    G = self.generator
    n = G.order()
    r = signature.r
    s = signature.s
    w = inverse_mod(s, n)
    u1 = (e * w) % n
    u2 = (r * w) % n
    X = u1 * G + u2 * self.public_key
    v = X.x() % n

    return v == r

```

ECDSA SIGNATURE VERIFICATION. To verify A's signature (r, s) on m, B obtains an authentic copy of A's domain parameters $D = (q, \mathrm{FR}, a, b, G, n, h)$ and associated public key Q. It is recommended that B also validates D and Q (see §5.4 and §6.2). B then does the following:

1. Verify that r and s are integers in the interval $[1, n-1]$.
2. Compute SHA-1(m) and convert this bit string to an integer e.
3. Compute $w = s^{-1} \bmod n$.
4. Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
5. Compute $X = u_1 G + u_2 Q$.
6. If $X = \mathcal{O}$, then reject the signature. Otherwise, convert the x-coordinate x_1 of X to an integer $\overline{x}_1$, and compute $v = \overline{x}_1 \bmod n$.
7. Accept the signature if and only if $v = r$.

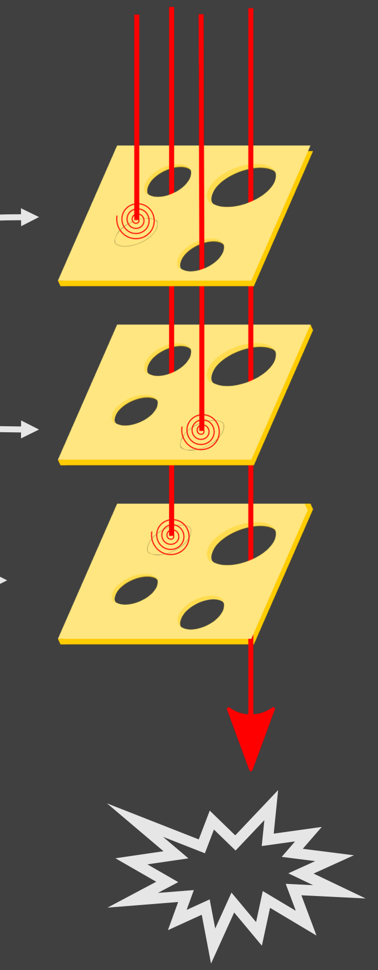
ANSI X9.62 ECDSA Signature Verification

Spotlight – ECDSA Signature Verification

Source: https://en.wikipedia.org/wiki/Swiss_cheese_model

```python
def ecdsa_verify(self, e=hash, signature):  # signature = (0,0)
    G = self.generator
    n = G.order()
    r = signature.r                          # r  = 0
    s = signature.s                          # s  = 0
    w = inverse_mod(s, n)                    # w  = s^(-1)    = 0
    u1 = (e * w) % n                         # u1 = e * 0     = 0
    u2 = (r * w) % n                         # u2 = 0 * 0     = 0
    X = u1 * G + u2 * self.public_key        # X = 0G + 0P    = 0
    v = X.x() % n                            # v = 0.x        = 0

    return v == r                            # 0 == 0 is always true
```



Easy signature verification bypass: *σ = (0, 0)* is a valid signature for *any* message and *any* public key!

Conclusion

- Presented a wide range of cryptography issues
- Validate all inputs!
- Test! Test! Test! (unit, integration/end-to-end, fuzzing)
- Hire an additional pair of eyes

Resources/References

- Dancing Offbit: The Story of a Single Character Typo That Broke a ChaCha-based PRNG
- CVE-2020-24613: TLS 1.3 Client Man-in-the-Middle Attack
- Rigging the Vote: Uniqueness in Verifiable Random Functions
- CVE-2021-{43568—43572}: Arbitrary ECDSA signature forgery due to missing zero check