Fun with the hidden number problem

Nadia Heninger

UPenn $\rightarrow$ MSR New England $\rightarrow$ UC San Diego

Joint with Joachim Breitner (DFINITY)

March 23, 2019
Theorem (Law of large numbers)
Average behavior converges almost surely to the expected value as the number of samples increases.

Adage (Law of truly large numbers [Diaconis and Mosteller])
When a sample size is large enough, any outrageous thing is likely to happen.
Conjecture (Cryptographic law of truly large numbers)

Given samples from enough independent cryptographic implementations, any outrageous vulnerability is likely to be present.
ECDSA (Elliptic Curve Digital Signature Algorithm)

Global Parameters Elliptic curve $E$, point $G$ on $E$ of order $n$.

Private Key
Integer $d$

Public Key
Curve point $Q = dG$

Signature Generation
Message Hash: $h$
Per-Signature “nonce”: Integer $k$
Signature on $h$: $(r, s)$  \[ r = x(kG) \quad s = k^{-1}(h + dr) \mod n \]
Potential ECDSA disasters: Public nonce

**Global Parameters** Elliptic curve $E$, point $G$ on $E$ of order $n$.

**Private Key** Integer $d$
**Public Key** Curve point $Q = dG$

**Signature Generation**
**Message Hash:** $h$
**Per-Signature “nonce”:** Integer $k$
**Signature on $h$:** $(r, s)$
\[
 r = x(kG) \quad s = k^{-1}(h + dr) \mod n
\]

**Potential pitfall #1**
Nonce $k$ must remain secret, or else the secret key $d$ is revealed.
\[
 d = (sk - h)r^{-1} \mod n
\]
Potential ECDSA disasters: Repeated nonce

**Global Parameters** Elliptic curve \( E \), point \( G \) on \( E \) of order \( n \).

**Private Key**
Integer \( d \)

**Public Key**
Curve point \( Q = dG \)

**Signature Generation**

Message Hash: \( h \)

Per-Signature “nonce”: Integer \( k \)

Signature on \( h \): \((r, s)\)
\[
 r = x(kG) \\
 s = k^{-1}(h + dr) \mod n
\]

**Potential pitfall #2**

If \( k \) is ever reused to sign distinct messages \( h_1, h_2 \), it is revealed

\[
 k = (h_1 - h_2)(s_1 - s_2)^{-1} \mod n
\]

and thus the long-term private key \( d \) is revealed.
Potential ECDSA disasters: Biased nonce
[Boneh Venkatesan 96], [Howgrave-Graham Smart 2001], [Nguyen Shparlinski 2003]

Potential pitfall #3

$k$ must be generated uniformly at random,
or we can use many signatures to compute the private key $d$.

\[
\begin{align*}
    k_1 - s_1^{-1}r_1d - s_1^{-1}h_1 &\equiv 0 \mod n \\
    k_2 - s_2^{-1}r_2d - s_2^{-1}h_2 &\equiv 0 \mod n \\
    & \vdots \\
    k_m - s_m^{-1}r_md - s_m^{-1}h_m &\equiv 0 \mod n
\end{align*}
\]

\[
\begin{aligned}
    \rightarrow & & \text{lattice attacks} & \rightarrow d
\end{aligned}
\]

If the $k_i$ are small, system of equations likely has unique solution
and lattice techniques can find $d$. 

Formulating ECDSA as a hidden number problem
[Howgrave-Graham Smart 2001], [Nguyen Shparlinski 2003]

We have a system of equations in unknowns $k_1, \ldots, k_m, d$:

$$k_1 - t_1 d - a_1 \equiv 0 \mod n$$
$$k_2 - t_2 d - a_2 \equiv 0 \mod n$$
$$\vdots$$
$$k_m - t_m d - a_m \equiv 0 \mod n$$

We assume the $k_i$ are small.

This is an instance of the hidden number problem [Boneh Venkatesan 96].
Solving the hidden number problem with CVP

Input:

\[ k_1 - t_1 d - a_1 \equiv 0 \mod n \]
\[ \vdots \]
\[ k_m - t_m d - a_m \equiv 0 \mod n \]

in unknowns \( k_1, \ldots, k_m, d \), where \( |k_i| < B \).

Construct the lattice basis

\[
M = \begin{bmatrix}
  n & & & \\
  n & n & & \\
  & & \ddots & \\
  t_1 & t_2 & \cdots & t_m
\end{bmatrix}
\]

Solve CVP with target vector \( v_t = (a_1, a_2, \ldots, a_m) \).

\( v_k = (k_1, k_2, \ldots, k_m) \) will be a close vector in this lattice.
Solving the hidden number problem with CVP embedding

\( k_1 - t_1 d - a_1 \equiv 0 \mod n \)

\( \vdots \)

\( k_m - t_m d - a_m \equiv 0 \mod n \)

in unknowns \( k_1, \ldots, k_m, d \), where \( |k_i| < B \).

LLL, BKZ implementations better than CVP implementations.

Construct the lattice basis

\[ M = \begin{bmatrix}
    n & & & & \\
    & n & & & \\
    & & \ddots & & \\
    & & & n & \\
    t_1 & t_2 & \ldots & t_m & B/n \\
    a_1 & a_2 & \ldots & a_m & B
\end{bmatrix} \]

\( v_k = (k_1, k_2, \ldots, k_m, Bd/n, B) \) is a short vector in this lattice.
Solving the hidden number problem with CVP embedding

Construct the lattice

$$M = \begin{bmatrix} n & n & \cdots & n \\ t_1 & t_2 & \cdots & t_m & B/n \\ a_1 & a_2 & \cdots & a_m & B \end{bmatrix}$$

Want vector

$$v_k = (k_1, k_2, \ldots, k_m, Bd/n, B)$$

We have:

- \( \dim L = m + 2 \) \hspace{1cm} \( \det L = B^2 n^{m-1} \)
- Ignoring approximation factors, LLL or BKZ will find a vector

$$|v| \leq (\det L)^{1/\dim L}$$

- We are searching for a vector with length \( |v_k| \leq \sqrt{m + 2B} \).
- Thus we expect to find \( v_k \) when

$$\log B \leq \left\lfloor \log n(m-1)/m - (\log m)/2 \right\rfloor$$
Solving the hidden number problem with lattices

We expect to find $v_k$ when

$$\log B \leq \lfloor \log n(m - 1)/m - (\log m)/2 \rfloor$$

Boneh and Venkatesan are interested in the limiting behavior:
set $m = \sqrt{\log n}$.

We wish to minimize the number of samples, so for 256-bit $n$:

| #samples | $\log |k|$ |
|----------|--------|
| 2        | 128    |
| 3        | 170    |
| 4        | 190    |
| 20       | 242    |
| 40       | 248    |
Shared unknown prefixes
Subtract samples to reduce to the small unknown case.

\[ k_1 - k_2 \]
Shared unknown suffixes
Subtract samples then shift to reduce to small unknown case.

\[ k_1 - k_2 \]

\[ (k_1 - k_2)2^{-\ell} \]
Solving the hidden number problem with Fourier analysis

[Bleichenbacher 2000], [De Mulder Hutter Marson Pearson 2013]

Input:

\[ k_1 - t_1 d - a_1 \equiv 0 \mod n \]

\[ \vdots \]

\[ k_m - t_m d - a_m \equiv 0 \mod n \]

in unknowns \( k_1, \ldots, k_m, d \), where \( |k_i| < B \).

Bleichenbacher’s Idea #1:

For all pairs \((t_j, a_j)\), let

\[ f(t_j) = e^{\frac{-2\pi i a_j}{n}} \]

Then the Fourier transform is

\[ \hat{f}(\alpha) = \sum_{t_j} f(t_j) e^{\frac{-2\pi i \alpha t_j}{n}} \]

\[ \hat{f}(\alpha) = \sum_{t_j} e^{\frac{-2\pi i (a_j + \alpha t_j)}{n}} = \begin{cases} \sum_{t_j} e^{\frac{-2\pi i k_j}{n}} & \text{if } \alpha = d \\ \sum_{t_j} e^{\frac{-2\pi i r_j}{n}} & r_j \text{ random if } \alpha \neq d \end{cases} \]
Visualizing $f(t_j) = e^{-\frac{2\pi i a_j}{n}}$
Visualizing $\hat{f}(\alpha) = \sum_{t_j} e^{\frac{-2\pi i (a_j + \alpha t_j)}{n}}$
Visualizing $f(t_j) = e^{-\frac{2\pi i a_j}{n}}$ for a few samples $t_j$
Visualizing $\hat{f}(\alpha) = \sum_{t_j} e^{-\frac{2\pi i (a_j + \alpha t_j)}{n}}$ for few samples $t_j$
1. We want to find large coefficients of \( \hat{f}(\alpha) = \sum t_j e^{\frac{-2\pi i (a_j + \alpha t_j)}{n}} \).

But \( n \) is cryptographically large, so we can’t compute the full Fourier transform.

2. Idea: If we restrict the \( t_j < T \) then the “peak” is spread out to have a width of \( n/T \).
Visualizing $f(t_j) = e^{-\frac{2\pi i a_j}{n}}$ for $t_j < T$
Visualizing $\hat{f}(\alpha) = \sum_{t_j} e^{\frac{-2\pi i (a_j + \alpha t_j)}{n}}$ for $t_j < T$
Visualizing $f(t_j) = e^{\frac{-2\pi i a_j}{n}}$ for few samples $t_j < T$
Visualizing $\hat{f}(\alpha) = \sum_{t_j} e^{\frac{-2\pi i (a_j + \alpha t_j)}{n}}$ for few samples $t_j < T$
1. We want to find large coefficients of $\hat{f}(\alpha) = \sum_{t_j} e^{\frac{-2\pi i (a_j + \alpha t_j)}{n}}$.

2. Restrict the $t_j < T$ so the “peak” is spread out to have a width of $n/T$.

3. Now we can compute a Fourier transform over $T$ points spaced evenly at intervals of $n/T$.

Let

$$g(t_j) = e^{\frac{-2\pi i a_j}{n}}$$

$$\hat{g}(\alpha) = \sum_{t_j} e^{\frac{-2\pi i a_j}{n}} e^{\frac{-2\pi i t_j \alpha}{T}}$$

$$\hat{g}(\alpha) \approx e^{\frac{-2\pi i (a_j + \alpha \lfloor n/T \rfloor t_j \alpha)}{T}} \begin{cases} \sum_{t_j} e^{\frac{-2\pi i k_j}{n}} & \text{if } \alpha \lfloor n/T \rfloor \approx d \\ \sum_{t_j} e^{\frac{-2\pi i r_j}{n}} & r_j \text{ random otherwise} \end{cases}$$
Visualizing $g(t_j) = e^{\frac{-2\pi i a_j}{n}}$ for a few samples $t_j$
Visualizing $\hat{g}(\alpha) \approx e^{\frac{-2\pi i (a_j + \alpha \lfloor n/T \rfloor) t_j \alpha}{T}}$ for few samples $t_j$. 
Solving the hidden number problem with Fourier analysis

1. We want to find large coefficients of \( \hat{f}(\alpha) = \sum_{t_j} e^{\frac{-2\pi i (a_j + \alpha t_j)}{n}} \).

2. Restrict the \( t_j < T \) so the “peak” is spread out to have a width of \( n/T \).

3. Compute a Fourier transform over \( T \) points spaced evenly at intervals of \( n/T \).

4. We have learned an approximation \( \tilde{d} \) to \( d \). We wish to shift and zoom the Fourier transform to an interval of width \( M \). Let

\[
    h(\lfloor t_j M/N \rfloor) = e^{\frac{-2\pi ia_j + \tilde{d} t_j}{n}}
\]

\[
    \hat{h}(\alpha) = \sum_{t_j} e^{\frac{-2\pi i a_j}{n}} e^{\frac{-2\pi i t_j \alpha}{T}}
\]

\[
    \hat{g}(\alpha) \approx \begin{cases} 
    \sum_{t_j} e^{\frac{-2\pi ik_j}{n}} & \text{if } \alpha \lfloor n/T \rfloor \approx d \mod M \\
    \sum_{t_j} e^{\frac{-2\pi ir_j}{n}} & r_j \text{ random otherwise}
    \end{cases}
\]
Visualizing $f(t_j) = e^{-\frac{2\pi i a_j}{n}}$ for few samples $t_j < T$
Visualizing $\hat{f}(\alpha) = \sum_{t_j} e^{-2\pi i (a_j + \alpha t_j)/n}$ for few samples $t_j < T$
Solving the hidden number problem with Fourier analysis

One final snag. How do we get samples $t_j < T$?

- Bleichenbacher’s method: sort and difference. Requires many samples. ($\approx 2^{32}$ samples for $n = 2^{256}$)

- Lattice attacks. [De Mulder et al. 2013] Introduces larger noise; not clear if better than just running lattice attack outright.
Where could we possibly find billions of ECDSA keys and signatures, many of them generated by amateur enthusiasts...
Extracting signatures and keys from cryptocurrencies

- Bitcoin, Ethereum, and Ripple all use secp256k1
- Sender signs hash $h$ of transaction.
- An “address” is a hash of a public key.
- Public key revealed by outgoing transactions from an address.
- Transactions recorded on each currency’s blockchain.
- Download client, sync blockchain, extract signatures.*

* This is much more annoying in practice than it sounds.
Cryptanalysis program for ECDSA signatures

1 billion signatures!

Scrape the blockchains

60 million public keys!

Group by public key

Check for duplicate $r$

Lattice attacks on biased nonces

50 CPU years

Now rich?

Yes

Retire

No

 Publish paper
Details of lattice attacks against cryptocurrency signatures

1. Cluster signatures by public key.
2. Select random subsets of 2, 3, 4, and 40 signatures and optimistically run attacks for short, prefix, and suffixed nonces.
3. If vulnerable, we get private key; if not, we get nothing.

**Weird snag:** Signature normalization.

- Signatures \((r, s)\) and \((r, -s)\) both validate.
- Bitcoin makes signatures unique by choosing the smaller of \(s\) and \(-s \mod n\).
- This negates \(k\).
- For prefix, suffix attacks must brute force signs of all nonces.
Repeated nonce results

Bitcoin nonces have been analyzed many times since 2013

\[(n - 1)/2\]

- multisig repeat*
- other repeat

- **Bitcoin**: 2.5M signatures with non-unique \( k \) from 1300 keys.
- **Ethereum**: 185 signatures; 3 keys.
- **Ripple**: 21 signatures; 1 key; 30 XRP.
Biased nonce results

(New; our results.)

- **Bitcoin:** 6000 signatures from 300 keys; 0.008 Bitcoin.
- **Ethereum:** 5 signatures from 1 key; 0.00002 Ether.
- **SSH:** 80 signatures from 4 keys.
Weird trick 1: Small signatures for dust transactions

- 99.9% of the repeated Bitcoin nonce values are
  0xfffffffffffffffffffffffffffffffffffffffffffffffff5d576e7357a4501ddfe92f46681b20a0

- This is \((n - 1)/2\) where \(n\) is the order of \(\text{secp256k1}\).

- The \(x\)-coordinate of \((1/2) \cdot G\) has 166 bits instead of 256.

- Signatures shorter by 11 bytes.

- Greg Maxwell suggested this to clear “dust” transactions.

**A mystery:** Why does \(G\) have this property?
Screwup 2: Human factors

- We traced one compromised key to darkwallet.is.
- Part of a 3-out-of-5 multisig address, used for donations 31oSGBBNrpCiENH3XMZpiP6GTC4tad4bMy.
- Holds 17 BTC ≈ $110k $60k.
- Amir Taaki, one the authors of darkwallet.is, explained:

  It's either me (I was calculating the signatures manually) or my friend who was working on darkwallet (it might have been an earlier version)
Screwup 2: More Human factors

After finding some very small nonces, we brute forced all 32-bit nonces.

- 275 signatures from 52 keys.

Observed nonce values: 1, 2, 9, 100, 1337, 13337, 133337, 1333337, 12345678, and 2147491839

Clearly hand-generated signatures.
Screwup 3: The Bitpay multisig disaster

Bitcore commit "update sign function to use elliptic":
+ return new bignum(SecureRandom.getRandomBuffer(8));
(2014-07-05, released with bitcore v0.1.28)

Bitcore commit "k should be 32 bytes, not 8 bytes":
- return new bignum(SecureRandom.getRandomBuffer(8));
+ return new bignum(SecureRandom.getRandomBuffer(32));
(2014-08-10, released with bitcore v0.1.35)

Nonce type

prefix+64
128+suffix*
128+suffix
160
128*
110
64*
64
32

2014  2015  2016  2017  2018
Screwup 3: The Bitpay multisig disaster

Bitcore commit “update sign function to use elliptic”:
+ return new bignum(SecureRandom.getRandomBuffer(8));
(2014-07-05, released with bitcore v0.1.28)

Nonce type

128+suffix

128+suffix

160

128*

110

64*

64

32

2014 2015 2016 2017 2018
Screwup 3: The Bitpay multisig disaster

**Bitcore commit “update sign function to use elliptic”:**
+ return new bignum(SecureRandom.getRandomBuffer(8));

(2014-07-05, released with bitcore v0.1.28)

**Bitcore commit “k should be 32 bytes, not 8 bytes”:**
- return new bignum(SecureRandom.getRandomBuffer(8));
+ return new bignum(SecureRandom.getRandomBuffer(32));

(2014-08-10, released with bitcore v0.1.35)

HT to Gregory Maxwell for finding this.
Searching for more 64-bit nonces

Precomputation:
1. Precompute $2^{39}$ elements.
2. Precompute $2^{32}$ lookup table of logs of these elements.
   (Took $\approx 4$ days on a few hundred nodes to precompute.)

Online computation:
3. For each of our $2^{30}$ signatures, do $2^{25}$ work to look up.
4. On one machine with 48 cores and 3TB RAM,
   17 days to look up 140,000 signatures.
   (10s per lookup).

Tentative conclusion: 64-bit nonces are not much more common than the ones we found.
Screwup 4: The SHA-256 round constant

60 signatures by SSH servers with a shared 32bit suffix:

<table>
<thead>
<tr>
<th>nonce $k$</th>
<th>secret key $d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>c010..85eaf27871c6</td>
<td>362e..33f9</td>
</tr>
<tr>
<td>f021..cb18f27871c6</td>
<td>362e..33f9</td>
</tr>
<tr>
<td></td>
<td>362e..33f9</td>
</tr>
<tr>
<td>1d39..69cef27871c6</td>
<td>362e..33f9</td>
</tr>
<tr>
<td>0009..e58ef27871c6</td>
<td>362e..33f9</td>
</tr>
<tr>
<td>9e00..4620f27871c6</td>
<td>ca42..3ad7</td>
</tr>
<tr>
<td>a8d8..f92ff27871c6</td>
<td>ca42..3ad7</td>
</tr>
<tr>
<td></td>
<td>ca42..3ad7</td>
</tr>
<tr>
<td>7aad..0f5bf27871c6</td>
<td>ca42..3ad7</td>
</tr>
<tr>
<td>20c1..5dd1f27871c6</td>
<td>ca42..3ad7</td>
</tr>
<tr>
<td>b620..447cf27871c6</td>
<td>713a..f2fa</td>
</tr>
<tr>
<td>3478..0fabf27871c6</td>
<td>713a..f2fa</td>
</tr>
<tr>
<td></td>
<td>713a..f2fa</td>
</tr>
<tr>
<td>4738..0017f27871c6</td>
<td>713a..f2fa</td>
</tr>
<tr>
<td>25b1..6638f27871c6</td>
<td>713a..f2fa</td>
</tr>
</tbody>
</table>
Screwup 4: The SHA-256 round constant
60 signatures by SSH servers with a shared 32bit suffix:

<table>
<thead>
<tr>
<th>nonce $k$</th>
<th>secret key $d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>c010..85eaf27871c6</td>
<td>362e..33f9</td>
</tr>
<tr>
<td>f021..cb18f27871c6</td>
<td>362e..33f9</td>
</tr>
<tr>
<td>:</td>
<td>362e..33f9</td>
</tr>
<tr>
<td>1d39..69cef27871c6</td>
<td>362e..33f9</td>
</tr>
<tr>
<td>0009..e58ef27871c6</td>
<td>362e..33f9</td>
</tr>
<tr>
<td>9e00..4620f27871c6</td>
<td>ca42..3ad7</td>
</tr>
<tr>
<td>a8d8..f92ff27871c6</td>
<td>ca42..3ad7</td>
</tr>
<tr>
<td>:</td>
<td>ca42..3ad7</td>
</tr>
<tr>
<td>7aad..0f5bf27871c6</td>
<td>ca42..3ad7</td>
</tr>
<tr>
<td>20c1..5dd1f27871c6</td>
<td>ca42..3ad7</td>
</tr>
<tr>
<td>b620..447cf27871c6</td>
<td>713a..f2fa</td>
</tr>
<tr>
<td>3478..0fabf27871c6</td>
<td>713a..f2fa</td>
</tr>
<tr>
<td>:</td>
<td>713a..f2fa</td>
</tr>
<tr>
<td>4738..0017f27871c6</td>
<td>713a..f2fa</td>
</tr>
<tr>
<td>25b1..6638f27871c6</td>
<td>713a..f2fa</td>
</tr>
</tbody>
</table>
SHA-224 and SHA-256 use the same sequence of sixty-four constant 32-bit words, K0, K1, ..., K63. These words represent the first 32 bits of the fractional parts of the cube roots of the first sixty-four prime numbers. In hex, these constant words are as follows (from left to right):

428a2f98 71374491 b5c0fbcf e9b5dba5
3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3
72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc
2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7
c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13
650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3
d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0cbc5
391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
74f82ee 78a5636f 84c87814 8cc70208
90befff8 a4506ceb bef9a3f7 c67178f2

25b1..6638f27871c6 713a..f2fa
Screwup 5: Memory-unsafe code

54 signatures with a shared 128bit suffix had a peculiar cause:

<table>
<thead>
<tr>
<th>nonce $k$</th>
<th>secret key $d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>30e7..6b9f0000..0000</td>
<td>0000..00000000..000b</td>
</tr>
<tr>
<td>7d52..688f0000..0000</td>
<td>0000..00000000..000b</td>
</tr>
<tr>
<td>02a9..4fcc0000..0000</td>
<td>0000..00000000..0018</td>
</tr>
<tr>
<td>232c..daba0000..0000</td>
<td>0000..00000000..0018</td>
</tr>
<tr>
<td>1315..80860f80..710b</td>
<td>0f80..710b75f7..ae4b</td>
</tr>
<tr>
<td>3da9..42420f80..710b</td>
<td>0f80..710b75f7..ae4b</td>
</tr>
<tr>
<td>60fe..970c0f80..710b</td>
<td>0f80..710b75f7..ae4b</td>
</tr>
<tr>
<td>32c4..b2ad448e..e255</td>
<td>448e..e25525a3..9d39</td>
</tr>
<tr>
<td>5e22..ef90448e..e255</td>
<td>448e..e25525a3..9d39</td>
</tr>
<tr>
<td>750c..3600448e..e255</td>
<td>448e..e25525a3..9d39</td>
</tr>
<tr>
<td>7917..0cde448e..e255</td>
<td>448e..e25525a3..9d39</td>
</tr>
<tr>
<td>1c9a..ec714c7a..0d8a</td>
<td>4c7a..0d8a35f8..c9ab</td>
</tr>
<tr>
<td>1d5f..7e434c7a..0d8a</td>
<td>4c7a..0d8a35f8..c9ab</td>
</tr>
</tbody>
</table>
## Screwup 5: Memory-unsafe code

54 signatures with a shared 128bit suffix had a peculiar cause:

<table>
<thead>
<tr>
<th>nonce $k$</th>
<th>secret key $d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>30e7..6b9f0000..0000</td>
<td>0000..00000000..000b</td>
</tr>
<tr>
<td>7d52..688f0000..0000</td>
<td>0000..00000000..000b</td>
</tr>
<tr>
<td>02a9..4fcc0000..0000</td>
<td>0000..00000000..0018</td>
</tr>
<tr>
<td>232c..daba0000..0000</td>
<td>0000..00000000..0018</td>
</tr>
<tr>
<td>1315..80860f80..710b</td>
<td>0f80..710b75f7..ae4b</td>
</tr>
<tr>
<td>3da9..42420f80..710b</td>
<td>0f80..710b75f7..ae4b</td>
</tr>
<tr>
<td>60fe..970c0f80..710b</td>
<td>0f80..710b75f7..ae4b</td>
</tr>
<tr>
<td>32c4..b2ad448e..e255</td>
<td>448e..e25525a3..9d39</td>
</tr>
<tr>
<td>5e22..ef90448e..e255</td>
<td>448e..e25525a3..9d39</td>
</tr>
<tr>
<td>750c..3600448e..e255</td>
<td>448e..e25525a3..9d39</td>
</tr>
<tr>
<td>7917..0cde448e..e255</td>
<td>448e..e25525a3..9d39</td>
</tr>
<tr>
<td>1c9a..ec714c7a..0d8a</td>
<td>4c7a..0d8a35f8..c9ab</td>
</tr>
<tr>
<td>1d5f..7e434c7a..0d8a</td>
<td>4c7a..0d8a35f8..c9ab</td>
</tr>
</tbody>
</table>
Possible explanation:

```c
char *create_signature(char *secret_key, char *hash) {
    char k[32];
    char d[16];
    fill_random(k, 32);
    memcpy(d, secret_key, 32);
    ...
    return signature;
}
```
Screwup 5: Memory-unsafe code

Possible explanation:

```c
char *create_signature(char *secret_key, char *hash) {
    char k[32];
    char d[16];
    fill_random(k, 32);
    memcpy(d, secret_key, 32); // facepalm
    ...
    return signature;
}
```
A simple countermeasure

Use deterministic (EC)DSA!

\[ k = H(d \| h) \]

Bitcoin, Ethereum, Ripple official libraries already do.

ed25519 builds in deterministic nonce generation from the start.
On cryptographic assumptions

Explicit assumptions
- Discrete log is hard.
- A hash function behaves like a random oracle.

Implicit assumptions
- The implementation is correct.
- The random number generator is functioning.
- The code implements all required validation checks.
On the design of cryptographic primitives

- Fragility under human error should be a cryptographic design consideration.

- Empirically, developers will make mistakes. How to minimize damage?

- One idea: Tie security to basic functionality.

- There is a tension between diversity (of primitive, implementation) and baseline security (few strongly vetted implementations).
Summary

- We computed a bunch of private keys using lattice attacks.
- We found evidence of various ECDSA implementation flaws.

What else is in the paper?
- Full algorithmic details!
- Tables with numbers!
- More examples of bad implementations!

*Biased Nonce Sense: Lattice Attacks against Weak ECDSA Signatures in Cryptocurrencies.* Joachim Breitner and Nadia Heninger.  
https://eprint.iacr.org/2019/023