Elliptic Curve Cryptography in Practice

Joppe W. Bos[1], J. Alex Halderman[2], Nadia Heninger[3], Jonathan Moore, Michael Naehrig[1], and Eric Wustrow[2]

[1] Microsoft Research
[2] University of Michigan
[3] University of Pennsylvania

Abstract. In this paper, we perform a review of elliptic curve cryptography (ECC), as it is used in practice today, in order to reveal unique mistakes and vulnerabilities that arise in implementations of ECC. We study four popular protocols that make use of this type of public-key cryptography: Bitcoin, secure shell (SSH), transport layer security (TLS), and the Austrian e-ID card. We are pleased to observe that about 1 in 10 systems support ECC across the TLS and SSH protocols. However, we find that despite the high stakes of money, access and resources protected by ECC, implementations suffer from vulnerabilities similar to those that plague previous cryptographic systems.

1 Introduction

Elliptic curve cryptography (ECC) [34, 39] is increasingly used in practice to instantiate public-key cryptography protocols, for example implementing digital signatures and key agreement. More than 25 years after their introduction to cryptography, the practical benefits of using elliptic curves are well-understood: they offer smaller key sizes [36] and more efficient implementations [6] at the same security level as other widely deployed schemes such as RSA [45]. In this paper, we provide two contributions:

- First, we study the current state of existing elliptic curve deployments in several different applications. Certicom released the first document providing standards for elliptic curve cryptography in 2000, and NIST standardized ECDSA in 2006. What does the deployment of these algorithms look like in 2013? In order to study this question, we collect cryptographic data from a number of different real-world deployments of elliptic curve cryptography: Bitcoin [40], secure shell (SSH) [48], transport layer security (TLS) [9], and the Austrian Citizen Card [31].
- Next, we perform a number of "sanity checks" on the data we collected, in particular on the public keys, key exchange data, and digital signatures, in order to detect implementation problems that might signal the presence of cryptographic vulnerabilities.

The security of deployed asymmetric cryptographic schemes relies on the believed hardness of number theoretic problems such as integer factorization and the computation of discrete logarithms in finite fields or in groups of points on an elliptic curve. However, most real-world cryptographic vulnerabilities do not stem from a weakness in the underlying hardness assumption, but rather from implementation issues such as side-channel attacks, software bugs or design flaws (cf. [27]). One such example are so-called cache attacks [42] (see [13] for an application to the asymmetric setting) that exploit the memory access pattern in cryptographic schemes using data dependent table lookups. Another class of problems is related to implementations which do not provide sufficient randomness and subsequently generate insecure cryptographic keys. Recent examples of implementations suffering from a

lack of randomness are the Debian OpenSSL vulnerability [52], the discovery of widespread weak RSA and DSA keys used for TLS, SSH, and PGP as documented in [35, 30] and recent results in [4] that show how to break a number of RSA keys obtained from Taiwan's national Citizen Digital Certificate database.

In order to survey the implementation landscape for elliptic curve cryptography, we collected several large cryptographic datasets:

- The first (and largest) dataset is obtained from the Bitcoin block chain. Bitcoin is an electronic crypto-currency, and elliptic curve cryptography is central to its operation: Bitcoin addresses are directly derived from elliptic-curve public keys, and transactions are authenticated using digital signatures. The public keys and signatures are published as part of the publicly available and auditable block chain to prevent double-spending.
- The second largest dataset we collected is drawn from an Internet-wide scan of HTTPS servers. Elliptic-curve cipher suites that offer forward secrecy by establishing a session key using elliptic-curve Diffie-Hellman key exchange [20] were introduced in 2006 and are growing in popularity for TLS. This dataset includes the Diffie-Hellman server key exchange messages, as well as public keys and signatures from servers using ECDSA.
- We also performed an Internet-wide scan of SSH servers. Elliptic-curve cipher suites for SSH were introduced in 2009, and are also growing more common as software support increases. This dataset includes elliptic curve Diffie-Hellman server key exchange messages, elliptic-curve public host keys, and ECDSA signatures.
- Finally, we collected certificate information, including public keys from the publicly available lightweight directory access protocol (LDAP) database for the Austrian Citizen Card. The Austrian e-ID contains public keys for encryption and digital signatures, and as of 2009, ECDSA signatures are offered.

Our main results can be categorized as follows.

Deployment. Elliptic curve cryptography is far from being supported as a standard option in most cryptographic deployments. Despite three NIST curves having been standardized, at the 128-bit security level or higher, the smallest curve size, secp256r1, is by far the most commonly used. Many servers seem to prefer the curves defined over smaller fields.

Weak keys. We observed significant numbers of non-related users sharing public (and hence private) keys in the wild in both TLS and SSH. Some of these cases were due to virtual machine deployments that apparently duplicated keys across distinct instances; others we were able to attribute to default or low-entropy keys generated by embedded devices, such as a network firewall product.

Vulnerable signatures. ECDSA, like DSA, has the property that poor randomness used during signature generation can compromise the long-term signing key. We found several cases of poor signature randomness used in Bitcoin, which can allow (and has allowed) attackers to steal money from these clients. There appear to be diverse causes for the poor randomness, including test values for uncommonly used implementations, and most prominently an Android Java bug that was discovered earlier this year (see [37] for a discussion of this bug in Android and related Java implementations).

2 Preliminaries

This section briefly discusses the standardized elliptic curves that are mainly used in practice. It also fixes notation for elliptic curve public-key pairs and introduces the basic concepts for key establishment and digital signatures in the elliptic curve setting.

2.1 Elliptic Curves Used in Practice

First, we briefly recap standardized elliptic curves that are used most commonly in real-world applications. All these curves are given in their short Weierstrass form $E : y^2 = x^3 + ax + b$ and are defined over a finite field $\mathbf{F}_p$, where $p > 3$ is prime and $a, b \in \mathbf{F}_p$. Given such a curve E, the cryptographic group that is employed in protocols is a large prime-order subgroup of the group $E(\mathbf{F}_p)$ of $\mathbf{F}_p$-rational points on E. The group of rational points consists of all solutions $(x, y) \in \mathbf{F}_p^2$ to the curve equation together with a point at infinity, the neutral element. The number of $\mathbf{F}_p$-rational points is denoted by $\#E(\mathbf{F}_p)$ and the prime order of the subgroup by n. A fixed generator of the cyclic subgroup is usually called the base point and denoted by $G \in E(\mathbf{F}_p)$.

In the FIPS 186-4 standard [51], NIST recommends five elliptic curves for use in the elliptic curve digital signature algorithm targeting five different security levels. Each curve is defined over a prime field defined by a generalized Mersenne prime. Such primes allow fast reduction based on the work by Solinas [47]. All curves have the same coefficient $a = -3$, supposedly chosen for efficiency reasons, and their group orders are all prime, meaning that $n = \#E(\mathbf{F}_p)$. The five recommended primes are

$$\begin{aligned} p_{192} &= 2^{192} - 2^{64} - 1, & p_{224} &= 2^{224} - 2^{96} + 1, \\ p_{256} &= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1, & p_{384} &= 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1, \\ p_{521} &= 2^{521} - 1. \end{aligned}$$

In the standard, these curves are named P-192. P-224, P-256, P-384, and P-521, but in practice they also appear as `nistp192`, `nistp224` etc. These along with other curves are also recommended by Certicom in the standards for efficient cryptography SEC2 [16], in which the curves are named `secp192r1`, `secp224r1`, `secp256r1`, `secp384r1`, `secp521r1`. But sometimes, other names are used, for example P-192 and P-256 are named `prime192v1` and `prime256v1` in OpenSSL.

For 256-bit primes, in addition to the NIST curve defined over $\mathbf{F}_{p_{256}}$, SEC2 also proposes a curve named `secp256k1` defined over $\mathbf{F}_p$ where $p = 2^{256} - 2^{32} - 977$. This curve is used in Bitcoin. It has a 256-bit prime order. Interestingly, this choice deviates from those made in FIPS 186-4 in that the curve coefficients are $a = 0$ and $b = 7$. This means that `secp256k1` has j-invariant 0 and thus possesses a very special structure.

A curve with j-invariant 0 has efficiently computable endomorphisms that can be used to speed up implementations, for example using the GLV decomposition for scalar multiplication [26]. Since for `secp256k1` $p \equiv 1 \pmod 6$, there exists a primitive 6th root of unity $\zeta \in \mathbf{F}_p$ and a corresponding curve automorphism $\psi : E \to E,\ (x, y) \mapsto (\zeta x, -y)$. This map allows the fast computation of certain multiples of any point $P \in E(\mathbf{F}_p)$, namely $\psi(P) = \lambda P$ for an integer λ with $\lambda^6 \equiv 1 \pmod n$. But efficient endomorphisms not only speed up scalar multiplication, they also speed up Pollard's rho algorithm [43] for computing discrete logarithms [24]. The automorphism group of E has order 6 and is generated by the map ψ above.

In contrast, an elliptic curve with j-invariant different from 0 and 1728 only has an automorphism group of order 2, such that the speed-up in Pollard's rho algorithm is a constant factor of up to $\sqrt{3}$ over such a curve.

Another consequence of the larger automorphism group is the existence of six twists (including the curve itself and the standard quadratic twist). An implementation using x-coordinate only arithmetic (such as the formulas in [11]) must pay attention to the curve's twist security (see [2, 3]). This means that its quadratic twist needs to have a large enough prime divisor for the discrete logarithm problem on the twist to be hard enough. This prevents an invalid-curve attack in which an attacker obtains multiples with secret scalars of a point on the quadratic twist, e.g. via fault injection [25]. The quadratic twist of `secp256k1` has a 220-bit prime factor and thus can be considered twist secure (e.g. as in [5]). A non-laddering implementation (using both x- and y-coordinates) can be compromised by an invalid-curve attack if the implementation does not check whether the point satisfies the correct curve equation [7]. This could lead to a more serious attack on `secp256k1`[4] since an attacker might obtain scalar multiples with secret scalars of a point on any curve over $\mathbf{F}_p$ with coefficient $a = 0$, i.e. on any of `secp256k1`'s twists. The largest prime divisors of the remaining four twists' group orders are of size 133, 188, 135, and 161 bits, respectively, but there are several other smaller prime factors that offer more choices for an invalid-curve attack.

2.2 Elliptic Curve Public-Key Pairs

Given a set of domain parameters that include a choice of base field prime p, an elliptic curve $E/\mathbf{F}_p$, and a base point G of order n on E, an elliptic curve key pair (d, Q) consists of a private key d, which is a randomly selected non-zero integer modulo the group order n, and a public key $Q = dG$, the d-multiple of the base point G. Thus the point Q is a randomly selected point in the group generated by G.

2.3 Elliptic Curve Key Exchange

There are several different standardized key exchange protocols (see [49, 17]) extending the basic elliptic curve Diffie-Hellman protocol, which works as follows. To agree on a shared key, Alice and Bob individually generate key pairs (d_a, Q_a) and (d_b, Q_b). They then exchange the public keys Q_a and Q_b, such that each can compute the point $P = d_aQ_b = d_bQ_a$ using their respective private keys. The shared secret key is derived from P by a key derivation function, generally being applied to its x-coordinate.

2.4 Elliptic Curve Digital Signatures

The Elliptic Curve Digital Signature Algorithm (ECDSA) was standardized in FIPS 186-4 [51]. The signer generates a key pair (d, Q) consisting of a private signing key d and a public verification key $Q = dG$. To sign a message m, the signer first chooses a per-message random integer k such that $1 \leq k \leq n - 1$, computes the point $(x_1, y_1) = kG$, transforms x_1 to an integer and computes $r = x_1 \mod n$. The message m is hashed to a bitstring of length no more than the bit length of n, which is then transformed to an integer e. The signature of m

[4] This invalid curve attack on `secp256k1` using fault injection has been mentioned before, for example by Paulo S.L.M. Barreto (@pbarreto): "In other words: given 13 faults and a good PC, one can break secp256k1 (and Bitcoin) in 1 minute.", October 21, 2013, 10:20 PM, Tweet.

is the pair (r, s) of integers modulo n, where $s = k^{-1}(e + dr) \mod n$. Note that r and s need to be different from 0, and k must not be revealed and must be a per-message secret, which means that it must not be used for more than one message.

It is important that the per-message secret k is not revealed, since otherwise the secret signing key d can be computed by $d \equiv r^{-1}(ks - e) \pmod{n}$ because r and s are given in the signature and e can be computed from the signed message. Even if only several consecutive bits of the per-message secrets for a certain number of signatures are known, it is possible to compute the private key (see [32]). Also, if the same value for k is used to sign two different messages m_1 and m_2 using the same signing key d and producing signatures (r, s_1) and (r, s_2), then k can be easily computed as $k \equiv (s_2 - s_1)^{-1}(e_1 - e_2) \pmod{n}$, which then allows recovery of the secret key.

3 Applications of Elliptic Curves

In this section, we survey deployments of elliptic curve cryptography in the real world and provide statistics on usage.

3.1 Bitcoin

The cryptocurrency Bitcoin is a distributed peer-to-peer digital currency which allows "online payments to be sent directly from one party to another without going through a financial institution" [40]. The (public) Bitcoin block chain is a journal of all the transactions ever executed. Each block in this journal contains the SHA-256 [50] hash of the previous block, hereby chaining the blocks together starting from the so-called genesis block. In Bitcoin, an ECDSA private key typically serves as a user's account. Transferring ownership of bitcoins from user A to user B is realized by attaching a digital signature (using user A's private key) of the hash of the previous transaction and information about the public key of user B at the end of a new transaction. The signature can be verified with the help of user A's public key from the previous transaction. Other issues, such as avoiding double-spending, are discussed in the original document [40].

The cryptographic signatures used in Bitcoin are ECDSA signatures and use the curve `secp256k1` (see Section 2). Given an ECDSA (possibly compressed) public-key K, a Bitcoin address is generated using the cryptographic hash functions SHA-256 and RIPEMD-160 [22]. The public-key is hashed twice: HASH160 = RIPEMD-160(SHA-256(K)). The Bitcoin address is computed directly from this HASH160 value (where $\|$ denotes concatenation) as

$$\text{base58}(\texttt{0x00} \,\|\, \text{HASH160} \,\|\, \lfloor \text{SHA-256}(\text{SHA-256}(\texttt{0x00} \,\|\, \text{HASH160}))/2^{224} \rfloor),$$

where base58 is a binary-to-text encoding scheme[5].

By participating in the Bitcoin peer-to-peer network, we downloaded the Bitcoin block chain up to block number 252 450 (all transactions up to mid-August 2013) in the Berkeley DB [41] format. We extracted 22 159 078 transactions in plain text: this resulted in a single 26 GB file. In our dataset we have 46 254 121 valid public keys containing an elliptic curve point on the curve, and 15 291 112 of these points are unique. There are 6 608 556 unique

[5] Binary strings are translated to a radix-58 system where the 58 text characters are the digits zero to nine, the lowercase letters "a" to "z" and the uppercase letters where zero (0), lowercase ℓ (l), uppercase i (I) and uppercase o (O) are omitted to ease distinguishably of the characters.

points represented in compressed (x-coordinate only) format and 8 682 692 unique points in uncompressed format (we found 136 points which occur in both compressed and uncompressed public keys). Since it is hard to tell if address reuse is due to the same user reusing their key in Bitcoin (see e.g. [44, 38] regarding privacy and anonymity in Bitcoin), there is no simple way to check if these duplicate public keys belong to the same or different owners.

Currently (October 2013) there are over 11.5 million bitcoins in circulation with an estimated value of over 2 billion USD. Bitcoin has been analyzed before in different settings (e.g. [1, 46]), but we perform, as far as we are aware, the first asymmetric cryptographic "sanity" check; see Section 4.4.

3.2 Secure Shell (SSH)

Elliptic curve cryptography can be used in three positions in the SSH protocol. In SSH-2, session keys are negotiated using a Diffie-Hellman key exchange. RFC 5656 [48] specifies the ephemeral Elliptic Curve Diffie-Hellman key exchange method used in SSH, following SEC1 [17]. Each server has a host key that allows the server to authenticate itself to the client. The server sends its host key to the client during the key exchange, and the user verifies that the key fingerprint matches their saved value. The server then authenticates itself by signing a transcript of the key exchange. This host key may be an ECDSA public key [48]. Finally, clients can use ECDSA public keys for client authentication.

We surveyed the state of elliptic curve deployment on the server side for SSH by scanning the complete public IPv4 space in October 2013 for SSH host keys, server Diffie-Hellman values, and signature values. We also collected the list of key exchange and authentication cipher suites offered by each server. We used ZMap [23], a fast Internet-wide port scanner, to scan for hosts with port 22 open, and attempted an SSH protocol handshake with the addresses accepting connections on port 22.

In order to focus on elliptic curve values, our client offered only elliptic curve cipher suites. This resulted in us discovering several implementations that provided unexpected responses to our non-standards-compliant SSH handshake: servers that provided RSA or prime-order DSA public keys, or servers that provided empty keys.

Of the 12 114 534 hosts where we successfully collected a set of cipher suites, 1 249 273 (10.3%) supported an ECDSA cipher suite for the host key. Of these, 1 247 741 (99.9%) supported `ecdsa-sha2-nistp256`, 74 supported `ecdsa-sha2-nistp384`, and 1458 (0.1%) supported `ecdsa-sha2-nistp521`. 1 674 700 hosts (13.8%) supported some form of ECDH key exchange. Of these, 1 672 458 (99.8%) supported the suites `ecdh-sha2-nistp256`, `ecdh-sha2-nistp384`, `ecdh-sha2-nistp521` in order of increasing security, and 25 supported them in the opposite order.

We successfully collected 1 245 051 P-256, 73 P-384, and 1436 P-521 public keys. In addition, 458 689 servers responded with a DSA public key, 29 648 responded with an RSA public key, and 7 935 responded with an empty host key, despite our client only claiming ECDSA support. The hosts responsible for these responses included several kinds of routers and embedded devices.

3.3 Transport Layer Security (TLS)

In TLS, elliptic curves can arise in several locations in the protocol. RFC 4492 [9] specifies elliptic curve cipher suites for TLS. All of the cipher suites specified in this RFC use the

elliptic curve Diffie-Hellman (ECDH) key exchange. The ECDH keys may either be long-term (in which case they are reused for different key exchanges) or ephemeral (in which case they are regenerated for each key exchange). TLS certificates also contain a public key that the server uses to authenticate itself; with ECDH key exchanges, this public key may be either ECDSA or RSA.

ECC support was added to TLS [9] through an additional set of cipher suites and three extensions in the client and server hello messages. The cipher suites indicate support for a particular selection of key exchange, identity verification, encryption, and message authenticity algorithms. For example, the cipher suite `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA` uses ephemeral ECDH for a key exchange, signed with an RSA key for identity verification, and uses AES-128 [19] in CBC mode for encryption and the SHA-1 hash function in an HMAC for message authentication. In addition, if a cipher suite that involves ECC is desired, the client must include a set of supported elliptic curves in a TLS extension in its `client_hello` message.

Unlike in SSH, a TLS server does not send its full preference of cipher suites or curves that it supports. Rather, the client sends its list of supported cipher suites and elliptic curves, and the server either replies with a single cipher suite from that list or closes the connection if it does not support any cipher suites in common with the client. If the suite requires ECC, the server similarly includes only a single curve type along with the key or signature. This makes learning which curves a server supports more difficult; a client must use multiple TLS connections to offer a varying set of curves in order to learn a server's support and ordered preference.

In October 2013, we used ZMap [23] to scan the IPv4 address space on port 443, and used an event-driven program to send a specially crafted `client_hello` message to each host with the port open. We offered 38 ECDH and ECDHE cipher suites and 28 different elliptic curves. Of the 30.2 million hosts with port 443 open, 2.2 million (7.2%) supported some form of ECDH and provided an ECC public key, along with information about which curve it uses. We then connected to these hosts again, excluding their known-supported curve type from our `client_hello`'s curve list. We repeated this process until we had an empty curve list, the server disconnected with an error, or the server presented a curve that was not offered to them (a violation of the protocol). This process allowed us to learn each server's ordered preference of support across our 28 curves. We found the most commonly supported curve type across the 2.2 million ECC-supporting hosts was `secp256r1`, supported by 98% of hosts. The curves `secp384r1` and `secp521r1` were supported by 80% and 17% respectively, with the remaining curves supported by fewer than 3% of hosts each.

We also found that the majority of hosts appear to prefer smaller curve sizes over larger ones. Of the 1.7 million hosts that supported more than one curve, 98.9% support a strictly increasing curve-size preference. For example, 354 767 hosts had a preference of "`secp256r1, secp384r1, secp521r1`", while only 190 hosts had the opposite order of "`secp521r1, secp384r1, secp256r1`". This suggests that most hosts prefer lower computation and bandwidth costs over increased security.

3.4 Austrian e-ID

Physical smart cards are increasingly being deployed for user authentication. These smart cards contain cryptographic hardware modules that perform the cryptographic computations; most often, these cards contain private keys for encryption and signatures. Elliptic curve

cryptography is an attractive option for these types of deployments because of the decreased key size and computational complexity relative to RSA or large prime-order groups.

Austria's national e-ID cards contain either an RSA or ECDSA public key, and can be used to provide legally binding digital signatures. We collected 828 911 Citizen Card certificates from the LDAP database `ldap.a-trust.at` in January 2013. Each certificate contained a public key and an RSA signature from the certificate authority. 477 985 (58%) certificates contained an elliptic curve public key, and 477 785 parsed correctly using OpenSSL. Of these, 253 047 used curve P-192, and 224 738 used curve P-256.

4 Cryptographic Sanity Check

There is long history of practical problems in cryptography related to insufficient randomness. The most notorious example in recent history is the Debian OpenSSL vulnerability [52]: a 2006 change in the code prevented any entropy from being incorporated into the OpenSSL entropy pool, so that the state of the pool was dependent only on the process ID and architecture of the host machine. A fixed number of cryptographic keys, nonces, or other random values of a given size could ever be generated by these implementations. The problem was discovered in 2008.

In 2012 two different teams of researchers showed independently that a significant number of RSA keys (not considering the keys affected due to the Debian OpenSSL bug) are insecure due to insufficient randomness [35, 30]. The latter paper also examined prime-order DSA SSH host keys and signatures, and found a significant number of SSH host keys could be compromised due to poor randomness during signature generation. Most of the vulnerable keys were attributed to poor entropy available at first boot on resource-limited embedded and headless devices such as routers. In 2013, another paper showed that a number of RSA keys obtained from Taiwan's national Citizen Digital Certificate database could be factored [4] due to a malfunctioning hardware random number generator on cryptographic smart cards.

In order to verify if similar vulnerabilities occur in the setting of elliptic curve cryptography, we gathered as much elliptic curve data as we could find and performed a number of cryptographic sanity checks:

4.1 Key Generation

An elliptic curve public key is a point $Q = dG$ which is a multiple of the generator G for $1 \leq d < n$. Poor randomness might manifest itself as repeated values of d, and thus repeated public keys observed in the wild. In contrast to RSA, where poor random number generators and bugs have resulted in distinct RSA moduli that can be factored using the greatest common divisor algorithm when they share exactly one prime factor in common, an elliptic curve public key appears to have no analogous property. We are unaware of any similar mathematical properties of the public keys alone that might result in complete compromise of the private keys, and they are unlikely to exist because discrete logarithms have strong hardcore properties [10, 33].

We checked for these problems by looking for collisions of elliptic curve points provided in public keys. In practice, however, it is not uncommon to encounter the same public key multiple times: individuals can use the same key for multiple transactions in Bitcoin or the same key pair can be used to protect different servers owned by the same entity.

4.2 Repeated Per-Message Signature Secrets

ECDSA signatures are randomized: each signature consists of two values (r, s): the value r is derived from an ephemeral public key kG generated using a random per-message secret k, and a signature value s that depends on k. It is essential for the security of ECDSA that signers use unpredictable and distinct values for k for every signature, since predictable or repeated values allow an adversary to efficiently compute the long-term private key from one or two signature values, as explained in Section 2. In a widely known security failure, the Sony PlayStation 3 video game console used a constant value for signatures generated using their ECDSA code signing key, allowing hackers to compute the secret code signing key [15].

We checked for these problems by parsing each signature and checking for colliding values of the ephemeral public key. In contrast to the above public key setting, there is no possible legitimate explanation for the observation of repeated signature secrets.

4.3 Unexpected, Illegal, and Known Weak Values

We also checked for public keys corresponding to the point at infinity, points that do not lie on the curve, and "public keys" that possibly do not have corresponding private keys. In addition, we generated a large list of elliptic curve points for which we know the private key. This is realized by multiplying the generator of the curve, as specified in the standard, by various integers s from different sets in the hope that poor entropy might have generated these scalars. We computed the elliptic curve scalar multiplication sG for these different values of the scalar s and stored the x-coordinate of this resulting point in a database (by restricting to the x-coordinate we represent both points $\pm sG$). We checked these self-generated points in this database against all the elliptic curve points extracted from the ECDSA public-keys and signatures to verify if we find collisions: if so, we can compute the private key. We considered three different sets in the setting of the `secp256k1` curve (as used in Bitcoin) and the NIST P-256 curve. The first set contains small integers i: where $10^0 \leq i \leq 10^6$. The second set contains 256-bit scalars of low Hamming weight: we used integers of Hamming-weight one ($\binom{256}{1} = 256$ scalars), two ($\binom{256}{2} = 32\,640$ scalars), and three ($\binom{256}{3} = 2\,763\,520$ scalars). The third set contains the Debian OpenSSL vulnerable keys. We generated the set of scalars produced by the broken Debian OpenSSL implementation run on a 64-bit little-endian byte order architecture implementation. For the Bitcoin curve we extended the first set by also considering the scalars $i\lambda$ such that the scalar multiplication corresponds to $i\lambda P = \psi(iP)$ (see Section 2). We did not find any matches between these self-generated points and the points encountered in the wild.

4.4 Bitcoin

Repeated Per-Message Secrets. We extracted 47 093 121 elliptic curve points from the signatures and verified that they are correct: i.e. the points are on the curve `secp256k1` (see Section 2). We also looked for duplicated nonces in the signature and found that 158 unique public keys had used the same signature nonces r value in more than one signature, making it possible to compute these users' private keys. We find that the total remaining balance across all 158 accounts is small: only 0.00031217 BTC, which is smaller than the transaction fee needed to claim them. However, we find that one address, `1HKywxiL4JziqXrzLKhmB6a74ma6kxbSDj`, appears to have stolen bitcoins from 10 of these addresses. This account made 11 transactions between March and October 2013. Each transaction contained inputs from addresses

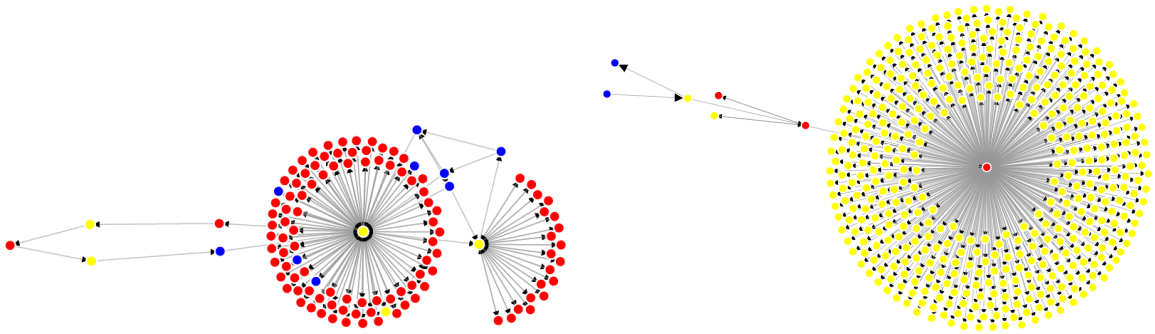

Fig. 1: Visualization of transactions between Bitcoin addresses that duplicated signature nonces (red), and addresses one (yellow) and two (blue) hops away in the transaction graph. The unique pattern across graphs suggests that multiple distinct implementations or usage patterns may be to blame for the generation of repeated nonces that expose users' private keys.

that duplicated signature nonces, and appear in our list. These transactions have netted this account over 59 bitcoins (approximately $12,000 USD).

To understand the root causes of the repeated signature nonces, we made a graph of transactions, starting with the vulnerable addresses and adding edges to other addresses indicating if they had sent or received bitcoins to one another. Next, we created edges from those second layer addresses, terminating the graph 2 degrees from the original vulnerable keys. This resulted in five distinct connected components, with the largest connected component containing 1649 addresses. Figure 1 shows the second and third largest connected graphs. The unique patterns of these two graphs suggest that there are several sets of unique users or implementations at play creating these types of failure.

We were able to identify three keys belonging to Bitcoincard [8], an embedded device that acts as a standalone Bitcoin client. We also identified several Blockchain.info accounts that duplicated nonces due to a bug in a Javascript-client's random number generator not being seeded correctly [28]. These funds were then subsequently transferred to the same address mentioned above. Note that there exists a timestamping scheme for Bitcoin that purposely leaks the private key of a transaction by deliberately using the same random nonce [18]. If this scheme is implemented and tested, then this might explain signed transactions with duplicated nonces containing a small amount.

Unspendable Bitcoins. It is possible to transfer bitcoins to an account for which (most likely) no corresponding cryptographic key-pair exists. These bitcoins remain stuck at these accounts forever and are essentially removed from circulation. This might result in deflation: increasing the value of the other (spendable) bitcoins. This situation is not Bitcoin specific; physical money can also be damaged or mutilated. The U.S. Department of the Treasury redeems mutilated currency valued at over 30 million USD per year [14]. We were unable to find information regarding the amount of money that is retracted permanently from circulation due to damage, although there are known cases where people have burned substantial amounts of money [29].

HASH160	Bitcoin address	balance in BTC
0000000000000000000000000000000000000000	1111111111111111111114oLvT2	2.94896715
0000000000000000000000000000000000000001	1111111111111111111BZbvjr	0.01000000
0000000000000000000000000000000000000002	1111111111111111111HeBAGj	0.00000001
0000000000000000000000000000000000000003	1111111111111111111QekFQw	0.00000001
0000000000000000000000000000000000000004	1111111111111111111UpYBrS	0.00000001
0000000000000000000000000000000000000005	1111111111111111111g4hiWR	0.00000001
0000000000000000000000000000000000000006	1111111111111111111jGyPM8	0.00000001
0000000000000000000000000000000000000007	1111111111111111111o9FmEC	0.00000001
0000000000000000000000000000000000000008	1111111111111111111ufYVpS	0.00000001
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	1GZQKjsC97yasxRj1wtYf5rC61AxpR1zmr	0.00012000
ffffffffffffffffffffffffffffffffffffffff	1QLbz7JHiBTspS962RLKV8GndWFwi5j6Qr	0.01000005
151 miscellaneous ASCII HASH160 values		1.32340175

public key	valid encoding	point on curve	Bitcoin address	balance in BTC
$\emptyset$	×	×	1HT7xU2Ngenf7D4yocz2SAcnNLW7rK8d4E	68.80080003
00	✓	✓	1FYMZEHnszCHKTBdFZ2DLrUuk3dGwYKQxh	2.08000002
$0\,\overset{128}{\ldots}\,0$	×	×	13VmALKHkCdSN1JULkP6RqW3LcbpWvgryV	0.00010000
$040\,\overset{126}{\ldots}\,0$	✓	×	16QaFeudRUt8NYy2yzjm3BMvG4xBbAsBFM	0.01000000

Table 1: A summary of the interesting HASH160 and public key values used in the Bitcoin block chain with the corresponding Bitcoin address and balance. Most likely, these addresses have no valid private key, leaving the account balances unspendable. The dots in the notation $0\,\overset{128}{\ldots}\,0$ represent 128 zeros (the key has 130 zeros in all). We find these addresses hold a total of 75 unspendable BTC.

We investigate a lower bound on the number of "unspendable" bitcoins. Since the HASH160 values and the Bitcoin addresses (which are directly derived from this HASH160 value) are an integral part of the Bitcoin block chain (i.e. the transaction history), people have used "interesting" invalid values for the ECDSA public-key or used the HASH160 value to embed a message. Such transactions to addresses without corresponding cryptographic key-pair are possible since the actual ECDSA keys are only required when the money in these accounts is spent. Given a Bitcoin address, or HASH160 value, it is infeasible to compute the corresponding cryptographic key-pair (since this requires computing preimages of the hash function used). In this section we assume that the interesting (or strange) values we encounter do not correspond to a valid cryptographic key-pair. Of course, it is possible (but unlikely) that these were generated in a valid manner.

Interesting HASH160 Values. Since no cryptographic key is required to generate a Bitcoin address, just a HASH160 value, our first idea was to check for addresses which have a HASH160 value which is a small integer i, where $0 \leq i < 100$. We found that the first nine values all exist and have a non-zero balance. This motivated us to search for repeated patterns when the HASH160 is displayed in hexadecimal. All of these 16 possibilities exist and three of them have a non-zero balance; see Table 1.

People have sometimes used HASH160 values to embed an ASCII encoded string into one or multiple HASH160 values within a transaction. ASCII encodes 128 specific characters (97 printable and 33 non-printable). The probability that an ECDSA public key results in a hexadecimal written HASH160 containing ASCII characters only is 2^{-20} (where we assume

the cryptographic hash functions used outputs uniform random data). Our dataset contains 53 019 716 HASH160 values (16 526 211 unique). Hence, we expect to find approximately 16 valid Bitcoin addresses with a HASH160 value containing ASCII characters only.

In our dataset we found 248 ASCII-only HASH160 values (180 unique). Out of these, 20 unique addresses have spent their money; i.e. they correspond to a valid Bitcoin address. This is in line with our estimate of 16. Out of the other 160 unique addresses 137 have a non-zero balance. When inspecting these values it is clear that people have inserted various messages in the Bitcoin transaction history (the messages range from a happy birthday message to a tribute). Typically only a small number of bitcoins are used in these transactions. See Table 1 for the details.

Interesting ECDSA Public Keys. Following the same reasoning as in the HASH160 setting, one could use "interesting" values for the public key itself. Before we outline our search for such values, let us recall the format of ECDSA public keys as specified in [17] where we assume the keys are represented in their hexadecimal value (this is the setting used in Bitcoin). A point $P = (x, y)$ can be represented as follows where $p = 2^{256} - 2^{32} - 977$ is the prime used in Bitcoin.

- If P is the point at infinity, then it is represented by the single byte `00`.
- An *uncompressed* point starts with the byte `04` followed by the 256-bit x- and 256-bit y-coordinate of the point ($\texttt{04} \parallel x \parallel y$). Hence $2\lceil \log_2(p)/8 \rceil + 1 = 65$ bytes are used to represent a point.
- A point is *compressed* by first computing a parity bit b of the y-coordinate as $b = (y \bmod 2) + 2$ and converting this to a byte value ($b \in \{\texttt{02}, \texttt{03}\}$). The $\lceil \log_2(p)/8 \rceil + 1 = 33$-byte compressed point is written as $b \parallel x$.

Similar to the HASH160 search, we started by looking for points that encode a small integer value. We generated all the Bitcoin addresses corresponding to the public keys with values the first 256 integers i ($0 \leq i < 256$) and various values for the parity bit. We used a single byte containing i, a 33-byte value $b \parallel 0 \overset{60}{\ldots} 0 \parallel i$, $b \in \{\texttt{00}, \texttt{02}, \texttt{03}\}$, and a 65-byte value $b \parallel 0 \overset{124}{\ldots} 0 \parallel i$, for $b \in \{\texttt{00}, \texttt{04}\}$. We found three addresses with a non-zero balance: the single byte `00`, and the 65-byte $b \parallel 0 \overset{124}{\ldots} 0 \parallel i$ for $i = \texttt{00}$ and $b \in \{\texttt{00}, \texttt{04}\}$. This first point is the point at infinity, which is a correctly encoded and valid point on the curve. Note, however, that this value is explicitly prohibited as a public key [17] since it can only occur for the private key $d = 0$ which is not allowed. The 65-byte values both seem to try and encode the point at infinity: in the case where $b = \texttt{00}$ the encoding is invalid while in the case $b = \texttt{04}$ the encoding is valid but the point $(x, y) = (0, 0)$ is not on the curve.

When looking for other values, we also tried the empty public key ($\emptyset$). This address contains a significant amount of bitcoins (over 68 BTC). We suspect money has been transferred to this account due to software bugs. These results are included in Table 1. In total we found that at least 75 BTC (over 14 000 USD) has been transferred to accounts which have (most likely) no valid corresponding ECDSA private key. Note that this is strictly a lower bound on the number of unspendable bitcoins, as we do not claim that this list is complete.

4.5 Secure Shell (SSH)

Duplicate public keys. An August 2013 SSH scan collected 1 353 151 valid elliptic curve public keys, of which 854 949 (63%) are unique. There were 1 246 560 valid elliptic curve public

keys in the October 2013 scan data, of which 848 218 (68%) are unique. We clustered the data by public key. Many of the most commonly repeated keys are from cloud hosting providers. For these types of hosts, repeated host keys could be due either to shared SSH infrastructure that is accessible via multiple IP addresses, in which case the repeated keys would not be a vulnerability, or they could be due to mistakes during virtual machine deployment that initialize multiple VMs for different customers from a snapshot that already contains an SSH host key pair. It appears that both cases are represented in our dataset. Digital Ocean released a security advisory in July 2013 [21] recommending that customers should regenerate SSH host keys due to repeated keys deployed on VM snapshots; we found 5 614 hosts that had served the public key whose fingerprint appears in Digital Ocean's setup guide.

We were also able to identify several types of network devices that appeared to be responsible for repeated host keys, either due to default keys present in the hardware or poor entropy on boot. We were able to attribute the repeated keys to these implementations because these devices served login pages over HTTP or HTTPS which identified the manufacturer and brand. We were unable to easily give an explanation for most of the repeated keys, as (unlike in the results reported in [30]) many of the clusters of repeated keys appeared to have almost nothing in common: different SSH versions and operating systems, different ports open, different results using nmap host identification, different content served over HTTP and HTTPS, and IP blocks belonging to many different hosting providers or home/small commercial Internet providers. We can speculate that some of these may be VM images, but in many cases we have no explanation whatsoever. We can rule out Debian weak keys as an explanation for these hosts, because the Debian bug was reported and fixed in 2008, while OpenSSH (which is almost universally given in the client version strings for the elliptic curve results) introduced support for elliptic curve cryptography in 2011.

We checked for repeated signature nonces and did not find any. We also checked for overlap with the set of TLS keys we collected and did not find any.

4.6 Transport Layer Security (TLS)

Duplicate public keys. Although we collected a total of over 5.4 million public keys from ECDH and ECDHE key exchanges, only 5.2 million of these were unique. As observed in [12], OpenSSL's default behavior is to use ephemeral-static ECDH (the key pair is ephemeral for each application instance and not (necessarily) per handshake instance) which might explain some of the observed duplicate keys. We found 120 900 distinct keys that were presented by more than one IP address, with the most common duplicated key presented by over 2 000 hosts. Many of these duplicated keys appear to be served from a single or small set of subnets, and appear to serve similarly configured web pages for various URLs, suggesting that these are part of a single shared hosting.

We also discovered one instance of a default key being used on a device sold to different consumers. We found about 1 800 of one brand of devices that present the same `secp256r1` public key for their ECDHE key exchange. Each device must also have the same private key, allowing an attacker who buys or compromises one device to passively decrypt traffic to other devices.

Duplicate server randomness. We also were surprised to find that several hosts duplicated the 32-byte random nonce used in the server hello message. We found 20 distinct nonces that were used more than once; 19 of which were re-used by more than one IP address. The most

repeated server random was repeated 1 541 times and was simply an ASCII string of 32 "f" characters. These devices all appear to be a UPS power monitor. However, we were unable to successfully establish any TLS sessions with these devices, either using a browser or OpenSSL.

For servers that happen to always duplicate a server random, it is clear there is an implementation problem to be fixed. However, for servers that only occasionally produce the same server random, it is indeed more troubling. More investigation is required to find the root cause of these collisions and determine if the problem extends to cryptographic keys.

4.7 Austrian e-ID

Of the 477 985 elliptic curve public keys that we extracted from the Austrian Citizen Card certificate database, 24 126 keys appear multiple times. However, in all but 5961 of these cases, the certificate subjects were equal. Of the nonequal subjects, all but 70 had identical "CN" fields. All of these remaining certificates with identical public keys issued to nonequal names appeared to be due to either minor character encoding or punctuation differences or name changes. Hence, there appears to be no abnormalities with the ECDSA keys in this dataset.

5 Conclusions

We explore the deployment of elliptic curve cryptography (ECC) in practice by investigating its usage in Bitcoin, SSH, TLS, and the Austrian citizen card. More than a decade after the first ECC standardization we find that this instantiation of public key cryptography is gaining in popularity. Although ECC is still far from the dominant choice for cryptography, the elliptic curve cryptographic landscape shows considerable deployment in 2013. Of the 12 million scanned hosts which support SSH, we found that 10.3% supported ECDSA for authentication and 13.8% supported a form of ECDH for key exchange. We scanned 30.2 million TLS servers and found that 7.2% support a form of ECDH. In the Austrian citizen card database, 58% of the 829 000 use ECDSA to create digital signatures. All asymmetric cryptography in Bitcoin is based on ECC.

Our cryptographic sanity checks on these datasets confirmed that, as expected, ECC is not immune to insufficient entropy and software bugs. We found many instances of repeated public SSH and TLS keys, some of which seem to correspond to different owners. For the Bitcoin data set, there are many signatures sharing ephemeral nonces, allowing attackers to compute the corresponding private keys and steal coins.

We hope that our work will encourage researchers and developers alike to remain diligent in discovering and tracking down these types of implementation problems, ultimately improving the security of the cryptographic protocols and libraries we depend on.

Acknowledgments

We thank Jaap W. Bos for valuable discussions about the financial market, Andy Modell for support in TLS scanning, and Sarah Meiklejohn for sharing her knowledge about Bitcoin.

References

1. S. Barber, X. Boyen, E. Shi, and E. Uzun. Bitter to better - how to make bitcoin a better currency. In A. D. Keromytis, editor, *Financial Cryptography*, volume 7397 of *LNCS*, pages 399–414. Springer, 2012.

2. D. J. Bernstein. A software implementation of NIST P-224. http://cr.yp.to/talks.html#2001.10.29, 2001.
3. D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006.
4. D. J. Bernstein, Y.-A. Chang, C.-M. Cheng, L.-P. Chou, N. Heninger, T. Lange, and N. van Someren. Factoring RSA keys from certified smart cards: Coppersmith in the wild (to appear). In *ASIACRYPT*, LNCS. Springer, 2013.
5. D. J. Bernstein and T. Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. http://safecurves.cr.yp.to, accessed 31 October 2013.
6. D. J. Bernstein and T. Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. http://bench.cr.yp.to, October 2013.
7. I. Biehl, B. Meyer, and V. Müller. Differential fault attacks on elliptic curve cryptosystems. In M. Bellare, editor, *CRYPTO*, volume 1880 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2000.
8. bitcoincard.org. Sample transaction. http://bitcoincard.org/blog/?page=post&blog=bitcoincard_blog&post_id=sample_yransaction, 2012.
9. S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic curve cryptography (ECC) cipher suites for transport layer security (TLS). RFC 4492, 2006.
10. D. Boneh and I. Shparlinski. On the unpredictability of bits of the elliptic curve Diffie–Hellman scheme. In J. Kilian, editor, *CRYPTO*, volume 2139 of *LNCS*, pages 201–212. Springer, 2001.
11. E. Brier, M. Joye, and T. E. D. Win. Weierstra elliptic curves and side-channel attacks. In *Public Key Cryptography PKC 2002, volume 2274 of LNCS*, pages 335–345. Springer, 2002.
12. B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren. Practical realisation and elimination of an ECC-related software bug attack. In O. Dunkelman, editor, *CT-RSA*, volume 7178 of *LNCS*, pages 171–186. Springer, 2012.
13. B. B. Brumley and R. M. Hakala. Cache-timing template attacks. In M. Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 667–684. Springer, 2009.
14. Bureau of Engraving and Printing – U.S. Department of the Treasury. Damaged currency. http://moneyfactory.gov/uscurrency/damagedcurrency.html, 2013.
15. "Bushing", H. M. Cantero, S. Boessenkool, and S. Peter. PS3 epic fail. http://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf, 2010.
16. Certicom Research. Standards for efficient cryptography 2: Recommended elliptic curve domain parameters. Standard SEC2, Certicom, 2000.
17. Certicom Research. Standards for efficient cryptography 1: Elliptic curve cryptography. Standard SEC1, Certicom, 2009.
18. J. Clark and A. Essex. CommitCoin: Carbon dating commitments with bitcoin - (short paper). In A. D. Keromytis, editor, *Financial Cryptography*, volume 7397 of *LNCS*, pages 390–398. Springer, 2012.
19. J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
20. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
21. DigitalOcean. Avoid duplicate SSH host keys. https://www.digitalocean.com/blog_posts/avoid-duplicate-ssh-host-keys, 2013.
22. H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A strengthened version of RIPEMD. In D. Gollmann, editor, *FSE*, volume 1039 of *LNCS*, pages 71–82. Springer, 1996.
23. Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *Proceedings of the 22nd USENIX Security Symposium*, August 2013.
24. I. M. Duursma, P. Gaudry, and F. Morain. Speeding up the discrete log computation on curves with automorphisms. In K.-Y. Lam, E. Okamoto, and C. Xing, editors, *ASIACRYPT*, volume 1716 of *LNCS*, pages 103–121. Springer, 1999.
25. P. Fouque, R. Lercier, D. Real, and F. Valette. Fault attack on elliptic curve Montgomery ladder implementation. In *FDTC*, pages 92–98, 2008.
26. R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In J. Kilian, editor, *CRYPTO*, volume 2139 of *LNCS*, pages 190–200. Springer, 2001.
27. M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 38–49. ACM, 2012.

28. D. Gilson. Blockchain.info issues refunds to Bitcoin theft victims. http://www.coindesk.com/blockchain-info-issues-refunds-to-bitcoin-theft-victims/, Aug 2013.
29. Gimpo. Watch the K Foundation Burn a Million Quid. http://www.imdb.com/title/tt0114897/, 1995.
30. N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.
31. A. Hollosi, G. Karlinger, T. Rössler, M. Centner, and et al. Die österreichische bürgerkarte. http://www.buergerkarte.at/konzept/securitylayer/spezifikation/20080220/, 2008.
32. N. Howgrave-Graham and N. P. Smart. Lattice attacks on digital signature schemes. *Des. Codes Cryptography*, 23(3):283–290, 2001.
33. D. Jetchev and R. Venkatesan. Bits security of the elliptic curve Diffie-Hellman secret keys. In D. Wagner, editor, *CRYPTO*, volume 5157 of *LNCS*, pages 75–92. Springer, 2008.
34. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.
35. A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter. Public keys. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO*, volume 7417 of *LNCS*, pages 626–642. Springer, 2012.
36. A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.
37. K. Michaelis, C. Meyer, and J. Schwenk. Randomly failed! The state of randomness in current Java implementations. In E. Dawson, editor, *CT-RSA*, volume 7779 of *LNCS*, pages 129–144. Springer, 2013.
38. I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous distributed E-Cash from Bitcoin. In *IEEE Symposium on Security and Privacy*, pages 397–411. IEEE Computer Society, 2013.
39. V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *CRYPTO*, volume 218 of *LNCS*, pages 417–426. Springer, 1986.
40. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2009.
41. M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191. USENIX, 1999.
42. D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In D. Pointcheval, editor, *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.
43. J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978.
44. F. Reid and M. Harrigan. An analysis of anonymity in the bitcoin system. In *SocialCom/PASSAT*, pages 1318–1326. IEEE, 2011.
45. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.
46. D. Ron and A. Shamir. Quantitative analysis of the full bitcoin transaction graph. In A.-R. Sadeghi, editor, *Financial Cryptography*, volume 7859 of *LNCS*, pages 6–24. Springer, 2013.
47. J. A. Solinas. Generalized Mersenne numbers. Technical Report CORR 99–39, Centre for Applied Cryptographic Research, University of Waterloo, 1999.
48. D. Stebila and J. Green. Elliptic curve algorithm integration in the secure shell transport layer. RFC 5656, 2009.
49. U.S. Department of Commerce/National Institute of Standards and Technology. Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography. Special Publication 800-56A, 2007. http://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.pdf.
50. U.S. Department of Commerce/National Institute of Standards and Technology. Secure Hash Standard (SHS). FIPS-180-4, 2012. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.
51. U.S. Department of Commerce/National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS-186-4, 2013. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.
52. S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage. When private keys are public: results from the 2008 Debian OpenSSL vulnerability. In A. Feldmann and L. Mathy, editors, *Internet Measurement Conference*, pages 15–27. ACM, 2009.