Biased Nonce Sense: Lattice Attacks against Weak ECDSA Signatures in Cryptocurrencies

Joachim Breitner[1] and Nadia Heninger[2]

[1] DFINITY Foundation, Zug, joachim@dfinity.org
[2] University of California, San Diego, nadiah@cs.ucsd.edu

Abstract. In this paper, we compute hundreds of Bitcoin private keys and dozens of Ethereum, Ripple, SSH, and HTTPS private keys by carrying out cryptanalytic attacks against digital signatures contained in public blockchains and Internet-wide scans. The ECDSA signature algorithm requires the generation of a per-message secret nonce. This nonce must be generated perfectly uniformly, or else an attacker can exploit the nonce biases to compute the long-term signing key. We use a lattice-based algorithm for solving the hidden number problem to efficiently compute private ECDSA keys that were used with biased signature nonces due to multiple apparent implementation vulnerabilities.

Keywords: Hidden number problem, ECDSA, Lattices, Bitcoin, Crypto

1 Introduction

The security of the ECDSA signature algorithm relies crucially on the proper generation of a per-signature nonce value that is used as an ephemeral private key. It is well known that if an ECDSA private key is ever used to sign two messages with the same signature nonce, the long-term private key is trivial to compute.[18,7,37,13,11,8]

Repeated nonce values are not the only type of bias that can render an ECDSA key insecure, however. In fact, *any* nonuniformity in the ECDSA signature nonces can reveal the private key, given sufficiently many signatures. In this paper, we carry out lattice-based cryptanalytic attacks against ECDSA signatures collected from the Bitcoin, Ethereum, and Ripple blockchains as well as Internet-wide scans of HTTPS and SSH hosts, and efficiently compute hundreds of Bitcoin private keys and a handful of Ethereum and SSH private keys. As a side effect, we also find numerous Bitcoin, Ethereum, Ripple, SSH, and HTTPS private keys that were compromised through repeated signature nonces.

The lattice attacks we apply are based on algorithms for solving the hidden number problem. [6] While the hidden number problem is a popular tool in the cryptanalytic literature for recovering private keys based on side channel attacks [5,14], to our knowledge we are the first to apply these techniques to already-generated keys in the wild, and the first to observe that these techniques may apply to signatures in cryptocurrencies. In total, we computed around 300 Bitcoin keys with these techniques. As of this writing, 818,975 satoshis, or around

$54, and 30.40 XRP, or about $14, remain in Bitcoin and Ripple accounts whose keys we were able to compute, suggesting that these flaws do not yet appear to be known, or else the funds would have already been stolen.

The attacks we use are significantly faster than naive brute force or the state of the art algorithms for the elliptic curve discrete log problem. Using a square root-time algorithm like Pollard rho [28], one could feasibly carry out a targeted attack against a small number of the 64-bit or 128-bit nonces we discovered; carrying out this attack against *all* of the approximately 2^{30} signatures in the Bitcoin blockchain would have required significantly more computational resources than we have access to. In contrast, we spent around 40 CPU-years total on our computations, implemented in Python, for all of the blockchains.

The nonce vulnerabilities fall into several classes that suggest that we have found several independent implementation vulnerabilities. We first use the hidden number problem algorithm to discover the long-term ECDSA signature key when used with nonces that are shorter than expected, and found keys used with nonces with lengths 64 bits, 110 bits, 128 bits, and 160 bits. We extend this technique to discover nonces with shared prefixes and suffixes, and found keys used with signature nonces that shared prefixes and varied in their 64 least significant bits, as well as keys used with signature nonces that shared suffixes and varied in their 128 and 224 most significant bits.

Ethics. We are unable to validate the existence of these vulnerabilities without actually computing the private keys for vulnerable addresses. In the case of cryptocurrencies, these keys give us, or any other attacker, the ability to claim the funds in the associated accounts. In the case of SSH or HTTPS, these keys would give us, or any other attacker, the ability to impersonate the end hosts. We did not do so, and in the course of our research we did not carry out any cryptocurrency transactions or active attacks ourselves; our research is entirely passive, and requires only observation of transactions or general-purpose network measurements. However, given that we find evidence that other attackers are already emptying the accounts of cryptocurrency users whose keys are revealed through known vulnerabilities (both repeated nonces and private keys posted online), we anticipate that users will be affected once knowledge of this flaw becomes public. We have attempted to disclose flaws to the small number of parties we were able to identify, but in most cases we were unable to identify any particular vendors, maintainers, or users to responsibly disclose to.

Countermeasures. All of the attacks we discuss in this paper can be prevented by using deterministic ECDSA nonce generation [29], which is already implemented in the default Bitcoin and Ethereum libraries.

2 Related Work

The Hidden Number Problem and ECDSA. The Hidden Number Problem and the lattice-based algorithm we used to solve it were formulated by Boneh and Venkatesan, who used it to prove the hardness of computing most significant bits for Diffie-Hellman [6]. Howgrave-Graham and Smart [19] and Nguyen and

Shparlinski [26] applied the hidden number problem to show that the DSA and ECDSA signature schemes are insecure if an attacker can learn some most significant bits of the signature nonces. Since then, this technique has been applied in practice in the context of side-channel attacks [5,14].

Repeated DSA/ECDSA signature nonces. A number of works have examined vulnerabilities in DSA and ECDSA due to repeated signature nonces in the wild. Heninger, Durumeric, Wustrow, and Halderman [18] compromised SSH host keys for 1% of SSH hosts in 2012 by searching for repeated DSA signature nonces from SSH handshakes. They traced the problems primarily back to implementation vulnerabilities in random number generation on low-resource devices. Bos, Halderman, Heninger, Moore, Naehrig, and Wustrow [7] documented repeated nonces in the Bitcoin blockchain in 2013, as part of a broader study of elliptic curve cryptography use. Valsorda studied repeated Bitcoin nonces in 2014 [37]. Courtois, Emirdag, and Valsorda [13] studied repeated Bitcoin nonces in 2014 and noted that it would be possible to chain compromises across keys. Castellucci and Valsorda studied repeated nonces and variants of weak keys and nonces repeated across keys in Bitcoin in 2016 [11]. Brengel and Rosow examined repeated nonces within signatures from the same key and chained compromised nonces across signatures from different keys in the Bitcoin blockchain in 2018 [8].

Key generation issues in cryptocurrencies. In 2013, a major bug in Android SecureRandom was blamed for the theft of Bitcoin from many users of Android wallets, due to the faulty random number generators generating repeated ECDSA signature nonces [20,22]. In 2015, the Blockchain.info Android application was discovered to be generating duplicate private keys because the application was seeding from random.org, which had started serving a 403 Redirect to their https URL several months prior [35].

Cryptocurrency cryptanalysis. The Large Bitcoin Collider is a project that is searching for Bitcoin private keys using an apparently linear brute force search algorithm [30] that has searched up to a 54 bit key space. For public keys that are already revealed, it would be more efficient to use square root discrete log algorithms [28,34] to recover short private keys of this type, but we are unaware of any dedicated efforts in this direction.

3 The elliptic curve digital signature algorithm (ECDSA)

The public domain parameters for an elliptic curve digital signature include an elliptic curve E over a finite field and a base point G of order n on E. The private signing key is an integer d modulo n, and the public signature verification key is a point $Q = dG$. Elliptic curve public keys can be represented in uncompressed form by providing both the x and y coordinates of the public point Q, or in compressed form by providing the x coordinate only and a single parity bit from the y value. [9]

To sign a message hash h, the signer chooses a per-message random integer k modulo n, computes the point kG, and then computes the values $(x_r, y_r) = kG \bmod n$, and outputs $r = x_r$ and $s = k^{-1}(h + dr) \bmod n$. The signature is the

pair (r, s). To verify a message hash using a public key Q, the verifier computes $(x'_r, y'_r) = hs^{-1}G + rs^{-1}Q$ and verifies that $x'_r \equiv r \bmod n$. If the bit length ℓ of the curve is shorter than the bit length of the hash function used to compute h, h is truncated to its ℓ most significant bits prior to the calculation. [25]

3.1 ECDSA in cryptocurrencies.

Bitcoin [23], Ethereum [10], and Ripple [33] all use the elliptic curve `secp256k1` [9]. A Bitcoin address is derived from a public key by repeatedly hashing the uncompressed or compressed ECDSA public key with SHA-256 and RIPEMD-160. An Ethereum address is the last 20 bytes of the Keccak-256 hash of the uncompressed ECDSA public key, where Keccak-256 is an early version of the SHA-3 standard. Ethereum public keys are not explicitly included along with the signature; instead, the signature includes an additional byte v that allows the public key to be derived from the signature. A Ripple address is derived from a compressed public key by repeatedly hashing with SHA-256 and RIPEMD-160, and concatenating portions of the hashes. For the purposes of the analysis in our paper, in all of these cryptocurrencies the ECDSA public key is only revealed after an address has been used to sign a transaction. Bitcoin and Ripple explicitly reveal the ECDSA public key in uncompressed or compressed format along with a signature; in Ethereum, clients must derive the public key from the signature itself using key recovery.

ECDSA signatures are used to authenticate the sending party of a transaction. Addresses can be single signature, corresponding to a single public key, or multisignature addresses, which require valid signatures from k out of a set of n public keys in order to spend money from a transaction. Users are typically recommended to use a fresh new address for every transaction [2].

Signature normalization. ECDSA signatures have the property that both the signatures (r, s) and $(r, -s)$ will validate with the same public key. In October 2015, Bitcoin introduced a change in the signing procedure to use the smaller of s and $-s \bmod n$ in a signature in order to make signatures unique.[3] Ethereum and Ripple also do this type of signature normalization, which affects our attack.

3.2 ECDSA in network protocols.

ECDSA signatures can also be used in other network protocols. In TLS, every certificate is signed either by a certificate authority or is self-signed. Most of these signatures remain RSA signatures in practice. However, when ephemeral Diffie-Hellman key exchange is chosen as part of the cipher suite in TLS 1.2 and below, the server signs its portion of the the key exchange, and the client uses the public key in the certificate to validate this signature [16]. In SSH, every host has a host key that it uses to sign the entire handshake between client and server [38]. The client authenticates the server by verifying the signature with the host public key.

[3] https://github.com/bitcoin-core/secp256k1/commit/0c6ab2ff

3.3 Elementary attacks on ECDSA

If an attacker learns the per-message nonce k used to generate an ECDSA signature, the long-term secret key d is easy to compute as $d = (sk - h)r^{-1} \bmod n$.

It is also well known that if the same nonce k is used to sign two different messages h_1 and h_2 with the same secret key, then the secret key is revealed. Let (r_1, s_1) be the signature generated on message hash h_1, and (r_2, s_2) be the signature on message hash h_2. We have immediately that $r_1 = r_2$, since $r_1 = r_2 = x(kG)$. Then we can compute $k = (h_1 - h_2)(s_1 - s_2)^{-1} \bmod n$, and recover the secret key as above.

4 Lattice Attacks on ECDSA

The signature nonce k must also be generated perfectly uniformly at random modulo n, or else techniques for solving the hidden number problem can be used to solve for the secret key d.

4.1 The hidden number problem.

In the hidden number problem as formulated by Boneh and Venkatesan [6], there is a secret integer α modulo a public prime p, and one is given information about the most significant bits of multiples $t_i\alpha \bmod p$, where the t_i are generated at random and known to the attacker. In other words, one is given m pairs of integers $\{(t_i, a_i)\}_{i=1}^{m}$ such that $t_i\alpha - a_i \bmod p = b_i$ with $|b_i| < B$ for some $B < p$.

One can reformulate this problem as seeking a solution $x_1 = b_1$, $x_2 = b_2$, ..., $x_m = b_m$, $y = \alpha$ to the underconstrained system of linear equations

$$\begin{aligned} x_1 - t_1 y + a_1 &\equiv 0 \bmod p \\ &\vdots \\ x_m - t_m y + a_m &\equiv 0 \bmod p \end{aligned} \tag{1}$$

There are two techniques used to solve this problem in the literature. The first uses lattice-based techniques [6,19,26] to solve this system in the case of larger biases and fewer samples (up to around 100 in practice, with B several bits smaller than p), and the second uses Fourier analysis [3,15] and is more suitable with many samples (at least 2^{32}) and very small bias. In this paper, we focus on the former technique, which is better suited to the limited number of signatures we encounter in the wild.

$$M = \begin{bmatrix} p & & & & & \\ & p & & & & \\ & & \ddots & & & \\ & & & p & & \\ t_1 & t_2 & \dots & t_m & B/p & \\ a_1 & a_2 & \dots & a_m & & B \end{bmatrix} \tag{2}$$

To solve the hidden number problem using lattices, consider the lattice generated by the rows of matrix M in Equation (2). The $m \times m$ upper left quadrant is a slightly rescaled version of the lattice given by Boneh and Venkatesan, who suggest using a CVP algorithm to find a vector that

is close to the target, which is the $(m+1)$st row in our lattice basis. The most efficient implementations of lattice algorithms are SVP approximation algorithms, so we follow [5] in embedding this lattice basis into a slightly larger one and using an SVP approximation algorithm instead.

The vector $v_b = (b_1, b_2, \dots, b_m, B\alpha/p, B)$ is a short vector generated by the rows of Equation (2), and by construction $|v_b| < \sqrt{m+2}B$. When $|v_b| \leq \det L^{1/\dim L}$, we hope to recover v_b among the short vectors of a reduced basis for the lattice generated by M. We have $\det M = B^2 p^{m-1}$ and $\dim M = m+2$. The LLL [21] or BKZ [31,32] lattice basis reduction algorithms can be used to find short vectors in this lattice. In practice on random lattices, the LLL algorithm will find a vector satisfying $|v| \leq 1.02^{\dim L} (\det L)^{1/\dim L}$ in polynomial time [27]. The performance of BKZ depends on the block size, and will in time exponential in the block size β find vectors $|v| \leq (1+\epsilon)_\beta^{\dim L} (\det L)^{1/\dim L}$ where ϵ_β depends on the block size, but $\epsilon_\beta = 0.01$ is achievable in practice [12].

In this paper, we focus on relatively small dimension lattices, so that the approximation factor of LLL or BKZ is largely insignificant. In this case, we expect to solve the problem when $\log B \leq \lfloor \log p(m-1)/m - (\log m)/2 \rfloor$.

4.2 Optimizations.

There are two further optimizations that should be applied to this attack. The first is that in the case of most significant bits known, the value b_i is always positive, and thus one can increase the bias by recentering the b_i by writing each equation as $x_i' - t_i y + a_m + B \equiv 0 \bmod p$ which has a solution $x_i' = b_i - B$. The second improvement is to decrease the dimension of the lattice by one by eliminating the variable y from Equations (1) so that one has $m-1$ equations in m unknowns, all bounded.

4.3 Implicit prefixes.

We are also interested in the case where the b_i share an identical prefix, or in other words, that they share most significant bits when viewed as an integer between 0 and p, but we do not know this prefix. That is, the input to the problem is samples $\{t_i, a_i\}_{i=1}^m$ satisfying $b_i + c + a_i \equiv t_i \alpha \bmod p$, with $|b_i| < B$ and $0 \leq c < p$ is unknown. We can reduce this problem to the previous problem with $m-1$ samples by using one of the samples to eliminate the unknown c. That is, we solve the hidden number problem with input $\{t_i' = t_i - t_m]\}_{i=1}^{m-1}$, $a_i' = a_i - a_m$, and the desired solutions $b_i' = b_i - b_m$ satisfy $|b_i'| \leq 2B$.

4.4 Implicit suffixes.

The technique described in Section 4.3 can also be adapted to solve for b_i that share an identical suffix, that is, that they share least significant bits when viewed as an integer between 0 and p. More precisely, the input to our problem in this case is samples $\{t_i, a_i\}_{i=1}^m$ satisfying $2^\ell b_i + c + a_i \equiv t_i \alpha \bmod p$, with $0 \leq b_i < B$,

$0 \le c < p$ unknown, and $2^\ell B \le p$. We can reduce this problem to the case of shared prefixes by multiplying each sample by $2^{-\ell} \bmod n$, so that our rescaled input is samples $\{2^{-\ell} t_i, 2^{-\ell} a_i\}_{i=1}^m$ satisfying $b_i + 2^{-\ell} c + 2^{-\ell} a_i \equiv (2^{-\ell} t_i)\alpha \bmod p$, where $|b_i| < B$ and $2^{-\ell} c$ is still unknown. At this point this is precisely the case of shared prefixes, so we may use the hidden number problem algorithm to solve the case of $m-1$ samples generated as $\{t'_i = 2^{-\ell}(t_i - t_m) \bmod n\}_{i=1}^{m-1}$, $a'_i = 2^{-\ell}(a_i - a_m) \bmod n$, and the desired solutions $b'_i = b_i - b_m$ satisfy $|b'_i| \le 2B$.

4.5 Breaking ECDSA with the hidden number problem

To attack ECDSA with biased k values using the hidden number problem [19,26], note that each signature (r_i, s_i) on h_i satisfies

$$k_i - s_i^{-1} r_i d - s_i^{-1} h_i \equiv 0 \bmod n \tag{3}$$

If the k_i are all small ($|k_i| < B$) or share a common prefix or suffix, then this is precisely our setting for the hidden number problem variants we describe above, with $k_i = b_i$, $\alpha = d$, $p = n$, and s_i, r_i, and h_i public per signature.

We construct the input to our problem by hypothesizing that a set of signatures contains one of the vulnerabilities necessary to carry out the attacks described in Sections 4.1, 4.3, or 4.4, construct the corresponding lattice, and apply a lattice basis reduction algorithm. For each candidate solution for k_i, we compute the value $d_{k_i} = (s_i k_i - h_i) r_i^{-1} \bmod n$, and compare $d_{k_i} G$ to the public key or address.

Experimentally, we found that for a 256-bit n, our case of interest for `secp256k1`, we were able to recover the private key from two signatures with 128-bit nonces by reducing a 3-dimensional lattice with 75% probability, from three signatures with 170-bit nonces with a 4-dimensional lattice with 95% probability, from 4 samples with 190-bit nonces with 100% probability; from 20 samples with 242-bit nonces by reducing a 21-dimensional lattice with 100% probability, and from 40 samples with 248-bit nonces and 41-dimensional lattices.

One can keep continuing by increasing the dimension of the lattice, to a practical limit of a bias of three or four bits for this 256-bit curve order, at the cost of solving near-exact SVP, which runs in time exponential in the lattice dimension, in a high-dimensional lattice.

Unfortunately for the attacker applying these attacks to cryptocurrency signatures, the signature normalization described in Section 3.1 adds complexity. We expect half of the signatures to contain a negated s value, but we will not be able to tell which. From Equation (3), negating s will negate the derived value of k_i. Thus an attack on *small* k_i would still be expected to succeed, since the lattice algorithm can recover both small positive or small negative values, but the normalizations required to solve for the case of shared prefixes in Section 4.3 or shared suffixes in Section 4.4 would produce outputs that do not have the desired properties. For these cases, we brute forced signs for the s_i.

The signature normalization also means that the relations defining ECDSA private key recovery from known or repeated nonces as described in Section 3.3 may not hold as described. For these cases we also brute forced sign values for s.

5 Bitcoin

5.1 Collecting data

To collect Bitcoin signatures we modified the official client to output hash values and signatures as they are verified, and re-validated the entire blockchain.

We used a snapshot of the blockchain from September 13, 2018 (block height 541,244). At this point, the blockchain contained 975,560,082 signatures from 446,605,479 distinct keys. 40,497,752 of these keys had been used to generate more than one signature. 569,396,463, or 58% of the signatures in our snapshot had been generated by one of these keys.

5.2 Cryptanalytic tests for biased nonces

We clustered signatures by public key and eliminated signatures that were fully identical, that is, that shared both the hash h and the signature (r, s). For keys associated with $m > 1$ distinct signatures, we ran the following randomized tests on subsamples of the signatures:

- Check if the set of distinct signatures generated by this key contains any duplicate r values. If so, we compute the private key and all signature nonces k using Section 3.3 and do not run any of the following.
- Select two signatures at random and check for nonces of length less than 128 bits. We repeated this test $2m$ times for each key.
- Select three signatures at random and check for nonces of length less than 170 bits. We repeated this test $2m$ times for each key.
- Select three signatures at random and check for nonces sharing 128 most significant bits, brute forcing signature normalizations. We repeated this test $2m$ times for each key.
- Select three signatures at random and check for nonces sharing 128 least significant bits, brute forcing signature normalizations. We repeated this test $2m$ times for each key.
- For $m \leq 40$, check for nonces of length less than $\lfloor 256(m-1)/m \rfloor - 1$ bits, using all m signatures without signature normalization.
- For $m > 40$, choose a random subset of 41 signatures and check for nonces of length less than 248 bits. We repeated this test $m/20$ times for each key, without signature normalization.

The parameters were chosen so that these tests would complete in a reasonable length of time even for the most common keys.

5.3 Running the cryptanalysis

We implemented these tests in Sage [36], using the built-in BKZ implementation for lattice basis reduction. We ran the computation parallelized across 2000 cores of a heterogeneous cluster with mostly Intel Xeon E5 processors. We ended up running the computation twice, once without signature normalization on a

snapshot of the blockchain from March 2018 and once with normalization in September 2018. For the low-dimensional lattice attacks, the bottleneck of the computation was the elliptic curve multiplications required to check whether we had found the correct private key. The total running time for both jobs was 38 CPU years, and the longest-running job (corresponding to a single key that had generated 1,021,572 signatures in March 2018) completed in 30 calendar days.

5.4 Results and analysis

Biased nonces. After running our attacks, we had computed the private keys for 302 distinct keys that were compromised via small nonces, nonces with shared prefixes, or nonces with shared suffixes. These keys had generated 6,026 signatures with these vulnerable nonces in the blockchain, and 7,328 signatures overall, including signatures that we did not classify as using vulnerable nonces.

For further analysis, we used the BlockSci library [4]. We classified keys by the signature nonce vulnerability that had compromised them and summarize the data in Table 1. Nearly all of the compromised keys had been used as part of multisignature addresses of type 1-out-of-1, 1-out-of-2, 1-out-of-3, 2-out-of-2, 2-out-of-3, 2-out-of-5, or 3-out-of-5.

On September 23, 2018, a total of 745,990 satoshis were in non-multisignature addresses whose keys were compromised by these biased nonces. An additional 72,985 satoshis were in a multisignature address where we possessed all of the necessary keys for the account. A further 6,480,000 satoshis were present in addresses for which we possessed one out of two necessary signatures.

We plot the signatures from biased nonces over time in Figure 1. Nearly all of the compromised nonces fell into a few clear classes based on the length of the variable portion of the nonce. We found short nonces of length 160 bits, 128 bits, 110 bits, 64 bits, and a few sporadic nonces below 32 bits. We also found nonces that shared a fixed prefix followed by a variable 64-bit suffix, and nonces that varied in the 128 most significant bits and shared a fixed 128-bit suffix. Most of the affected keys were part of multisignature addresses.

Table 1. Biased signatures and keys. We classified the compromised keys and signatures by the type of nonce vulnerability that had compromised the private key. Nearly all of the compromised keys had been used as part of multisignature addresses.

Nonce Type	Signatures	Distinct Keys	Multisignature Keys
Prefix + 64 bits	27	2	0
128 bits + Suffix	121	13	4
160 bits	3	1	0
128 bits	4	2	2
110 bits	2	1	0
64 bits	5,863	280	279
$\leq$ 32 bits	6	3	0

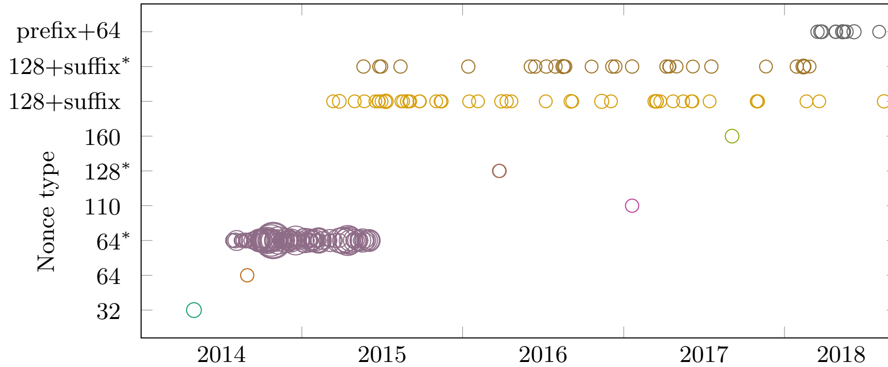

Fig. 1. Bitcoin signatures with small and biased nonces over time. We plot signatures with biased nonces over time, grouped by the class of bias we observed, and whether they are used with multisignature addresses (marked with *). Larger circles correspond to more signatures on a given date. We note that the different types of biases appear at different date ranges, suggesting that these vulnerabilities are specific to distinct implementations.

64-bit nonces. We found 5,863 signatures from 280 distinct keys that used 64-bit nonces. All but one of these keys was used as part of multisignature addresses. All of these signatures appeared between July 26, 2014 and June 1, 2015.

Two accounts related to these keys have a non-zero balance: One 2-out-of-2 address, for which we have one private key, has a single satoshi. One 2-out-of-3 address, for which we have two private keys, has a balance of 72,985 satoshis.

Since nearly all of these keys are part of multisignature addresses, we hypothesize that this may be a faulty implementation intended for multifactor security, such as a hardware token.

64-bit nonces and single-signature keys. Our lattice attack only applies when at least *two* signatures with a small nonce are created using the same secret key. A single 64-bit nonce requires only 2^{32} time to break using Shanks's baby-step-giant-step algorithm [34] or the Pollard rho algorithm [28], feasible with only modest computation resources. Applying this attack to *all* of the 2^{30} Bitcoin signatures was beyond our resources, but we investigated a random sample.

The bottleneck is the random accesses into the precomputed lookup tables, so we chose the parameters so that they fit into the RAM of our largest-memory machine (2.2TB). A single core can check a single signature in ≈ 7 minutes.[4]

We spent 15 calendar days of computation time, or 17,000 core-hours, to check a random sample of 144,000 Bitcoin signatures and found one nonce that had already been computed via the lattice attacks and no previously unknown small nonces. We conclude that the lattice attack appears to have found most of the vulnerable signatures.

[4] The code can be found at:https://github.com/nomeata/secp265k1-lookup-table.

110-, 128-, and 160-bit nonces. We found a few sporadic signatures that used larger nonce lengths that were broken by our lattice techniques. These may be individual programming errors, but do not appear to be part of common implementations. None of the affected accounts had a non-zero balance.

- Three 160-bit nonces, all of which were used with the same key, all on the same date in September 2017. This key did not produce any more signatures in our data. We hypothesized that a 160-bit nonce length might be explained by a user generating a nonce using a hash function with 160-bit output, as in deterministic ECDSA, but were unable to verify this.
- Four 128-bit nonces from two keys. Each key generated two signatures with 128-bit nonces on the same two days in March 2016, and no further signatures.
- One signed 110-bit nonce, used with one key in January 2017, which had also generated a normal-looking 256-bit nonce on the same day.

256-bit nonces with shared 128-bit suffixes. 121 signatures were compromised by nonces that shared a 128-bit suffix with at least one other signature. 55 of these signatures were used with multisignature addresses and 66 were generated by non-multisignature addresses. 13 keys were compromised this way, which had generated a total of 224 signatures. There were 20 distinct suffixes that had been used by these keys. The earliest signature of this type that we found was from March 2015, and the most recent was from August 2018. Some of the keys were used with nonces that all shared the same suffix, and some were used with nonces of varying and occasionally unique suffixes.

We found that a number of the addresses associated with these compromised signatures had been posted on the web along with their private keys for a variety of reasons: they corresponded to small integer private keys, private keys derived from easy-to-guess passwords such as "satoshi", or private keys used as examples in documentation. All of the affected accounts had a zero balance.

Interestingly, in 54 of the signatures, the 128-bit nonce suffix is identical to the 128 most significant bits of the private key. The vulnerable transactions emptied the relevant accounts. We hypothesize that the vulnerable nonce suffixes we observe may actually be due to a custom implementation used by an *attacker* who is emptying accounts from Bitcoin addresses that were already compromised online. The overlapping bits between the nonce and the private key might be an artifact from a bug in a program written in a memory-unsafe language like C.

256-bit nonces with shared 192-bit prefixes. We computed 2 keys that had been used with 27 signatures with nonces sharing prefixes. Each key had some signatures with the shared prefix and some without. One of the two keys has a balance of 495,990 satoshis, and seems to be in current use at the time of writing.

Repeated nonces As a side effect of our analysis, we also calculated 1,296 private keys from repeated signature nonces. These keys had generated 4,295,141 signatures. Nearly all of the repeated nonces, in 2,456,870 signatures, used `0x7fffffffffffffffffffffffffffffff5d576e7357a4501ddfe92f46681b20a0`.

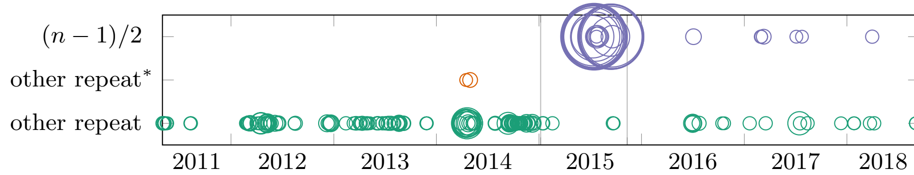

Fig. 2. Bitcoin signatures with repeated nonces over time. We plot repeated signature nonces over time, separating the value $(n-1)/2$, which seems to have been used intentionally, from other repeated nonces. Larger circles correspond to more signatures on a given date. Signatures involving multisignature addresses are marked with *. The vertical bars mark when the development (left) and release (right) versions of the official bitcoin client began to create nonces deterministically [29].

As noted by [8], this is $(n-1)/2$ where n is the order of the `secp256k1` curve. The x-coordinate of $G \cdot (n-1)/2$ is 166 bits long, where one would expect a random point to have 256 bits. It appears to not be known why `secp256k1` has this property; Gregory Maxwell [1] notes that `secp224k1` shares the same 166-bit string doubled to produce the generator, and speculates that this value is the output of SHA-1. According to [1], this value is used to sweep "dust" transactions.

We note that even for a "final" transaction for a given key, an attacker could observe a proposed transaction, derive the secret key, and race the original transaction. This is not a concern if the key has already been compromised, however. Some of the transactions using $k = (n-1)/2$ are withdrawing money from addresses derived from easily guessable brainwallet passwords.

There were no funds left in any of the addresses with repeated nonces at the time that we examined the results. Since this failure mode of ECDSA is well known, it appears that multiple entities regularly scan the blockchain for repeated nonces and remove any funds from vulnerable keys. However, there were two multisignature addresses with nonzero account balances for which repeated nonces revealed one of the necessary addresses:

– The website https://www.darkwallet.is/ asks for donations to be sent to a 3-out-of-5 multisignature address, which currently holds a balance of 1,722,498,619 satoshis (approx. 110 kUSD). One of the five keys was compromised by a repeated signature nonce. We contacted Amir Taaki, one of the founders, who

Table 2. Repeated signature nonces. Nearly all of the repeated signature nonces on the Bitcoin blockchain have the value $(n-1)/2$. These represented the majority of keys compromised through repeated nonces.

Nonce	Total Signatures	Repeated Nonce Signatures	Distinct Keys
$(n-1)/2$	4,275,639	2,456,870	918
Others	19,052	2,214	378

told us that signatures from these addresses had been calculated manually, suggesting that the random number generator may not have been seeded.
- One 2-out-of-3 address had a balance of 179,400 satoshis.

In Figure 2, we plot non-unique signature nonces over time. Clusters of repeated nonces in 2013 appear to correspond to one of the reported RNG vulnerabilities discussed in Section 2. The rate of repeats decreases after 2014.

Other small nonces. We brute forced all 32-bit nonce values, and found 275 signatures from 52 keys. The small number and the observed nonces (1, 2, 9, 100, 1337, 13337, 133337, 1333337, 12345678, and 2147491839) do not point to a flawed implementation, but rather hand-crafted transactions and signatures.

6 Ethereum

Collecting data. We collected Ethereum signatures by querying a local Ethereum node via its RPC interface. We ran our analysis on a snapshot of the blockchain from September 17, 2018 (block 6,346,730). It contained 311,118,952 signatures from 34,754,686 distinct public keys. 19,558,608 (57%) keys had generated more than one signature, resulting in 295,922,874 (95%) signatures from such keys.

Running the cryptanalysis. We clustered signatures by public key, and examined the keys that had generated more than one signature. We ran the same tests as for Bitcoin, and as with Bitcoin, we ran the computation twice, once with signature normalization on our September 2018 blockchain snapshot and once without on a snapshot from July 2018. The total computation took 9.5 CPU years, and the longest-running job (corresponding to a single key that had generated 1,321,734 signatures in July 2018) completed in 25 calendar days.

Results and analysis.

256-bit nonce with 192-bit prefix. One key was compromised via biased nonces. It had generated seven signatures in our dataset, of which five nonces shared the same nonzero, random-looking 192-bit prefix and differed only in the last 64 bits of the nonce. The remaining two signatures that had been generated by this key look random, and do not share this prefix. The key holds 0.00002 Ether.

Repeated nonces. Three keys were compromised from repeated nonces, with 185 signatures between them. The repeated nonces include four occurrences of the nonce 1, two occurrences of a seemingly random 256-bit nonce, and `123456789abcdef`. No funds are held by these keys.

7 Ripple

Collecting data. We downloaded a portion of the Ripple blockchain that included 218,101,343 signatures. There were 571,482 unique public keys, of which 379,575 had generated more than one signature, totaling 217,909,436 signatures (99%) that were generated by a key that had been used more than once.

Running the cryptanalysis. We clustered signatures by public key, and examined the keys that had generated more than one signature. We ran the same tests as for Bitcoin. The total computation time took 1.1 CPU years, and the longest single computation took 5 calendar days for a single key that had generated 361,366 signatures.

Results and analysis. We found one private key that had been compromised by a repeated signature nonce. This key had generated 21 signatures. It holds 30.40 XRP (approx. 14 USD) and 1.81 CNY. We deduce that attackers have not yet begun to systematically observe the Ripple blockchain for repeated nonces.

8 SSH

Collecting data. We gathered DSA and ECDSA signatures from Internet-wide scans of SSH on port 22 that were performed by Censys [17] between April 3, 2018 and September 18, 2018. The scans contained between 7.9 million and 9.4 million DSA and ECDSA signatures each, for a total of 196,884,009 signatures from 20,103,764 distinct public keys. These included 191,855,472 NIST P-256 signatures, 2,634,869 DSA signatures, 2,095,181 NIST P-521 signatures, 164,919 NIST P-384 signatures, and 133,568 ed25519 signatures.

Running the cryptanalysis. The SSH dataset included a wide variety of different DSA groups. We ran the tests described in Section 5.2, scaled to the relevant group size. The total computation time was 2.8 CPU-years, and the longest computation took 8 days to process the most common key, which had been used for 1,450,916 signatures from our scans.

Results and analysis.

256-bit keys with 32-bit shared suffixes. Three private keys produced signatures whose nonces all shared the suffix `f27871c6`. The hosts have gone offline since, and we were unable to identify the implementation. This suffix is one of the "constant words" used in the calculation of a SHA-2 hash [24], with swapped byte order. We can can speculate that the server is using SHA-2 to generate the nonce, but has a bug in the implementation.

224-bit keys with 160-bit nonces. One further key was compromised due to the use of small nonces. All 23 signatures by this key used a 160 bit nonce with a 2048-bit DSA public key with a 224-bit subgroup, and were observed at the same IP address. We speculate that this may be due to the use of a 160-bit hash function like SHA-1 or MD5 being used to generate the nonce in a 224-bit group.

Repeated nonces. 681 signatures were compromised by repeated nonces. Of these, 612 used DSA, and 69 used ECDSA with NIST P-256. These came from 34 distinct public keys on 80 distinct IP addresses.

We compared this number to repeated nonces found in a March 25, 2012 scan of SSH that requested only DSA host keys provided by the authors of [18]. In the 2012 scan, 22,182 nonces had been used more than once, and 58 distinct keys were vulnerable on 24,893 distinct IP addresses. We conclude that many of these vulnerable implementations have been taken offline or patched since then.

9 HTTPS

Collecting data. We gathered ECDSA signatures from weekly Internet-wide scans of HTTPS on port 443 performed by Censys [17] between April 3, 2018 and September 6, 2018. The number of ECDSA signatures per scan increased from 1.5 million to 1.9 million, resulting in 50,313,795 total ECDSA signatures from 182,843 distinct keys on 3,333,482 distinct IP address. 50,096,848 signatures were from NIST P-256, 212,523 were from NIST P-384, 4,400 were from NIST P-521, and 24 were from NIST P-224.

Running the cryptanalysis. We ran the same sequence of tests as described in Section 5.2, scaling the number of bits to the curve order. The total computation time was 152 CPU-days, and the longest computation took 17 days to process a single key that had produced 4,093,917 signatures from our scan.

Results and analysis. We did not find any small or biased signature nonces. We found three different sources of signatures with repeated nonces, which we hypothesize are due to flawed random number generators. These resulted in 462 vulnerable signatures that had been generated by 7 distinct private keys on 97 distinct IP addresses.

10 Acknowledgements

We thank Luke Valenta and Zakir Durumeric for help in updating ZGrab and Censys to collect HTTPS and SSH signature hashes, and Tanja Lange for the reference on the surprisingly small binary representation of $k = 1/2$ in `secp256k1`. Much of the work for this paper was done while the authors were at the University of Pennsylvania. This work was supported by the National Science Foundation under grants no. CNS-1651344 and CNS-1513671. We are grateful to Cisco for donating much of the computing cluster used to carry out our computations.

References

1. The most repeated r value on the blockchain. https://bitcointalk.org/index.php?topic=1118704.0 (2015)
2. Bitcoin wiki: Address reuse. https://en.bitcoin.it/wiki/Address_reuse (2018)
3. Akavia, A.: Solving hidden number problem with one bit oracle and advice. In: Halevi, S. (ed.) Advances in Cryptology - CRYPTO 2009. pp. 337–354. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
4. Bartoletti, M., Lande, S., Pompianu, L., Bracciali, A.: A general framework for blockchain analytics. In: Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers. pp. 7:1–7:6. SERIAL '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3152824.3152831, http://doi.acm.org/10.1145/3152824.3152831
5. Benger, N., van de Pol, J., Smart, N.P., Yarom, Y.: "Ooh aah... just a little bit": A small amount of side channel can go a long way. In: Batina, L., Robshaw, M. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2014. pp. 75–92. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
6. Boneh, D., Venkatesan, R.: Hardness of computing the most significant bits of secret keys in diffie-hellman and related schemes. In: Koblitz, N. (ed.) Advances in Cryptology — CRYPTO '96. pp. 129–142. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
7. Bos, J.W., Halderman, J.A., Heninger, N., Moore, J., Naehrig, M., Wustrow, E.: Elliptic curve cryptography in practice. In: Christin, N., Safavi-Naini, R. (eds.) Financial Cryptography and Data Security. pp. 157–175. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
8. Brengel, M., Rossow, C.: Identifying key leakage of bitcoin users. In: Bailey, M., Holz, T., Stamatogiannakis, M., Ioannidis, S. (eds.) Research in Attacks, Intrusions, and Defenses. pp. 623–643. Springer International Publishing, Cham (2018)
9. Brown, D.R.L.: SEC 2: Recommended elliptic curve domain parameters. http://www.secg.org/sec2-v2.pdf (2010)
10. Buterin, V.: Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper (2013)
11. Castellucci, R., Valsorda, F.: Stealing bitcoin with math (2016), https://news.webamooz.com/wp-content/uploads/bot/offsecmag/151.pdf
12. Chen, Y., Nguyen, P.Q.: BKZ 2.0: Better lattice security estimates. In: ASIACRYPT. Lecture Notes in Computer Science, vol. 7073, pp. 1–20. Springer (2011)
13. Courtois, N.T., Emirdag, P., Valsorda, F.: Private key recovery combination attacks: On extreme fragility of popular bitcoin key management, wallet and cold storage solutions in presence of poor rng events. Cryptology ePrint Archive, Report 2014/848 (2014), https://eprint.iacr.org/2014/848
14. Dall, F., De Micheli, G., Eisenbarth, T., Genkin, D., Heninger, N., Moghimi, A., Yarom, Y.: Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(2), 171–191 (May 2018). https://doi.org/10.13154/tches.v2018.i2.171-191, https://tches.iacr.org/index.php/TCHES/article/view/879
15. De Mulder, E., Hutter, M., Marson, M.E., Pearson, P.: Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA. In: Bertoni, G., Coron, J.S. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2013. pp. 435–452. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

16. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) protocol. IETF RFC RFC5246 (2008)
17. Durumeric, Z., Adrian, D., Mirian, A., Bailey, M., Halderman, J.A.: A search engine backed by Internet-wide scanning. In: 22nd ACM Conference on Computer and Communications Security (Oct 2015)
18. Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your Ps and Qs: Detection of widespread weak keys in network devices. In: Proceedings of the 21st USENIX Security Symposium (Aug 2012)
19. Howgrave-Graham, N.A., Smart, N.P.: Lattice attacks on digital signature schemes. Designs, Codes and Cryptography **23**(3), 283–290 (Aug 2001). https://doi.org/10.1023/A:1011214926272, https://doi.org/10.1023/A:1011214926272
20. Klyubin, A.: Some SecureRandom thoughts. https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html (August 2013)
21. Lenstra, A.K., Lenstra, H.W., Lovasz, L.: Factoring polynomials with rational coefficients. MATH. ANN **261**, 515–534 (1982)
22. Michaelis, K., Meyer, C., Schwenk, J.: Randomly Failed! The State of Randomness in Current Java Implementations. In: CT-RSA. vol. 7779, pp. 129–144. Springer (2013)
23. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf (2009)
24. National Institute of Standards and Technology: FIPS PUB 180-2: Secure Hash Standard (Aug 2002)
25. National Institute of Standards and Technology: FIPS PUB 186-4: Digital Signature Standard (DSS) (Jul 2013)
26. Nguyen, P.Q., Shparlinski, I.E.: The insecurity of the elliptic curve digital signature algorithm with partially known nonces. Designs, Codes and Cryptography **30**(2), 201–217 (Sep 2003). https://doi.org/10.1023/A:1025436905711, https://doi.org/10.1023/A:1025436905711
27. Nguyen, P.Q., Stehlé, D.: LLL on the average. In: Hess, F., Pauli, S., Pohst, M. (eds.) Algorithmic Number Theory. pp. 238–256. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
28. Pollard, J.M.: Monte Carlo methods for index computation (mod p). In: Mathematics of Computation. vol. 32 (1978)
29. Pornin, T.: Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). https://tools.ietf.org/html/rfc6979 (2013)
30. rico666: Large bitcoin collider. https://lbc.cryptoguru.org/
31. Schnorr, C.P.: A hierarchy of polynomial time lattice basis reduction algorithms. Theor. Comput. Sci. **53**(2-3), 201–224 (Aug 1987). https://doi.org/10.1016/0304-3975(87)90064-8, http://dx.doi.org/10.1016/0304-3975(87)90064-8
32. Schnorr, C.P., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. Math. Program. **66**(2), 181–199 (Sep 1994). https://doi.org/10.1007/BF01581144, http://dx.doi.org/10.1007/BF01581144
33. Schwartz, D., Youngs, N., Britto, A.: The Ripple protocol consensus algorithm. https://ripple.com/files/ripple_consensus_whitepaper.pdf (2014), https://ripple.com/files/ripple_consensus_whitepaper.pdf, accessed: 2016-08-08
34. Shanks, D.: Class number, a theory of factorization, and genera. In: Proc. of Symp. Math. Soc., 1971. vol. 20, pp. 41–440 (1971)
35. Team, B.: Android wallet security update. https://blog.blockchain.com/2015/05/28/android-wallet-security-update/
36. The Sage Developers: SageMath, the Sage Mathematics Software System (Version 8.1) (2017), http://www.sagemath.org

37. Valsorda, F.: Exploiting ECDSA failures in the bitcoin blockchain. Hack In The Box (HITB) (2014)
38. Ylonen, T., Lonvick, C.: The Secure Shell (SSH) transport layer protocol. IETF RFC 4253 (2006)