Identifying Key Leakage of Bitcoin Users

Michael Brengel[1] and Christian Rossow[2]

[1] CISPA, Saarland University
`michael.brengel@cispa.saarland`
[2] CISPA, Saarland University
`rossow@cispa.saarland`

Abstract. We study key leakage in the context of cryptocurrencies. First, we consider the problem of *explicit* key leakage occurring on open-source intelligence platforms. To do this, we monitor the Pastebin feed from Sep 2017–Mar 2018 to find exposed secret Bitcoin keys, revealing that attackers could have stolen 22.40 BTC worth roughly \$178,000 given current exchange rates. Then, we focus on *implicit* key leakage by exploiting the wrong usage of cryptographic primitives and scan Bitcoin's blockchain for ECDSA nonce reuse. We systematically outline how an attacker can use duplicate r values to leak nonces and secret keys, which goes beyond the simple case where the same nonce and the same key have been used in conjunction more than once. Our results show that ECDSA nonce reuse has been a recurring problem in the Bitcoin ecosystem and has already been exploited by attackers. In fact, an attacker could have exploited nonce reuse to steal 412.80 BTC worth roughly \$3.3 million.

1 Introduction

Cryptocurrencies have become popular entities in global financial markets. A prime example of such a currency is *Bitcoin* [17] with a current market capitalization of over \$135 billion [1] or *Ethereum* [23] with a current market capitalization of over \$44 billion [2]. As such, it comes as no surprise that malicious actors constantly try to *steal* those currencies, i.e., change ownership of cryptocurrency assets without consent of the legitimate owners. The decentralized and *anonymous* (or at least *pseudonymous*) nature of those currencies makes such malicious activities more attractive, as traceback and prosecution by law enforcement agencies is significantly harder than with traditional currencies.

In terms of stealing cryptocurrency assets, there are several possibilities. A cryptocurrency is usually based on a *cryptographic protocol*, which uses several *cryptographic primitives* such as *elliptic curves* [15] or *digital signatures* [12], which one could try to attack. However, both the protocol and the primitives are usually well studied and are either proven secure in theory, or have been subject to an auditing process by experts in the field. Therefore, the best attackers can hope for in this setting are implementation flaws, which are usually short-lived due to the open-source nature of cryptocurrency implementations. The most prominent incident of such an implementation flaw happened in February 2014, when attackers found a vulnerability in the Mt. Gox Bitcoin exchange, which allowed them to steal 850,000 BTC worth around $450 million at that time. While the attack did not affect the Bitcoin protocol itself, it exploited the inherent *transaction malleability* of Bitcoin transactions to break some assumptions of the internal accounting system of Mt. Gox [11].

While such large-scale incidents are rare, a more common and thus also severe class of attacks against cryptocurrencies aims to leak cryptographic keys. Cryptocurrency assets are cryptographically protected by a collection of *secret keys*, which is called a *wallet*. If this wallet is stored in an insecure manner, i.e., in plain on disk without any additional protection, then malware can simply scan the disk for such wallets and report them to the attacker, which in turn can use them to steal assets. Due to the popularity of cryptocurrencies, attackers have massively deployed malware that aims to leak such secret keys. A well-known case of such malware was the Pony Botnet, which operated from September 2013 to January 2014 [18]. The malware scanned the victim's machine for various confidential credentials including cryptocurrency keys, which resulted in financial damage of $220,000. Modern wallets now use more sophisticated means of key management such as additional encryption with a password, two-factor authentication or hardware-based security [13], which protects against such local attacks.

In this paper, we take a different perspective and study whether *remote* attack vectors allow leaking cryptographic keys from users. First, we study whether users (accidentally or knowingly) *explicitly* leak cryptographic keys, that is, post them publicly. To this end, we leverage the notion of *open-source intelligence* (OSINT) with respect to cryptocurrency leaks. As a case study, we consider Bitcoin as it is the most prevalent cryptocurrency currently used, but any other cryptocurrency would be suitable as well. As an OSINT platform we consider Pastebin [3], which is a popular information-sharing web application on the Internet, and has already proven to leak different types of privacy-related information [16]. However, other OSINT platforms such as Twitter, Reddit, Facebook or GitHub would also work. We envision a scenario where a victim uses Pastebin to share a piece of information including Bitcoin secrets such as a code snippet performing a transaction or the debug output of wallet software. The victim creates this *paste* to privately share the information, not knowing that it will be publicly available in the Pastebin feed. An attacker that monitors this feed can then scan each new paste for Bitcoin keys, for example using their well-known format, and use those keys to steal Bitcoins. To simulate this, we have monitored the Pastebin feed since September 2017 for Bitcoin secrets. Our results show that an attacker could have stolen 22.40 BTC during this timespan.

We then also study the possibility of *implicit* key leakage, given that cryptocurrency users (or software developers) may misapply cryptographic primitives. In particular, keeping our focus on Bitcoin, we study the incorrect use of the *Elliptic Curve Digital Signature Algorithm* (ECDSA), which, however, also applies to other cryptocurrencies that are based on this primitive. To sign a message m using ECDSA with a secret key sk, one must compute a signature, which involves a randomly chosen nonce k. It is well known that apart from the secret key, the nonce must also be kept secret, as an attacker can otherwise use the signature and k to retrieve sk. Similarly, if one signs two distinct messages m_1 and m_2 using the same k and the same sk, then an attacker can recompute sk based on the structure of the signature and the knowledge that both the key and the nonce have been reused. While such a duplicate occurrence should not happen in practice, as the set of possible nonces is sufficiently large, i.e., almost 2^{256} in the case of Bitcoin, such duplicates can still appear for other reasons. One such reason could be the use of weak random number generators [4] or vulnerable software that is not aware of the implications of nonce reuses. Another scenario which could also be responsible for such duplicate occurrences is cloning or resetting a virtual machine, which could possibly result in reusing the same seed for the random number generator. While there is anecdotal evidence for duplicate nonces in the Bitcoin blockchain, there is no systematic study on the actual impact or the prevalence of this phenomenon, i.e., the potential financial damage that can be caused. To fill this gap, we scan the Bitcoin blockchain for duplicate nonces and simulate an attack scenario in which a malicious actor actively monitors incoming transactions to look for duplicate nonce occurrences to leak keys and steal Bitcoins. In particular, we systematically outline how an attacker can use duplicate nonces to leak secrets, which has not been shown before in such detail. This goes beyond naïve cases where the same key and nonce pair was used twice to sign two distinct messages. In fact, we show that it is also possible to leak secrets by exploiting cyclic dependencies between keys and duplicate nonces. Our results show that an attacker could have used this methodology to steal 412.80 BTC.

To summarize, our contributions are as follows: (i) We assess the threat of explicit Bitcoin key leaks using OSINT. We instantiate this idea by monitoring the public feed of Pastebin for leaked secret keys. Our results demonstrate how an attacker doing this could have stolen 22.40 BTC. (ii) We systematically demonstrate how attackers can monitor Bitcoin transactions to scan for implicit key leaks. We develop a methodology that can map signatures with duplicate nonces to linear equation systems using a bipartite graph representation. (iii) We assess the impact of implicit key leaks in the context of Bitcoin. That is, we analyze how prevalent they are and how much Bitcoins an attacker could have stolen by exploiting them. Finally, we study if such exploitation has happened in the past. Our results show that an attacker could have stolen 412.80 BTC and that attackers have exploited nonce reuse in the past to steal Bitcoins.

2 Background

In this section, we outline the preliminaries required for the scope of this paper in order to grasp our ideas using the Bitcoin technology.

Blockchain and Mining. The central component of the Bitcoin protocol is the Bitcoin *blockchain*, which is a distributed append-only log, also called a *ledger*. The idea of this ledger is to keep track of all transactions that have ever occurred in the Bitcoin network. The ledger consists of a sequence of *blocks*, each of which consists of a set of *transactions*. Adding such a block to the blockchain requires solving a computational puzzle using the Hashcash proof-of-work system [9]. The process of adding blocks to the blockchain is called *mining* and is rewarded with Bitcoins. Transactions and blocks are created and distributed by the peers of the network. Before transactions are mined, they are put in a temporary buffer called the *mempool*. *Miners*, i.e., the peers which mine blocks, will then take transactions from the mempool to build and mine a block and finally, announce a newly mined block to the network.

Transactions. A Bitcoin transaction T consists of a sequence of *inputs* $T_i = [i_1, \ldots, i_m]$ and a sequence of *outputs* $T_o = [o_1, \ldots, o_n]$ and is uniquely identified by a *transaction ID*, which is generated by computing a hash of the transaction. Inputs and outputs are therefore uniquely identified by the ID of the transaction which contains them and their index in the input list and output list, respectively. An output $o_j \in T_o$ carries a value, which is the number of *satoshis* that this output is worth. A satoshi is defined to be such that one Bitcoin (BTC) equals 10^8 satoshis. The purpose of a transaction is to spend outputs by creating new ones, which represents the money flow. To do this, every input $i_j \in T_i$ uniquely references an output of another previous transaction, i.e., the ones which will be spent, and creates new outputs that can be spent by future transactions. An output can only be referenced once, and the outputs in the blockchain which have not been referenced at any given moment in time is called the set of *unspent outputs*. Every transaction carries an implicit *transaction fee*, which is the difference between the sum of the values of the outputs and the sum of the value of the referenced outputs. Transaction fees will be paid to the miners, which thus prioritize transactions based on their fees, i.e., the higher the fee, the faster the transaction will be mined. Since a block can only be 1 MiB in size, miners will usually consider transaction fees as a function of satoshis per byte of the transaction, i.e., the larger the transaction the larger the nominal value of the fee should be. Transaction fees are an essential economical element of the Bitcoin network and change constantly depending on the number of transactions in the mempool and how much peers are willing to pay the miners. Special transactions without any inputs referencing other outputs are so-called *coinbase* transactions and are created when a block is mined to reward the miner, which is how Bitcoins are initially created. That is, before a miner mines a block, they will first create a coinbase transaction which will be put in the block and rewards them with Bitcoins. This reward is a fixed amount, which gets halved every 210,000 blocks, plus the fees of all transactions in the block.

Scripts. Transactions in the Bitcoin network are verified by using a small stack-based language, the programs of which are called *scripts*. Every input and output contains a script, which is often referred to as scriptSig and scriptPubKey, respectively. These scripts can perform arithmetic, cryptography, flow control and so on. In order for a transaction to be valid, one must concatenate the scriptSig of each input with the scriptPubKey of its referenced output, which yields a new set of scripts, i.e., one for each input. All of these scripts are then evaluated, and for the transaction to be valid, there must be only one element on the stack after evaluation and this element must be equal to *true*. The scriptPubKey can therefore be considered a means of protection, i.e., one can only redeem an output if they can provide a correct scriptSig. The scripting language contains special instructions for elliptic curve cryptography, which is used within this scripting framework to cryptographically secure transactions. In this context, every user has a *secret key* sk and a *public key* pk. The most prevalent type of transaction is called a *Pay To Pubkey Hash* (P2PKH) transaction. Outputs belonging to such transactions have a scriptPubKey that verifies that the sender of the transaction possesses the correct public key by comparing it against a hash. Additionally, the script verifies a signature, which means that a working scriptSig must provide both the public key pk as well as a valid signature that can be verified with pk, which means that the sender must know sk.

Bitcoin Addresses. A Bitcoin address is a serialized hash of pk, which is generated by hashing the public key with the SHA-256 and the RIPMED-160 hash functions and appending and prepending a version byte and checksum bytes. The hash is then serialized using base58 encoding, which is a more human-readability-friendly version of the base64 encoding and removes ambiguous-looking characters (e.g., zero ("0") and capital o ("O")). An example of such an address is `16UwLL9Risc3QfPqBUvKofHmBQ7wMtjvM`. Before hashing, pk must be serialized, for which there are two options, namely the *compressed* public key and the *uncompressed* public key. We omit the technical details here as they are not required for the scope of this paper. It is only important that both serialization options yield different addresses, which means that every public key pk corresponds to *two* addresses, which can be used independently of each other. This means that if an attacker leaks a secret key, they gain control over the *balances* of two addresses. We can define the balance of a P2PKH address by using the previously mentioned scripts. For instance, we determine that the balance of a P2PKH address encoding a hash h, is the sum of the values of all unspent outputs that can be redeemed with the public key pk that h is a hash for.

3 Explicit Key Leaks: Open Source Intelligence

In this section, we will outline the methodology that we use to discover explicit Bitcoin key leaks, i.e., cases where users (knowingly or not) directly disclose sensitive Bitcoin key material to the public. To this end, we follow the general idea of *open source intelligence* (OSINT), in which an attacker harvests publicly available information to derive sensitive information. To evaluate this idea in the context of Bitcoin secrets, we chose Pastebin as an OSINT platform. Given its popularity, we expect that Bitcoin users accidentally leak secret information there. Examples of such leaks would be users publishing code snippets doing Bitcoin transactions or the debug output of some wallet software which users want to share privately, not knowing that these *pastes* are then publicly visible in the Pastebin feed. We monitored all pastes starting from September 2017 and scanned each paste for Bitcoin secrets, i.e., secret keys.

3.1 Finding Bitcoin Secrets

To scan a paste for secret Bitcoin keys, we leverage the observation that Bitcoin keys are serialized using a well-known format. A secret key is an integer `sk`, which we will describe further in Section 4.1. An agreed-upon format for serializing those keys is the *Wallet Import Format* (WIF). To convert a secret key `sk` into this format, the following procedure is applied. First, `sk` is converted to a 32-bytes-long big-endian representation, which we call b. Then, `0x80` is prepended to b and optionally `0x01` is appended if the secret key will correspond to a compressed public key. Then SHA256 is applied twice on b, and we call the last four bytes of this hash c. The WIF is defined to be the base58 encoding of $b||c$.

The last 4 bytes in this format are a checksum for the remaining bytes, which is used in practice to avoid copy and paste mistakes. However, this checksum also allows a systematic scan for instances of WIF strings in text with a very low probability of a false positives. In our Bitcoin monitoring tool, we thus proceed for each new paste as follows. First, we move a sliding window over the content of the paste to discover all valid base58 encoded substrings of the paste which are 51 or 52 characters long and start with either "5", "K" or "L". Both of these constraints are a consequence of the base58 encoding and the fact that the fixed byte `0x80` is prepended. For each string which matches these criteria, we compute and verify the checksum as described above. If the checksum verifies, we have found a valid WIF string and we can compute the corresponding secret key `sk`. Finally, we check if the secret key is in the valid range (cf. Section 4.1), and if this is the case, then we consider this key for further analysis.

3.2 Results

To apply our methodology, we monitored and scanned all public pastes on Pastebin since September 2017. We identified 21,464 secret keys, which correspond to 42,936 addresses, i.e., 2 addresses per key as described in Section 2. However, most of these addresses are unused, i.e., there is no transaction in the blockchain which transferred Bitcoins from or to these addresses. As of now, 391 (0.91%) of those addresses held a balance at some point in time. However, for stealing Bitcoins it is not sufficient that an address held a balance at some point in time. Instead, we also have to take into account that the address held a balance *after* we have seen the corresponding secret key in a paste. If we respect this constraint, we find that 165 (0.38%) addresses held a balance after we have seen their secret key in a paste. Those keys were scattered among a total of 34 pastes. Summing up those balances gives a total of 326.70 BTC.

It should be mentioned, though, that this is still not a guarantee that this number of Bitcoins could have been stolen. This is due to the fact that we determine the balance of an address at some point in time based on the blockchain, not the mempool. That is, we take the latest block that was mined before the paste was published and check the balance of an affected address up to this block. It could be the case that there was a transaction in the meantime which redeemed outputs from the given address, i.e., there could be a pending transaction in the mempool. In this case, an attacker could not easily create a transaction to steal the Bitcoins. Current network rules discourage the distribution of transactions that double-spend outputs unless the transaction is explicitly marked as a *replace-by-fee* (RBF) transaction. An attacker could try to mine a stealing transaction themselves or try to directly announce the stealing transaction to mining pools which do not follow these network rules. Alternatively, if the blocking transaction has a low fee, the attacker could wait until a significant number of peers do not have the transaction in their copy of the mempool anymore. This would increase the chances that the new stealing transaction will be pushed to more peers, which in turn will increase the chances that the stealing transactions will be mined. However, none of these methods guarantees success, and therefore the amount of 326.70 BTC is an upper limit.

To get a more conservative estimation of the amount of stealable Bitcoins, we have to consider pending transactions. That is, we only considered cases where there was no transaction in between which was not marked as RBF. As it turns out, this was the case for 26 addresses in 119 pastes. For the remaining cases, there was a blocking transaction in between, i.e., the paste containing the secret key was published after the blocking transaction was distributed. For example, one paste contained an address holding a balance of 40.84 BTC for which a transaction was already placed in the mempool. In total, we found that an attacker could have stolen 22.40 BTC. We excluded transaction fees in this analysis as they are highly dynamic over time and the number of stealable outputs was so small that the resulting fees would not be a significant factor.

This demonstrates that an attacker can cause significant financial loss with relatively simple means. This is amplified by the fact that an attacker could expand this methodology to other cryptocurrencies and OSINT platforms.

4 Implicit Key Leaks: Incorrectly Used Cryptography

Seeing that even explicit key leaks pose a problem to Bitcoin users, in this section, we will study how users implicitly leak secrets. To this end, we will first describe the most important cryptographic primitive in Bitcoin, namely ECDSA. We then show how the incorrect use of this primitive opens severe vulnerabilities. That is, we will systematically describe how an attacker monitoring the transactions of the Bitcoin network can use nonce reuse to steal Bitcoins, and what amount of damage could have been caused (or was caused) in the past by attackers.

4.1 Elliptic Curve Digital Signature Algorithm (ECDSA)

Bitcoin uses the *Elliptic Curve Digital Signature Algorithm* (ECDSA) to cryptographically secure transactions. The scheme is based on the computational infeasibility assumption of solving the *Elliptic Curve Discrete Logarithm Problem* (ECDLP), i.e., given two points Q and Qk on the curve, there is no polynomial-time algorithm for recovering k. Bitcoin uses the *secp256k1* curve, which is based on the equation $y^2 = x^3 + 7$ over the finite field $\mathbb{F}_p$ with the 256-bit prime number $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$. Furthermore, secp256k1 uses a *generator point* G with the 256-bit *group order* $n = 2^{256} -$ `0x14551231950B75FC4402DA1732FC9BEBF`, i.e., n is the smallest number such that $Gn = 0$. To create and verify signatures, we need the notion of a secret key sk and a public key pk. In the context of elliptic curve cryptography, sk is a randomly chosen integer from $\{1, \dots, n-1\}$ and the public key pk can be derived by multiplying the generator G with sk, i.e., $\mathsf{pk} = G\mathsf{sk}$. This derivation is considered secure, as recovering sk from pk would require solving ECDLP.

To sign a message m with a secret key sk using ECDSA, the following procedure is followed: First a hash of the message $h = H(m)$ is created using a cryptographic hash function H. The hash h is then interpreted as a number and truncated so that it does not contain more bits than the group order n. In the case of Bitcoin, we have $H = \text{SHA256}^2$, i.e., applying SHA256 twice, which means that h will not be truncated as n is a 256-bit number. Then, a random nonce k is chosen from $\{1, \dots, n-1\}$. After that, the *r value* is computed, which is the x-coordinate of the point that is yielded by multiplying the generator point G with k, which we denote by $r = (Gk)_x \bmod n$. Finally, the value $s = k^{-1}(h + r\mathsf{sk}) \bmod n$ is computed and the tuple (r, s) is returned as the signature. If $r = 0$ or $s = 0$, then this procedure is repeated until both r and s are non-zero. To verify that (r, s) is a valid signature for a message m using the public key pk, one proceeds as follows: First the hash $h = H(m)$ is created and truncated as before. Then, the curve point $(x, y) = (Gh + \mathsf{pk}r)s^{-1}$ is calculated and the signature is considered valid if $x = r$. The correctness follows from the observation that $\mathsf{pk} = G\mathsf{sk}$, which implies $(Gh + \mathsf{pk}r)s^{-1} = G(h + \mathsf{sk}r)s^{-1} = Gkss^{-1} = Gk$.

In terms of key or nonce leakage, note that the equation $s = k^{-1}(h + r\mathsf{sk}) \bmod n$ contains two unknowns and therefore cannot be used to leak the secret key or the nonce. Recovering k from $r = (Gk)_x$ would require solving ECDLP, similar to how $\mathsf{pk} = G\mathsf{sk}$ cannot be used to recover sk.

4.2 Using Duplicate Nonces to Leak Keys

It is known that ECDSA fails catastrophically if *nonce reuse* occurs. Nonce reuse means that there are multiple signatures using the same nonce k, which might allow an attacker to leak secret keys under certain circumstances. For instance, if the same k (and thereby the same r value) and sk are used to create 2 signatures (r, s_1) and (r, s_2) for two distinct messages m_1 and m_2, then we have[3]:

$$s_1 = k^{-1}(h_1 + r\mathsf{sk}) \qquad\qquad s_2 = k^{-1}(h_2 + r\mathsf{sk}), \tag{1}$$

This allows leaking the secret key sk with:

$$\frac{s_2h_1 - s_1h_2}{r(s_1 - s_2)} = \frac{h_1h_2 + rh_1\mathsf{sk} - h_1h_2 - rh_2\mathsf{sk}}{rh_1 + r\mathsf{sk} - rh_2 - r\mathsf{sk}} = \frac{rh_1\mathsf{sk} - rh_2\mathsf{sk}}{rh_1 - rh_2} = \mathsf{sk}. \tag{2}$$

[3] Note that all calculations on signatures are done modulo n, which we omit for brevity.

Similarly, k can be leaked with:

$$\frac{h_1 - h_2}{s_1 - s_2} = \frac{h_1 - h_2}{k^{-1}(h_1 - h_2 + \mathsf{sk}(r - r))} = k. \tag{3}$$

However, not every kind of nonce reuse leads to cases where an attacker can leak secrets. For instance, consider the case where a nonce k is used with two different keys sk_1 and sk_2 to sign two distinct messages, i.e.,:

$$s_1 = k^{-1}(h_1 + r\mathsf{sk}_1) \qquad s_2 = k^{-1}(h_2 + r\mathsf{sk}_2). \tag{4}$$

It turns out that it is not possible in this case to leak any secrets. To get a better understanding of this, we need to consider the fundamental underlying problem that constitutes the act of leaking secrets in this setting. If we rewrite Equation (1) to look as follows:

$$s_1 k - r\mathsf{sk} = h_1 \qquad s_2 k - r\mathsf{sk} = h_2$$

it becomes evident that this is a system of linear equations. In particular, this system consists of 2 linearly independent equations, since $h_1 \neq h_2$, and 2 unknowns, i.e., k and sk, and is therefore uniquely solvable. On the other hand, Equation (4) consists of 2 equations and 3 unknowns, i.e., k, sk_1 and sk_2, and is therefore not uniquely solvable as there are more unknowns than equations.

4.3 Beyond Single-Key Nonce Reuse

Interestingly, in some cases secrets leak even though the nonces are not reused with the same secret key. For example, consider the following case, where two keys $\mathsf{sk}_1, \mathsf{sk}_2$ are used with the same pair of nonces k_1, k_2, i.e.,:

$$\begin{aligned} s_{1,1} &= k_1^{-1}(h_{1,1} + r_1\mathsf{sk}_1) \qquad & s_{1,2} &= k_1^{-1}(h_{1,2} + r_1\mathsf{sk}_2) \\ s_{2,1} &= k_2^{-1}(h_{2,1} + r_2\mathsf{sk}_1) \qquad & s_{2,2} &= k_2^{-1}(h_{2,2} + r_2\mathsf{sk}_2) \end{aligned}$$

Here, no nonce is used twice by the same key, but nonces have been reused across keys. The system thus consists of 4 linearly independent equations and 4 unknowns and is thus uniquely solvable. A solution for sk_2 that can be computed, with Gaussian elimination for example, would be:

$$\mathsf{sk}_2 = \frac{r_1 s_{1,2}(h_{2,2}s_{2,1} - h_{2,1}s_{2,2}) - r_2 s_{2,2}(h_{1,2}s_{1,1} - h_{1,1}s_{1,2})}{r_1 r_2 (s_{1,2}s_{2,1} - s_{1,1}s_{2,2})}.$$

In general, we can think of this problem as follows. An attacker is given a set of signatures $\mathcal{S} = \{(h_1, r_1, s_1, \mathsf{pk}_1), \ldots, (h_n, r_n, s_n, \mathsf{pk}_n)\}$, which can be extracted from the Bitcoin blockchain, for example. Each tuple $(h_i, r_i, s_i, \mathsf{pk}_i) \in S$ corresponds to a signature $(r_i = (Gk_i)_x, s_i = k_i^{-1}(h_i + r\mathsf{sk}))$ where $\mathsf{pk} = G\mathsf{sk}$. The goal of the attacker is to leak as many keys (or nonces) as possible by solving systems of linear equations. To achieve this, an attacker has to identify subsets of solvable systems. They can do so by reducing this problem to graph theory.

For instance, we build an undirected bipartite graph $G = (V_{\mathsf{pk}} \cup V_r, E)$, where $V_{\mathsf{pk}} = \{\mathsf{pk}_i \mid (\cdot,\cdot,\cdot,\mathsf{pk}_i) \in \mathcal{S}\}$, $V_r = \{r_i \mid (\cdot, r_i, \cdot, \cdot) \in \mathcal{S}\}$ and $E = \{\{r_i, pk_i\} \mid (\cdot, r_i, \cdot, \mathsf{pk}_i) \in \mathcal{S}\}$. The graph G consists of two types of nodes, r values r_i and public keys pk_i, each of which corresponds to an unknown (a nonce k_i and a secret key sk_i). An edge $\{r, \mathsf{pk}\}$ in this graph corresponds to a signature, which in turn corresponds to an equation in the system of linear equations that $\mathcal{S}$ constitutes. As a pre-filtering step, we first collect all the r values and public keys that appear at least twice in conjunction, i.e., we collect $F = \{r, \mathsf{pk} \mid |\{(\cdot, r, \cdot, \mathsf{pk}) \in \mathcal{S}\}| > 1\}$. Since this corresponds to the same nonce being used by the same key at least twice, it means that we can leak the used secrets k and sk using Equation (3) and Equation (2) with the appropriate signatures. Additionally, we can leak all the secrets which correspond to the nodes that are reachable by every public key and nonce in F. To understand this, assume we have an r value $r_i \in F$, which means that we can leak the nonce k_i as described. Now assume that there is a node $\mathsf{pk}_j \in V_{\mathsf{pk}}$ such that $\{r_i, \mathsf{pk}_j\} \in E$, which implies the existence of the equation $s_j = k_i^{-1}(h_j + r\mathsf{sk}_j)$. Since we know k_i, we can leak sk_j with $\mathsf{sk}_j = \frac{s_j k_i - h_j}{r}$. The same is analogously true if we assume a public key $\mathsf{pk}_i \in F$ and an r value $r_j \in V_r$ such that $\{r_j, \mathsf{pk}_i\} \in E$. By applying this argument inductively, it becomes evident that we can leak the secrets associated with all nodes that are reachable from every $r_i \in F$ and every $\mathsf{pk}_i \in F$.

In the next step, we need to identify the nodes and edges which can be mapped to a solvable system of linearly independent equations. This can be achieved by finding non-trivial *cycles* in G, i.e., distinct nodes $r_0, \mathsf{pk}_0, \ldots, r_n, \mathsf{pk}_n$ for $n > 0$ such that $\{r_i, \mathsf{pk}_i\} \in E$ and $\{\mathsf{pk}_i, r_{i+1 \bmod n}\} \in E$ for $0 \leq i \leq n$. Such a cycle contains $2(n+1)$ nodes, i.e., unknowns, and $2(n+1)$ edges, i.e., equations, and thus directly implies the existence of a solvable system of linear equations. Hence, for all such cycles we can leak the corresponding secrets, and, as before, we can also leak the secrets of the reachable nodes. The output of this whole process is two sets $V'_{\mathsf{pk}} \subseteq V_{\mathsf{pk}}$ and $V'_r \subseteq V_r$, which are the public keys and r values for which we have leaked the secret keys and nonces, respectively. If we remove the nodes in $V'_{\mathsf{pk}} \cup V'_r$ and their edges from G, the resulting graph should not contain any non-trivial cycles. This means that no more secrets can be leaked and hence V'_{pk} and V'_r are optimal with respect to their size.

There is, however, a little twist to the methodology we described here. We consider two signatures (r_1, s_1) and (r_2, s_2) a case of nonce reuse if the r values coincide, i.e., if $r_1 = r_2$. This is not strictly true, as the r value is only the x-coordinate of Gk. Since elliptic curves are based on a Weierstrass equation of the form $y^2 = x^3 + bx + a$, there are always two nonces k which lead to the same r value[4]. In particular, if we have $Gk = (x, y)$, then we have $G(-k) = (x, -y)$. This means that if the r values coincide, we need to take into account that one nonce might be the additive inverse of the other rather than being equal. To respect this, we must consider for every signature (r, s) the signature $(r, -s)$ as well, which is the signature that is yielded by negating k. For each such combination we have to solve the system of linear equations and check if the

[4] Recall that $r \neq 0$ (cf. Section 4.1).

r value	Occurrences
`0x00000000000000000000003b78ce563f89a0ed9414f5aa28ad0d96d6795f9c63`	2,276,718
`0x00006fcf15e8d272d1a995af6fcc9d6c0c2f4c0b6b0525142e8af866dd8dad4b`	7,895
`0x1206589b08a84cb090431daa4f8d18934a20c8fa52ad534c5ba0abb3232be1d9`	265
`0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798`	251
`0x2ef0d2ae4c49c37703ba16a3126e27763e124ff3338fb93577ed7bd79ed0d19e`	91
`0x06cce13d7911baa7856dec8c6358aaa1fb119b5a77d0e4d75d5a61acae05fcfb`	83
`0xd47ce4c025c35ec440bc81d99834a624875161a26bf56ef7fdc0f5d52f843ad1`	76
`0x281d3da7518241cd8ee30cd57ae3173a1bd9ee5e3b02a46ba30f25cd5b4c6aa8`	68
`0x8216f63d28f4dc0b6909a330d2af09b93df9dd3b853958c4d203d530328d8ed1`	64
`0x5d4eb477760cf19ff00fcb4bab0856de9e1ce7764d829a71d379367684712be4`	52

Table 1. The 10 most frequent r values and their number of occurrences.

returned solutions are correct to leak the correct keys and nonces. This can be done by double-checking that each leaked secret key sk corresponds to the given public key pk, which can be done by verifying the equality $G\mathsf{sk} = \mathsf{pk}$.

4.4 Results

We will now outline our results regarding nonce reuse in the Bitcoin blockchain. To achieve this, we downloaded a copy of the Bitcoin blockchain up until block 506071, which was mined on 2018-01-25 16:04:14 UTC. We parsed all inputs from all P2PKH transactions to extract their ECDSA signatures.

In total, we extracted 647,110,920 signatures and we found 1,068 distinct r values appearing at least twice and used by 4,433 keys. In total, these duplicate r values make up for 2,290,850 (0.35%) of all r values. In Table 1, we show the top 10 most frequent duplicate r values along with their number of appearances. The most frequent duplicate r value appears 2,276,671 times, which makes up 99.38% of all duplicate occurrences. This r value is special, as it is extraordinary small, given that its 90 most significant bits are all 0. Additionally, the corresponding nonce k for this r value is $k = \frac{1}{2} \mod n$. As this is unlikely to be a coincidence, it is believed that the designers of the secp256k1 curve chose the generator point G based on these values. It is also believed that this r value is used on purpose by peers to save transaction fees. Bitcoin uses the DER encoding to serialize signatures, which can compress the leading bits of this r value, which reduces the transaction size and leads to smaller transaction fees. If peers use this nonce only for the "last" transaction of an address, i.e., the final transaction which removes all funds, then this should be secure as long as the transaction is marked as non-replaceable. But since this transaction still leaks the secret key of the address, the peer needs to make sure that they will never use the address again. Our analysis revealed that this r value was primarily used in two time periods. The first block which contains this value is block 364,767 and the last one is block 477,411. In total, we identified 1,550 blocks which contain this r value. We found that the r value was used excessively in 2 time periods, which is depicted in Figure 1. We can see that between block 365,000 and block 366,000 and between block 374,000 and block 375,000, the value is used roughly 1 million times each, which makes up almost all of its appearances.

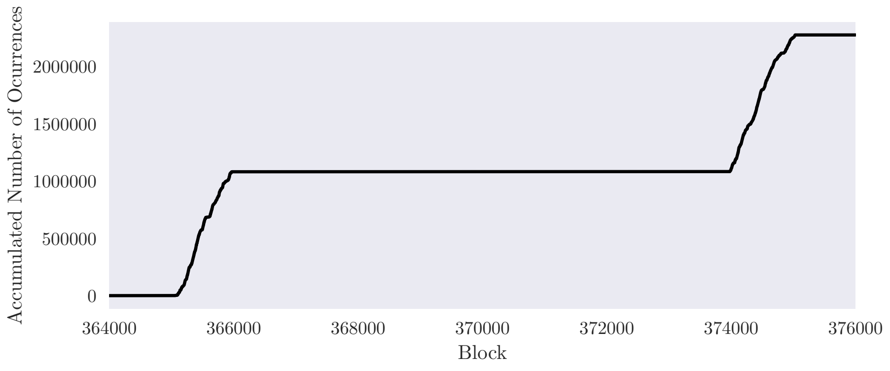

Fig. 1. Number of occurrences of the most prominent duplicate r value over time.

Inspecting the other duplicate r values a bit more closely reveals further interesting cases. The second most used r value also has 16 leading 0 bits, which is also an indication that the corresponding nonce was not chosen randomly. The fourth most used r value corresponds to the nonce $k = 1$, which is an indication of either a broken random number generator or a hand-crafted transaction where the nonce was not randomized and the creator simply took the x-coordinate of G. Another r value we found was using the nonce $k = 12345678$, which is also an indication of an ad-hoc generated transaction using a fixed nonce rather than a secure random one. Similarly, we found two other r values where the corresponding nonces where suspiciously small, i.e., in one case the nonce was $k =$ `0x80001fff` and in another case the nonce also had 74 leading 0 bits. In another case the nonce was $k = \sum_{i=0}^{32} 16^i$, i.e., `0x0101...01` in hexadecimal notation, which looks like a pattern that a human would produce.

4.5 Measuring the Impact of Weak Nonces

We will now assess how much damage an attacker could have caused by using the previously described methodology for leaking keys and nonces. To do this, we put ourselves in the position of an attacker who monitors the transactions of the blockchain. That is, we use our copy of the blockchain to create an ordered sequence of signatures $[(\Delta_1, h_1, r_1, s_1, \mathsf{pk}_1), \ldots, (\Delta_n, h_n, r_n, s_n, \mathsf{pk}_n)]$ where Δ_i is a block number such that $\Delta_i \leq \Delta_j$ for $i \leq j$ and the remaining elements are the components of a signature found in a transaction of block Δ_i. We then process these entries in order as follows. We add each signature $s_i = k^{-1}(h_i + r_i)$ for a public key pk_i in block Δ_i to a database, which allows us to quickly identify duplicate r values as well as their signatures. Each identified duplicate r value r_i is then added to the graph G along with the used public key pk_i. However, before adding these 2 nodes to the graph, we make a few checks. First, we check if we have leaked both k_i and sk_i, in which case we can completely disregard both, as adding them will not lead to new leaks. Second, we check if G already contains

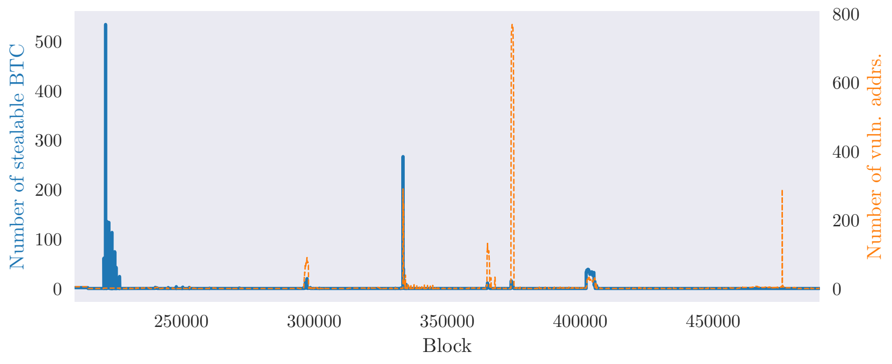

Fig. 2. Number of stealable Bitcoins + number of vulnerable Bitcoin addresses attributed to ECDSA nonce reuse over time.

the edge $\{r_i, \mathsf{pk}_i\}$, in which case we can leak both k_i and sk_i. Third, we check if we have already leaked either k_i or sk_i, in which case we can then leak sk_i or k_i, respectively. In the last two cases, we can also leak the secrets corresponding to all the nodes reachable from both r_i and pk_i as discussed previously. Only if none of these three conditions apply, we add the edge $\{r_i, \mathsf{pk}_i\}$ to G. After processing all signatures of a block, we look for cycles in G to identify solvable systems of linear equations in order to leak secrets as outlined previously. Whenever we find a new leak, we make sure that we remove the corresponding signatures from the database and that we remove the corresponding nodes and their edges from G, as we will otherwise redundantly reconsider the same r values and cycles.

Using this methodology, we managed to leak 892 out of the 1,550 possible nonces (57.55%) and 2,537 out of the 4,433 secret keys that were used in conjunction with these nonces (57.23%). In total, this gives us theoretical control over the balances of 5,074 addresses, i.e., two addresses per key. During this whole operation we identified 23 cycles in the graph and the longest cycle consisted of 12 nodes, which represents a system of 12 linear equations and 12 unknowns (6 nonces + 6 secret keys). The final shape of G did not contain any more cycles, which means that we have leaked the maximum number of secrets.

Figure 2 depicts the number of Bitcoins that an attacker could have stolen at any point in time, i.e., the block height, as well as the number of *vulnerable* addresses at each moment in time. We consider an address at a certain block vulnerable if we have leaked the key of the address and if it held a balance at that block. There are a few notable spikes for both the number of stealable Bitcoins as well as the number of vulnerable addresses. The first significant spike occurs roughly between block 221,000 and block 227,000, where the peak stealable balance is 533.82 BTC. Interestingly, there was only one vulnerable address during this spike. The next spike occurs roughly between block 296,000 and block 298,000 with a peak stealable balance of 20 BTC, which was stealable for a times-

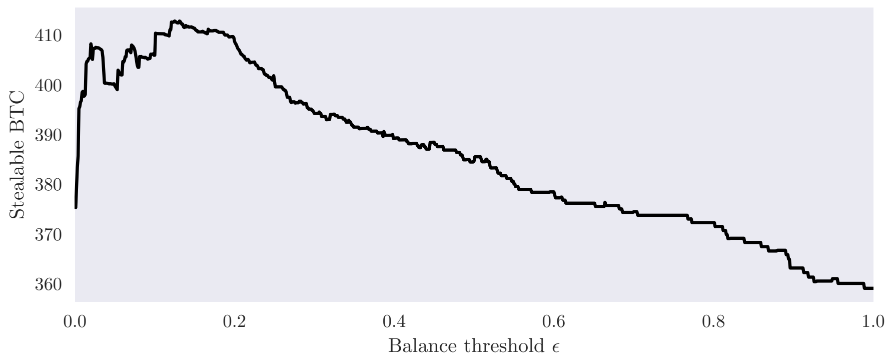

Fig. 3. Number of Bitcoins an attacker could have stolen based on a balance threshold.

pan of 3 blocks from block 297283 until block 297285. From block 297,261 up to block 297,304 there were 90 vulnerable addresses, which is also the maximum number of vulnerable addresses of the spike. The next spike is slightly shorter and happens at around block 333,300 and lasts roughly until block 333,600. During this timespan, an attacker could have stolen up to 266.73 BTC at block 333,387 and there were 290 peak vulnerable addresses at block 333,393. This is followed by two similarly long-lasting spikes between blocks 365,000 and 366,000 and blocks 374,000 and 375,000. In the former case, 11.21 BTC was stealable and there were 131 vulnerable addresses at some point, and in the latter case 15.41 BTC was stealable and there were 769 vulnerable addresses from block 374,386 to 374,386. This is also the largest number of addresses that were vulnerable at a time over the whole timespan. Finally, while the number of vulnerable addresses suddenly jumps to 289 at block 475,963, there are only 0.0064 BTC stealable at the peak. At the current state of our copy of the blockchain, there are 5 vulnerable addresses holding an accumulated balance of 4002 satoshis, i.e., 0.00004002 BTC, which is unlikely to be stolen given current transaction fees.

To assess how much an attacker could have stolen over time, we consider two scenarios. First, we assume an attacker which steals the peak balance of each address over time. That is, we take the sum of the peak balances of each address, which gives a total of 1021.58 BTC. Here, we implicitly assume that the owner notices the fraud and therefore we ignore all future funds. However, this attack model requires an attacker to know the peak balance in advance, which is unrealistic. Therefore, we consider a second more realistic attacking scenario in which an attacker defines a balance threshold ϵ. In this setting, an attacker only steals a balance if it is equal to or larger than ϵ and we assume again that we can only steal once from an address. Figure 3 plots the number of Bitcoins that an attacker could have stolen in this scenario depending on ϵ. We let ϵ range between 0 and 1 BTC with 0.001 increments. The optimal balance threshold according to the plotted function is $\epsilon = 0.125$, which an attacker could have used to steal 412.80 BTC. Note that even though one address alone had a balance of 533.82 BTC at some point, it does not mean that an attacker in this setting can steal it completely. This is because we assume that we can steal only once from an address once its balance surpasses the balance threshold ϵ, after which we conservatively assume that the owner of the address becomes aware of the problem. While this means that choosing a large ϵ such as $\epsilon = 500$ BTC would yield a larger profit for the attacker, we believe that it is not an optimal choice. Given the current value of Bitcoin, we believe that it is unrealistic for a single individual to hold such a large balance. Additionally, if we assume that there are multiple competing attackers, then we also have to take this into consideration when choosing ϵ. We therefore let ϵ range between 0 and 1 BTC as we believe that this is a good compromise between what is currently practical and what is optimal in theory. After said optimum, the number of BTC starts to decrease steeply, and for $\epsilon = 1$, there are 359.04 stealable Bitcoins, i.e., 13.02% less than in the optimal case. Similar to our previous OSINT analysis in Section 3.2, we also ignored transaction fees here due to their negligible impact. Additionally, we also did not consider blocking transactions in this case, as an attacker monitoring transactions can create stealing transactions as soon as possible.

4.6 Identifying Past Attacks

Given that the phenomenon of ECDSA nonce reuse is a known problem, we now try to assess if it has been used by attackers in the past to steal Bitcoins. To do this, we tried to identify for each of the 7 spikes in Figure 2 the moment in time when the number of stealable Bitcoins suddenly dropped. Then we tried to find transactions, which were created during that time and whose outputs referenced inputs of many vulnerable addresses. In the case of the first spike, it is hard to argue whether it was used by an attacker as only 1 address was vulnerable in this timespan. However, we identified several cases where the balance of the address suddenly dropped by over 99.99%, which one could argue is an incident where Bitcoins have been stolen. In the second, third, sixth and seventh spike we found cases where the number of stealable Bitcoins decreased suddenly and we identified in all cases a single transactions referencing all the responsible vulnerable addresses, which makes us believe that Bitcoins were stolen. In the case of the last spike, however, only 0.00064 BTC were stolen.

Regarding the fourth and fifth spike, we did not observe a similar suspicious drop regarding the number of Bitcoins, but only regarding the number of vulnerable addresses. To see the difference, consider Figure 4, which shows and compares a zoomed in view of the second and the fifth spike. In the former, we can see a sudden drop in the number of stealable Bitcoins, i.e., while there are 7.49 stealable BTC at block 297,304, there are only 0.2 BTC stealable at block 297,305. We identified a single transaction which transferred all the stealable Bitcoins, indicating a theft. The fact that the number of vulnerable addresses did not decrease to 0 at the same time can be explained by various reasons. For instance, it could be possible that the attacker was not aware of the remaining vulnerable addresses. Or, it could be the case that the attacker used a balance

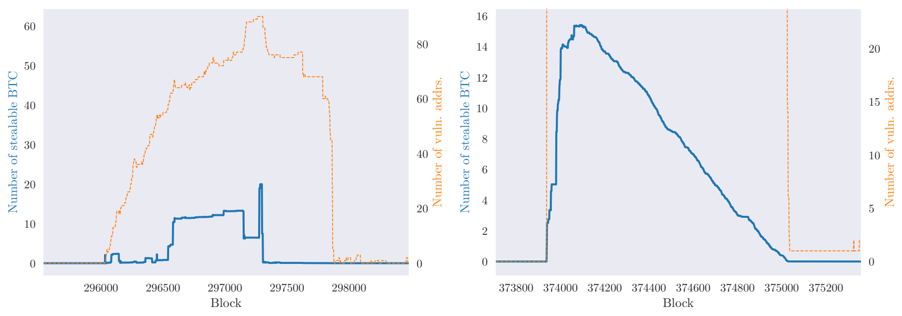

Fig. 4. Comparison of a spike where Bitcoins might have been stolen due to a sudden drop in stealable Bitcoins (left) and a case where we see a smooth decrease indicating that no coins might have been stolen (right).

threshold and determined that the remaining addresses are not worth stealing from based on this threshold, because as we can see, the 0.2 BTC are shared among 86 vulnerable addresses. In the second spike in Figure 4, we observe a smooth and monotone decrease over time regarding the number of stealable Bitcoins and then a sudden decrease of the number of vulnerable addresses at the same time the stealable BTC drop. This phenomenon could be explained by the fact that all the addresses belong to the same individual and that at the end all the so-called *change addresses* are emptied by the wallet. Change addresses are addresses which are used to accumulate leftover transaction outputs. For example, if an address A wants to send 1 BTC to an address B using a single output, which is worth 5 BTC, then the resulting transaction will create two outputs, one that is worth 1 BTC and can be spent by address B and one that is worth 4 BTC and can be spent by a change address that belongs to the owner of A. The final transaction of the wallet will then use all accumulated outputs of the change addresses, which could be an explanation for the sudden drop.

5 Discussion

In this section, we will consider the ethical aspects of our work and describe how the problem of key leakage of cryptocurrencies can be tackled.

5.1 Ethical Considerations

Given that we systematically describe how attackers can steal Bitcoins abusing leaked keys, we have to address the ethical aspects that come along with such a work. On the one hand, we believe that raising awareness of these attack vectors is fundamental and important to improve the security of the cryptocurrency ecosystem. On the other hand, one could argue that the amount of detail we put into outlining these methods is not beneficial as it allows for easy reproducibility by attackers. Yet we believe that this is the right way to tackle this problem as "security by obscurity" has proven in the past to be an insufficient means in the area of security. We also note that ECDSA nonce reuse is known to be a problem and it has been reported in forum posts that this phenomenon has occurred in the Bitcoin blockchain [5,6,7]. However, as we have shown in Section 4.4, this apparently known problem still regularly occurs and is abused by attackers, with the latest case of nonce reuse appearing in a block mined on 2017-07-15. This constant recurrence leads us to believe that it will happen again, unless we emphasize this problem better, which is why we outline the attack in detail.

Another ethical aspect of dealing with attacks on cryptocurrencies is that a responsible disclosure process in terms of notifying the victims is not trivial. A fundamental downside here is that Bitcoin itself is decentralized by design and intends to ensure the anonymity (or pseudonymity) of the peers. This means that (i) we have no dedicated point of contact, which we could inform about our findings and (ii) we cannot reach out to the legitimate owners of the vulnerable addresses. We tried to handle this problem as responsibly as possible and refrain from disclosing problematic addresses and/or transactions. For example, we did not mention any vulnerable addresses or URLs to pastes containing them, as we cannot be sure that the owners of those addresses are aware of the vulnerability. While an attacker can reproduce our methodology to find any future vulnerable addresses using Pastebin, it should not be easily possible to find the addresses we have discovered, since the Pastebin feed only lists the most current 250 pastes. While Pastebin can be searched using standard search engines like Google, it should be very hard to discover the pastes we have found, since search engines only offer a keyword-based search, rather than a regex-based search which would be required to find the addresses. In our ECDSA case study, we also did not mention any vulnerable addresses. However, an attacker can fully reproduce our results here, as the Bitcoin blockchain contains all the necessary historical information. Therefore, not mentioning vulnerable addresses is not as effective as in the case of our OSINT case study. Yet we also do not see a reason to do so, as one could argue that this makes it too easy for an attacker.

5.2 Countermeasures

Explicitly leaking keys is not strictly a technical problem, as users seemingly publish private information without knowing the consequences of doing so. However, there are some technical solutions that could be applied on OSINT platforms. For example, Pastebin could include a check in their logic, which scans pastes for secrets such as Bitcoin secret keys encoded in the WIF format. In fact, they could provide immediate feedback to users about the security implications of pasting such content. We have therefore contacted Pastebin with a detailed description of our work, proposing to adopt such a methodology.

To avoid ECDSA nonce reuse, there are a few solutions that can be applied. One such solution proposed by RFC 6979 [19] is to choose the nonce k deterministically based on the message m and the key sk. As inputs differ, this scheme provides unique nonces and hardens against nonce reuse. However, since this solution is backwards-compatible with the existing ECDSA scheme, it also means that peers do not *have to* follow this proposal. In particular, one cannot verify that a signature has been created with the deterministic nonce choice as proposed by RFC 6979. Another way of dealing with this problem is to incorporate a duplicate nonce check into the Bitcoin protocol. For example, a check for duplicate r values could be incorporated into the transaction verification process. Each peer verifies each transaction of a block, which includes verifying the signature and other sanity checks. Here, the protocol could also support a check for duplicate r values, i.e., checking, for each r value of each signature, if it already occurs in the blockchain. From a performance perspective, a Bloom filter could help to scale this process. The more peers follow this, the less likely it will become that a transaction containing a duplicate r value will be added to the blockchain. However, an attacker monitoring the mempool instead of the blockchain might still be able to observe transactions containing duplicate r values. Therefore, one would need to additionally adapt the network rules such that a new rule is added, which discourages the distribution of transactions which contain duplicate nonces. If such a transaction reaches a peer which follows this new set of rules, the duplicate r value will be detected and the transaction will not be relayed further. Additionally, the peer sending the transaction should be notified with an error message about the problem to create awareness. The more peers follow this new set of rules, the less likely it becomes that transactions containing duplicate r values are distributed among the network. In total, we believe that the adoption of all proposals, i.e., deterministic ECDSA and adapting the network rules as well as the transaction verification process, are sufficient means to eliminate nonce reuse from cryptocurrencies.

6 Related Work

In this section we discuss other work in the areas of OSINT, Bitcoin key leakage and ECDSA nonce reuse, and how they relate to our work.

OSINT has been applied before to expose or harvest privacy-related information. Matic et al. [16] performed a study in which they monitored the Pastebin feed between late 2011 and early 2012 to develop a framework for detecting sensitive information in pastes. They discovered almost 200,000 compromised accounts of several websites as well as lists of compromised servers or leaked database dumps. In a slightly different vein, Sabottke et al. [20] design a Twitter-based exploit detector that can predict vulnerabilities such as code execution or Denial-of-Service attacks solely based on tweets. Similarly, Zhu et al. [24] show how they can use academic security literature as OSINT to automatically engineer features for malware detection. While all of these works show the potential of OSINT, they are only remotely related to our work as our use case is different.

In terms of leaking Bitcoin secrets to steal money, there have been a few other papers targeting this problem. Vasek et al. [22] have outlined how one can attack passphrase-based wallets (*brain wallets*). The authors developed a tool called *Brainflayer*, which uses brute force and a dictionary to generate weak passphrases, which would have allowed an attacker to steal Bitcoins worth $100,000 at that time. This approach is similar to ours in the sense that an attacker exploits the fact that users treat sensitive information wrongly, i.e., passphrases in the case of Brainflayer and secret keys or nonces in our case. In contrast to searching for weak passphrases, we harvest OSINT and cryptographic primitives. More related are works by Castellucci et al. [10] and Valsorda [21], which both consider ECDSA nonce reuse with respect to Bitcoin. However, both only cover the basic case, where a nonce is used in conjunction with the same key twice. We generalize this concept to systems of linear equations and systematically outline how an attacker can use a graph-based approach to leak secrets.

The general problem of nonce reuse in respect to ECDSA (or the closely related DSA scheme) has been studied in other contexts. A notable incident occurred in 2010, when it was discovered that Sony reused the same nonce to sign software for the PlayStation 3 game console [8]. Furthermore, Heninger et al. [14] studied the impact of weak keys and nonce reuse in the case of TLS and SSH servers. The authors collected over 9 million signatures and found that 0.05% of these signatures contained the same r value as at least one other signature. Additionally, the authors used a subset of those signatures where a key and a nonce appear in conjunction at least twice to leak 281 secret keys. Apart from studying a different use case, i.e., Bitcoin, our work is different here in that we systematically outline how an attacker can leak keys, which goes beyond the simple case where the same key and nonce is used more than once.

7 Conclusion

We have studied the problem of implicit and explicit key leakage in the context of cryptocurrencies, which shows how an attacker can leverage OSINT or duplicate nonces to leak secret keys. Our case studies have shown the practical relevance of these issues. An attacker monitoring Pastebin or scanning transactions for nonce reuse could have stolen up to 22.40 BTC and 412.80 BTC, respectively. Our work emphasizes aspects that are important for both the users and the developers of cryptocurrencies. For instance, our Pastebin case study shows the importance of making users aware of how to deal with cryptocurrency secrets. Our results regarding ECDSA show that nonce reuse is a recurring problem and highlight the benefits of incorporating countermeasures on the protocol level. In the case that cryptocurrencies become even more popular, it will become more lucrative for miscreants to perform key leakage attacks similar to the ones we described here. This highlights the importance of our research, which apart from creating awareness of the problem, also can foster future research on the topic of explicit and implicit key leakage in the context of cryptocurrencies.

Acknowledgements. This work was supported by the European Union's Horizon 2020 research and innovation programme, RAMSES, under grant agreement No. 700326.

References

1. https://coinmarketcap.com/currencies/bitcoin/, Accessed: 2018-03-27
2. https://coinmarketcap.com/currencies/ethereum/, Accessed: 2018-03-27
3. https://pastebin.com, Accessed: 2018-03-27
4. https://bitcoin.org/en/alert/2013-08-11-android, Accessed: 2018-03-27
5. https://bitcointalk.org/index.php?topic=581411.0, Accessed: 2018-03-27
6. https://bitcointalk.org/index.php?topic=1118704.0, Accessed: 2018-03-27
7. https://bitcointalk.org/index.php?topic=1431060.0, Accessed: 2018-03-27
8. https://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf (2010), Accessed: 2018-03-27
9. Back, A.: Hashcash - A Denial of Service Counter-Measure (2002)
10. Castellucci, R., Valsorda, F.: Stealing Bitcoin with Math. https://news.webamooz.com/wp-content/uploads/bot/offsecmag/151.pdf, Accessed: 2018-03-27
11. Decker, C., Wattenhofer, R.: Bitcoin Transaction Malleability and MtGox. In: Proc. of the European Symposium on Research in Computer Security (ESORICS) (2014)
12. Diffie, W., Hellman, M.: New Directions in Cryptography. IEEE Transactions on Information Theory (1976)
13. Eskandari, S., Barrera, D., Stobert, E., Clark, J.: A First Look at the Usability of Bitcoin Key Management. In: Proc. of the Workshop on Usable Security (USEC) (2015)
14. Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In: Proc. of the USENIX Security Symposium (USENIX Security) (2012)
15. Koblitz, N.: Elliptic Curve Cryptosystems. Mathematics of Computation (1987)
16. Matic, S., Fattori, A., Bruschi, D., Cavallaro, L.: Peering into the Muddy Waters of Pastebin. ERCIM News (2012)
17. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)
18. Naware, A.M.: Bitcoins, Its Advantages and Security Threats. International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) (2016)
19. Pornin, T.: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). https://rfc-editor.org/rfc/rfc6979.txt (2013)
20. Sabottke, C., Suciu, O., Dumitras, T.: Vulnerability Disclosure in the Age of Social Media: Exploiting Twitter for Predicting Real-World Exploits. In: Proc. of the Usenix Security Symposium (USENIX Security) (2015)
21. Valsorda, F.: Exploiting ECDSA Failures in the Bitcoin Blockchain. In: Proc. of Hack In The Box (HITB) (2014)
22. Vasek, M., Bonneau, J., Castellucci, R., Keith, C., Moore, T.: The Bitcoin Brain Drain: Examining the Use and Abuse of Bitcoin Brain Wallets. In: Proc. of the the International Conference on Financial Cryptography (2016)
23. Wood, G.: Ethereum: A next-generation smart contract and decentralized application platform. https://ethereum.github.io/yellowpaper/paper.pdf (2018), Accessed: 2018-03-27
24. Zhu, Z., Dumitras, T.: FeatureSmith: Automatically Engineering Features for Malware Detection by Mining the Security Literature. In: Proc. of the Conference on Computer and Communications Security (CCS) (2016)