

EXPLOITING ECDSA FAILURES IN THE BITCOIN BLOCKCHAIN

Filippo Valsorda

HITB2014KUL



FILIPPO VALSORDA

CloudFlare security team

@FiloSottile

I mess with cryptography.
And open source.

filippo.io

BUT YOU PROBABLY
KNOW ME FOR THIS

HTTPS://FILIPPO.IO/HEARTBLEED

Heartbleed test

[FAQ/status](#)



If there are problems, head to the [FAQ](#)

Results are now cached globally for 1 hour.

Enter a URL or a hostname to test the server for CVE-2014-0160.

Go!

Advanced (might cause false results): ignore certificates

You can specify a port like this `example.com:4433`. 443 by default.

Go [here](#) for all your Heartbleed information needs.

If you want me to fix Heartbleed for you, write some Go or design some crypto, I'm a freelancer (for now?), so get in contact: [click here!](#) And if you want to [donate](#) something, I've put a couple of buttons [here](#).

BITCOIN

A WALLET

Public key +
Private key

THE ADDRESS: HASH (PUBLIC KEY)

1DY5YvRxSwomrK7nELDZzAidQQ6ktjRR9A

A TRANSACTION

A signed statement,
published to the world
and recorded in the *blockchain*

“THIS MONEY I CAN SPEND, CAN NOW BE SPENT BY Y”

A: This money I can spend, can now be spent by X

...: This money I can spend, can now be spent by ...

...: This money I can spend, can now be spent by ...

...: This money I can spend, can now be spent by ...

X: This money I can spend, can now be spent by Y

...: This money I can spend, can now be spent by ...

...: This money I can spend, can now be spent by ...

Y has this money to
spend

Signed with A's private key

A: This money I can spend, can now be spent by X

HASH OF X'S PUBLIC KEY

ACTUALLY

OP_DUP OP_HASH160

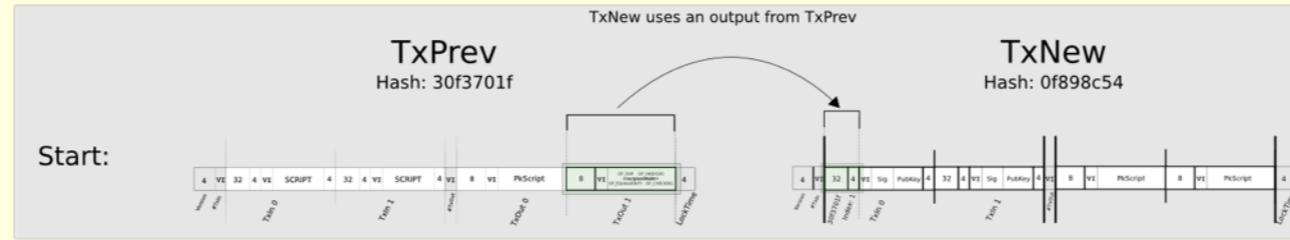
<pubKeyHash>

OP_EQUALVERIFY

OP_CHECKSIG

<sig> <pubKey>

Transaction Verification Steps: OP_CHECKSIG (SIGHASH_ALL only)



Prepare: Execute TxIn.sigScript to get Sig and Key onto stack, execute TxOut.PkScript up to OP_CHECKSIG

Step 1: Pop public key and signature off the stack: `pubKeyStr = stack.pop(), sigStr = stack.pop()`

Step 2: From TxPrev.PkScript, create subscript from last OP_CODESEPARATOR to end of script (if no OP_CS, simply copy PkScript)

PrevTx.PkScript:

SCRIPT_PART1	OP_CODESEPARATOR	SCRIPT_PART2	OP_CODESEPARATOR	OP_DUP	OP_HASH160	<recipientAddr>	OP_EQUALVERIFY	OP_CHECKSIG	OP_CODE_SEPARATOR	SCRIPT_PART4
--------------	------------------	--------------	------------------	--------	------------	-----------------	----------------	-------------	-------------------	--------------

Subscript:

OP_DUP	OP_HASH160	<recipientAddr>	OP_EQUALVERIFY	OP_CHECKSIG	OP_CODE_SEPARATOR	SCRIPT_PART4
--------	------------	-----------------	----------------	-------------	-------------------	--------------

Step 3: Remove signature from subscript, if present
(Not standard to have a sig in the subscript)

Step 4: Remove OP_CODESEPARATORS from Subscript

OP_DUP	OP_HASH160	<recipientAddr>	OP_EQUALVERIFY	OP_CHECKSIG	SCRIPT_PART4
--------	------------	-----------------	----------------	-------------	--------------

Step 5: Extract hashtype from signature:

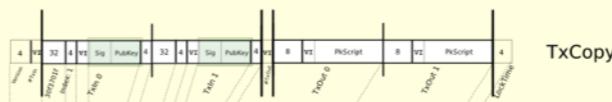
Before: sigStr =

Signature	Hashtype
= 70 bytes (DER)	1

After: sigStr =

= 70 bytes (DER)	hashTypeCode = 4 (expand to 4 bytes) <small>LITTLE-ENDIAN !</small>
------------------	---

Step 6: Copy TxNew to TxCopy (to be modified)



Step 7: Set all TxIn scripts in TxCopy to empty strings
Make sure that the VAR_INT's representing script length are reevaluated to a single 0x00 byte for each TxIn



Step 8: Copy Subscript into the TxIn script you are checking:
Make sure VAR_INT preceding SUBSCRIPT is reevaluated to represent the size of SUBSCRIPT



TxCopy.serialize() → verifyThisStr

ECDSA_CheckSignature(`pubKeyStr`, `sigStr`, sha256²(`verifyThisStr`))

Step 9: Serialize TxCopy, append 4-byte hashTypeCode:

Step 10: Verify signature against string in Step 9,
(hashed string needs to be big-endian)

Repeat all steps for each TxIn object and associated TxOut

EDSA

ELLIPTIC CURVE DIGITAL SIGNATURE ALGORITHM

A EC based signature scheme

AS SEEN IN TLS, DNSSEC, THE PS3...

A SUMMARY

Global: point G on a curve

Private key: a random number d

Public key: $d \times G$

SIGNATURE

$e = \text{hash}(\text{message})$

k = a random number

$(x, y) = k \times G$ $r = x$

Sig: $[r, (e+r*d)/k]$

SEEMS FINE, RIGHT?

Unless...

WHAT HAPPENS IF THAT K IS NOT RANDOM?

IF YOU REUSE K

$$k_1 = k_2$$

$$(x, y) = k \times G \quad r = x$$

$$r_1 = r_2$$

Sig₁: [r, (e₁+r*d)/k]

Sig₂: [r, (e₂+r*d)/k]

IF YOU REUSE K

$$k_1 = k_2$$

$$(x, y) = k \times G \quad r = x$$

$$r_1 = r_2$$

Sig₁: $[r, (e_1 + r * d) / k]$

Sig₂: $[r, (e_2 + r * d) / k]$

IF YOU REUSE K

$$k_1 = k_2$$

$$(x, y) = k \times G \quad r = x$$

$$r_1 = r_2$$

Sig₁: $[r, (e_1 + r * d) / k]$

Sig₂: $[r, (e_2 + r * d) / k]$

IF YOU REUSE K

$$k = (e_1 - e_2) / \\ [(e_1 + r \cdot d) / k - (e_2 + r \cdot d) / k]$$

$$d = \frac{[(e_1 + r \cdot d) / k] * k - e_1}{r}$$

Boom.

ImperialViolet

» SUDDEN DEATH ENTROPY FAILURES (15 Jun 2013)

During the time that the [RSA patent](#) was in force, [DSA](#) was the signature algorithm of choice for any software that didn't want to deal with patent licenses. (Which is why lots of old PGP keys are still DSA.) It has slowly disappeared since the patent expired and it appears that 4096-bit RSA is now the algorithm of choice if you're on the run from the NSA [1]. (And if you're a journalist trying to get a reply: keyid `bda0df3c`.)

But DSA can also be used with elliptic curves in the form of ECDSA and, in that form, it's likely that we'll see it return in the future, at least to some extent. SSH and GPG both support ECDSA now and CAs are starting to offer ECDSA certificates for HTTPS.

Unfortunately, DSA has an important weakness that RSA doesn't: an entropy failure leaks your private key. If you used a machine affected by the [Debian entropy bug](#) then, in that time, messages that you encrypted with RSA can be broken. But if you signed anything with a DSA key, then your private key is compromised.

The randomness in DSA is absolutely critical. Given enough signatures, leaking just a handful bits per signature is sufficient to break it. In the limit, you can make the [make the mistake that Sony did](#) and not understand that the random number needs to be generated for each message and, seemingly, just pick one and code it in. (See [XKCD](#).)

Sony's ECDSA code

```
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
              // guaranteed to be random.
}
```

f311EwERTlqwu

Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices

Nadia Heninger^{†*}

Zakir Durumeric^{‡*}

[†]*University of California, San Diego*

nadiyah@cs.ucsd.edu

Eric Wustrow[‡]

J. Alex Halderman[‡]

[‡]*The University of Michigan*

{zakir, ewust, jhalderm}@umich.edu

Abstract

RSA and DSA can fail catastrophically when used with malfunctioning random number generators, but the extent to which these problems arise in practice has never been comprehensively studied at Internet scale. We perform the largest ever network survey of TLS and SSH servers and present evidence that vulnerable keys are surprisingly widespread. We find that 0.75% of TLS certificates share keys due to insufficient entropy during key generation, and we suspect that another 1.70% come from the same faulty implementations and may be susceptible to compromise. Even more alarmingly, we are able to obtain RSA private keys for 0.50% of TLS hosts and 0.03% of SSH hosts, because their public keys shared nontrivial common factors due to entropy problems, and DSA private keys for 1.03% of SSH hosts, because of insufficient signature randomness. We cluster and investigate the vul-

expect that today’s widely used operating systems and server software generate random numbers securely. In this paper, we test that proposition empirically by examining the public keys in use on the Internet.

The first component of our study is the most comprehensive Internet-wide survey to date of two of the most important cryptographic protocols, TLS and SSH (Section 3.1). By scanning the public IPv4 address space, we collected 5.8 million unique TLS certificates from 12.8 million hosts and 6.2 million unique SSH host keys from 10.2 million hosts. This is 67% more TLS hosts than the latest released EFF SSL Observatory dataset [18]. Our techniques take less than 24 hours to scan the entire address space for listening hosts and less than 96 hours to retrieve keys from them. The results give us a macroscopic perspective of the universe of keys.

Next, we analyze this dataset to find evidence of several

THE BLOCKCHAIN

REMINDER

To spend money:
the public key of the address;
a signature w/ that key

WHEN MONEY IS MOVED A SIGNATURE IS PUBLISHED

AN EASY SEARCH

```
for block in chain:  
    for tx in block:  
        for input in tx:  
            ...
```

A INPUT IS MONEY BEING SPENT IN THE TX

AN EASY SEARCH

Extract r from the signature;
take note of where we found
it in a lookup table;
check if we found it before.

DONE!

If anyone reuses k ,
we will find two equal r .

DONE!

Well... No.
I mean, yes, but there are
100M inputs in the blockchain.

OUT OF MEMORY! :(

A SMARTER SEARCH

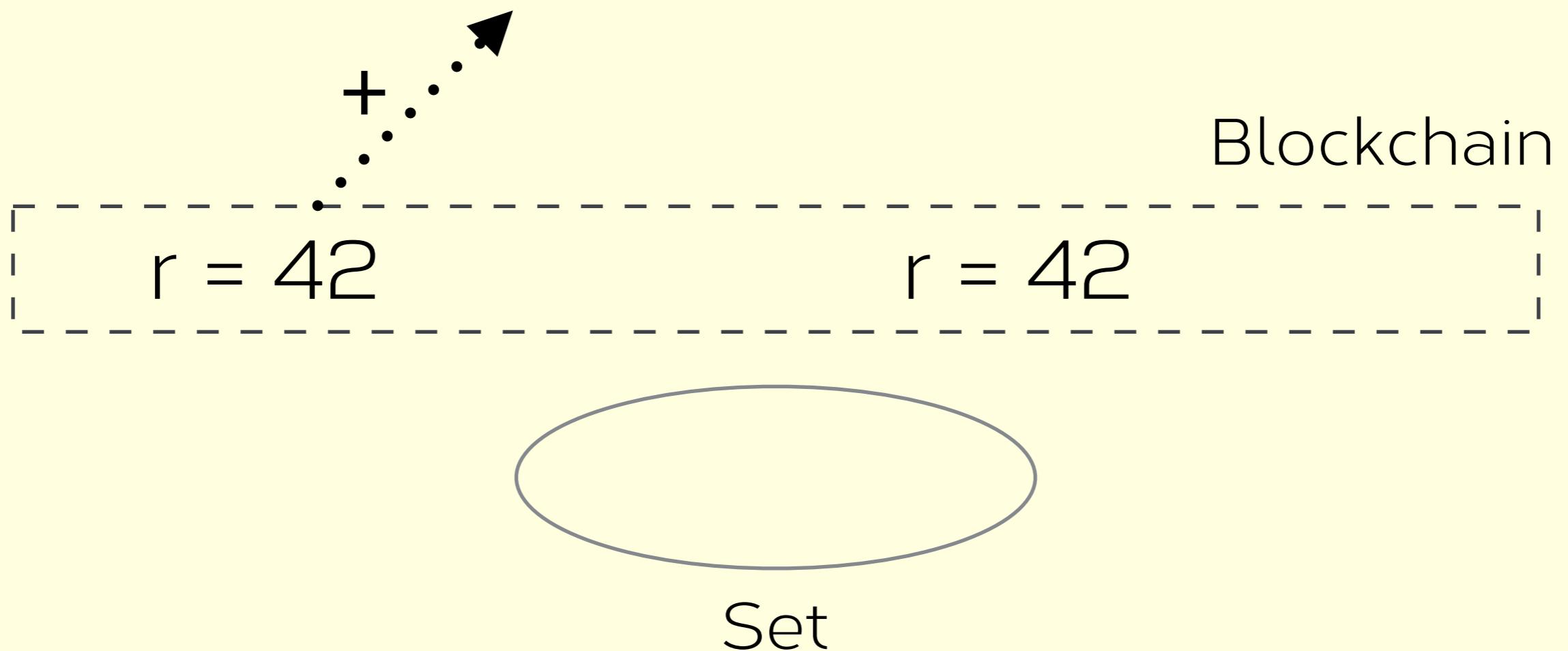
First pass: filter the possible r .

Add to a Bloom filter,
if present add to a set.

Second pass: if r present in
the set, export sig and pubkey.

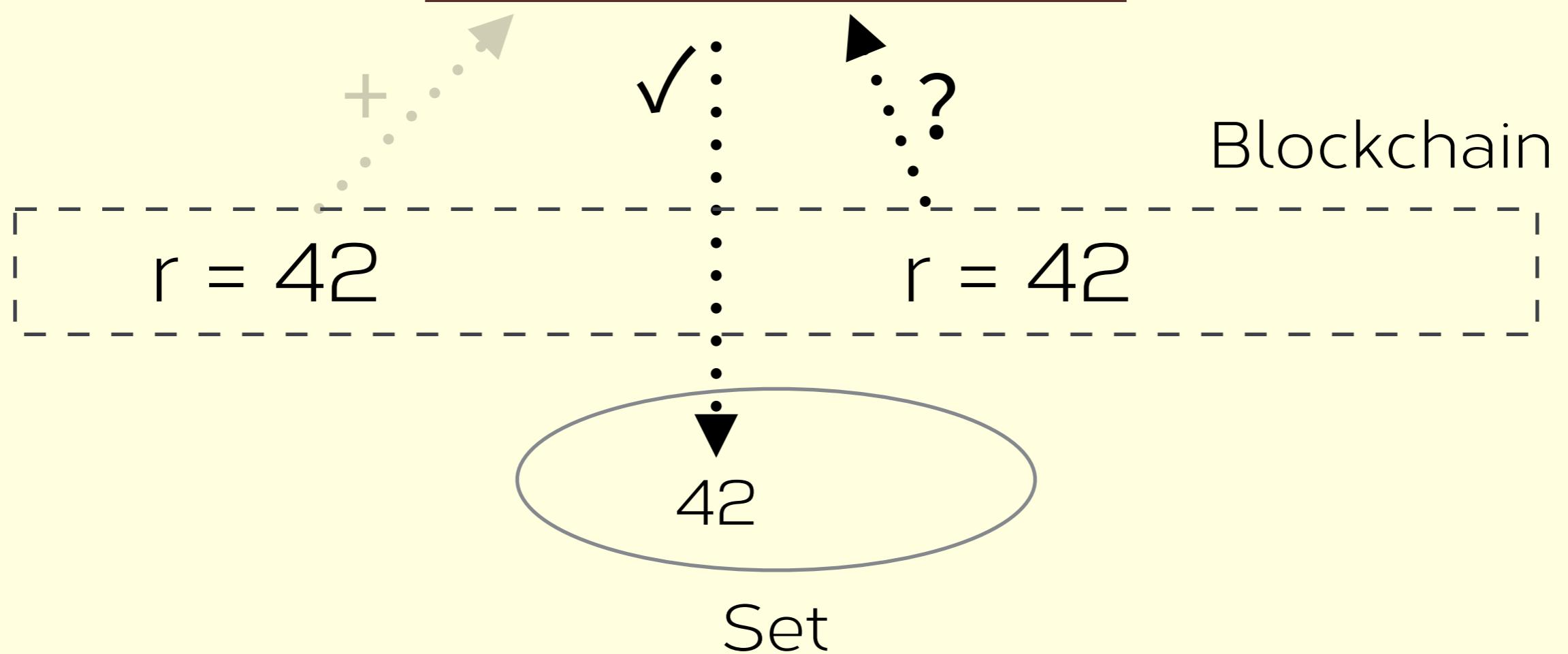
A SMARTER SEARCH

BLOOM FILTER



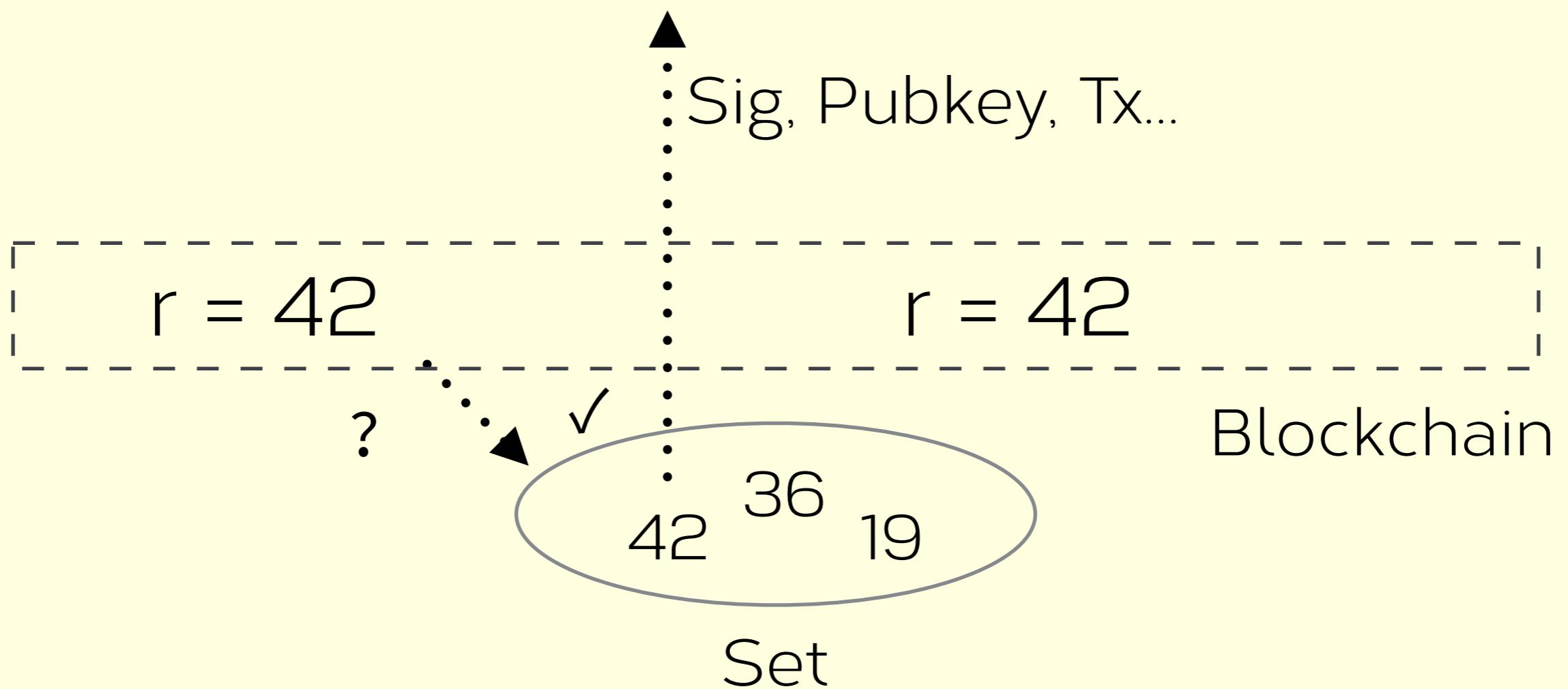
A SMARTER SEARCH

BLOOM FILTER



A SMARTER SEARCH

FINAL LIST



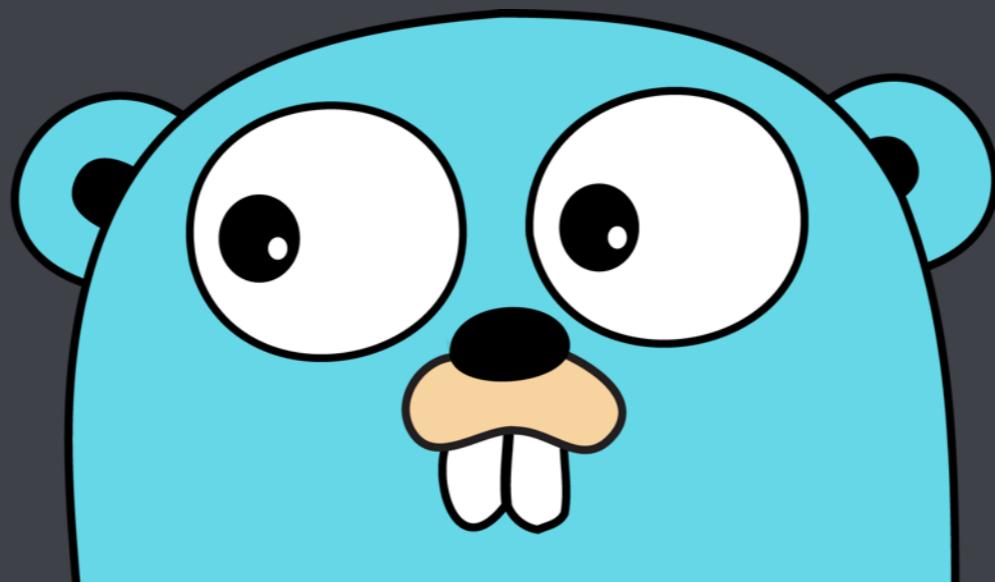
FINALLY

Group the list by (r, pubkey)
and recover d
from pairs of signatures!

BLOCKCHAINR

A ready to use tool

[GITHUB.COM/FILOSOTTILE/BLOCKCHAINR](https://github.com/filosottile/blockchainr)



RESULTS

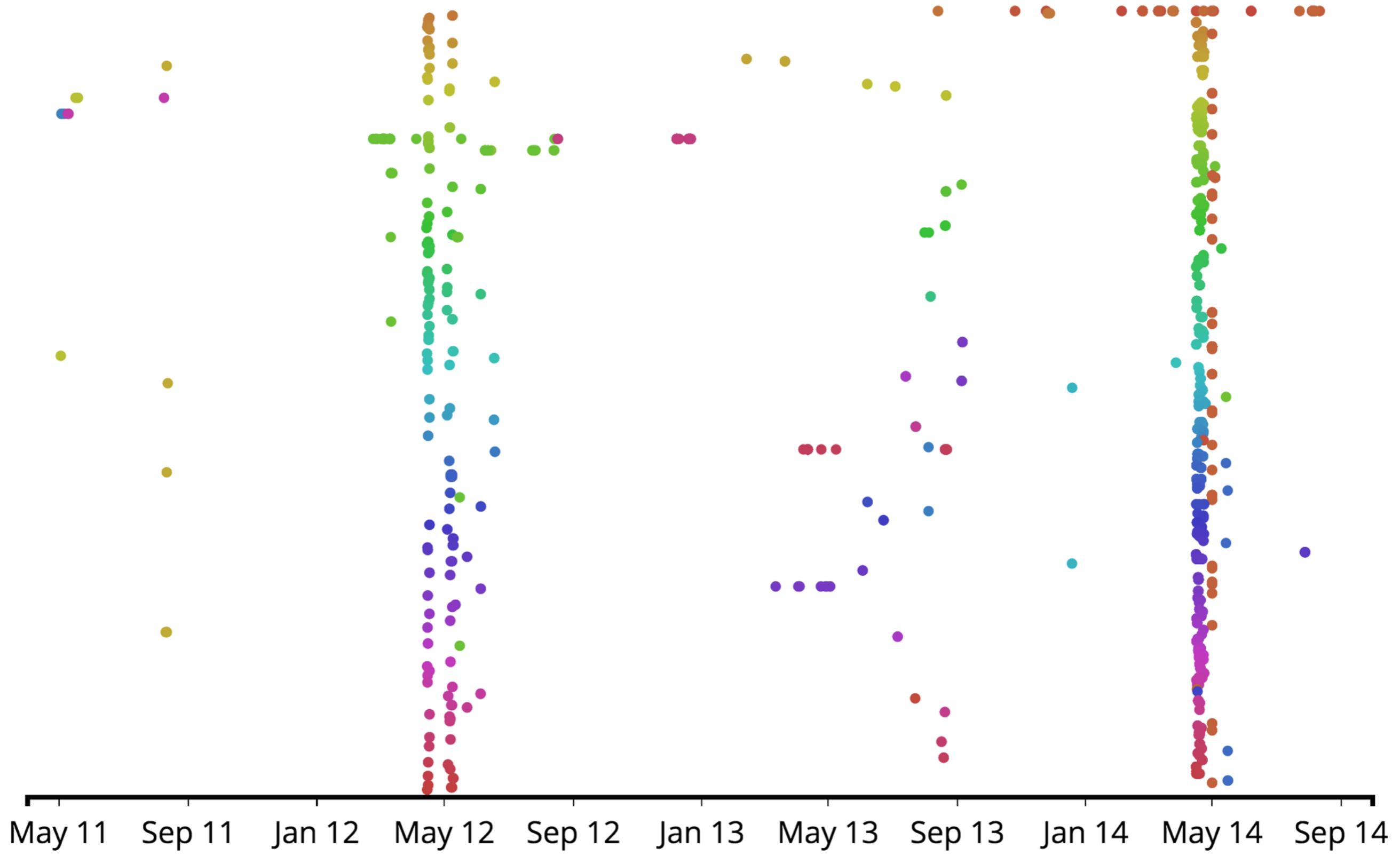
**IF YOU WANT TO
FOLLOW FROM HOME**

<https://filippo.io/hitb>

DOES THIS HAPPEN?

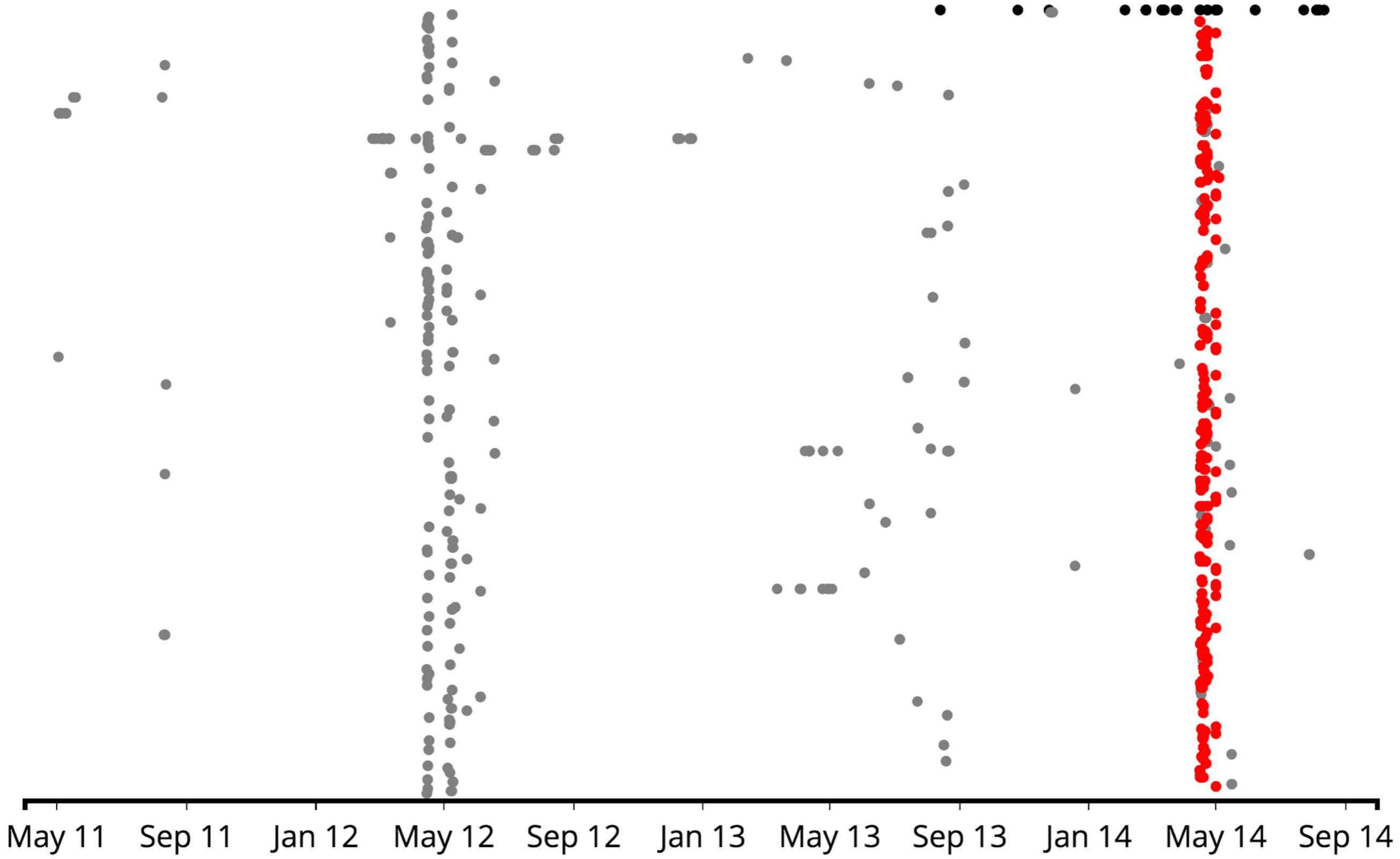
DOES THIS HAPPEN?

Yes.

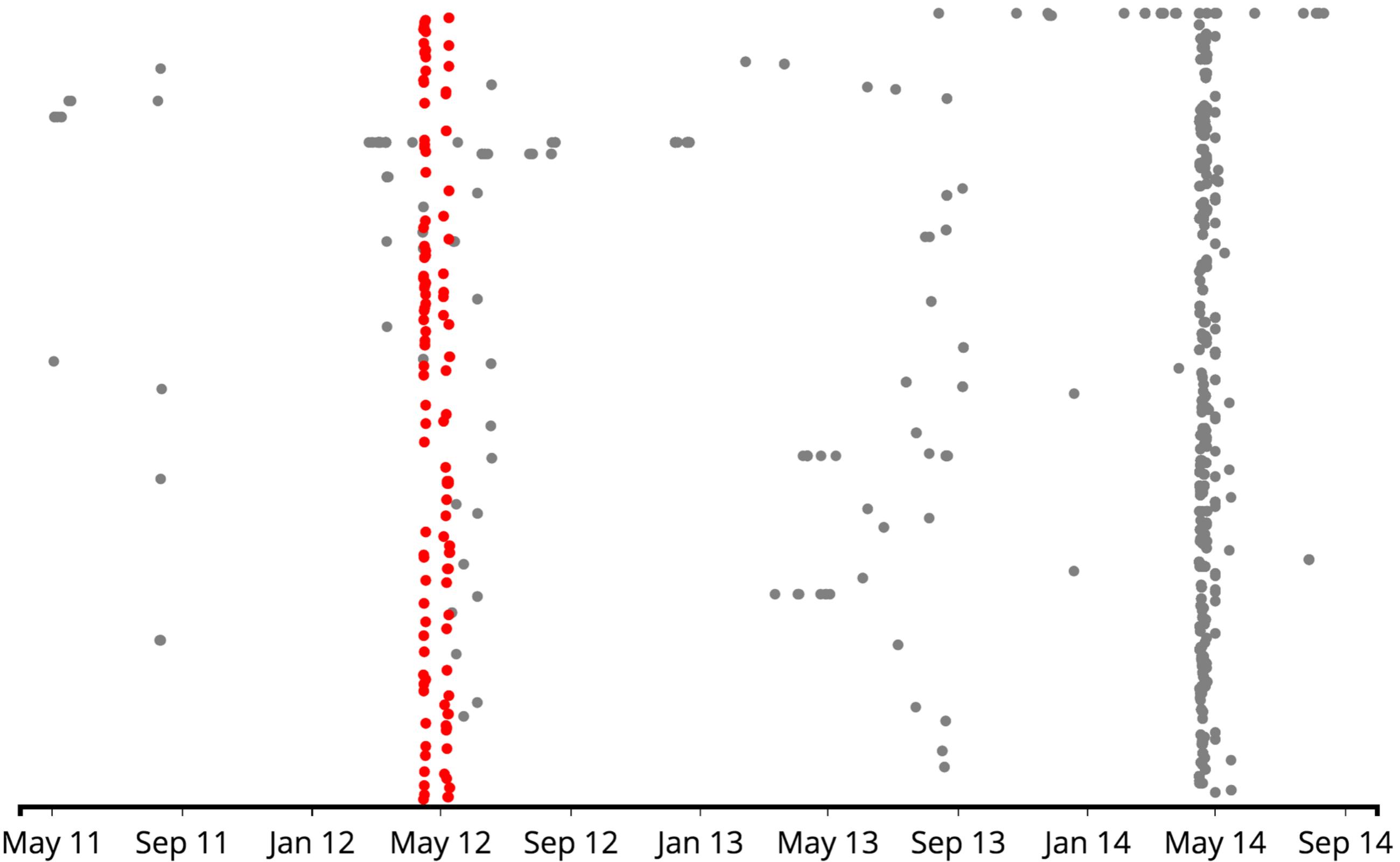


VERTICAL: ADDRESS

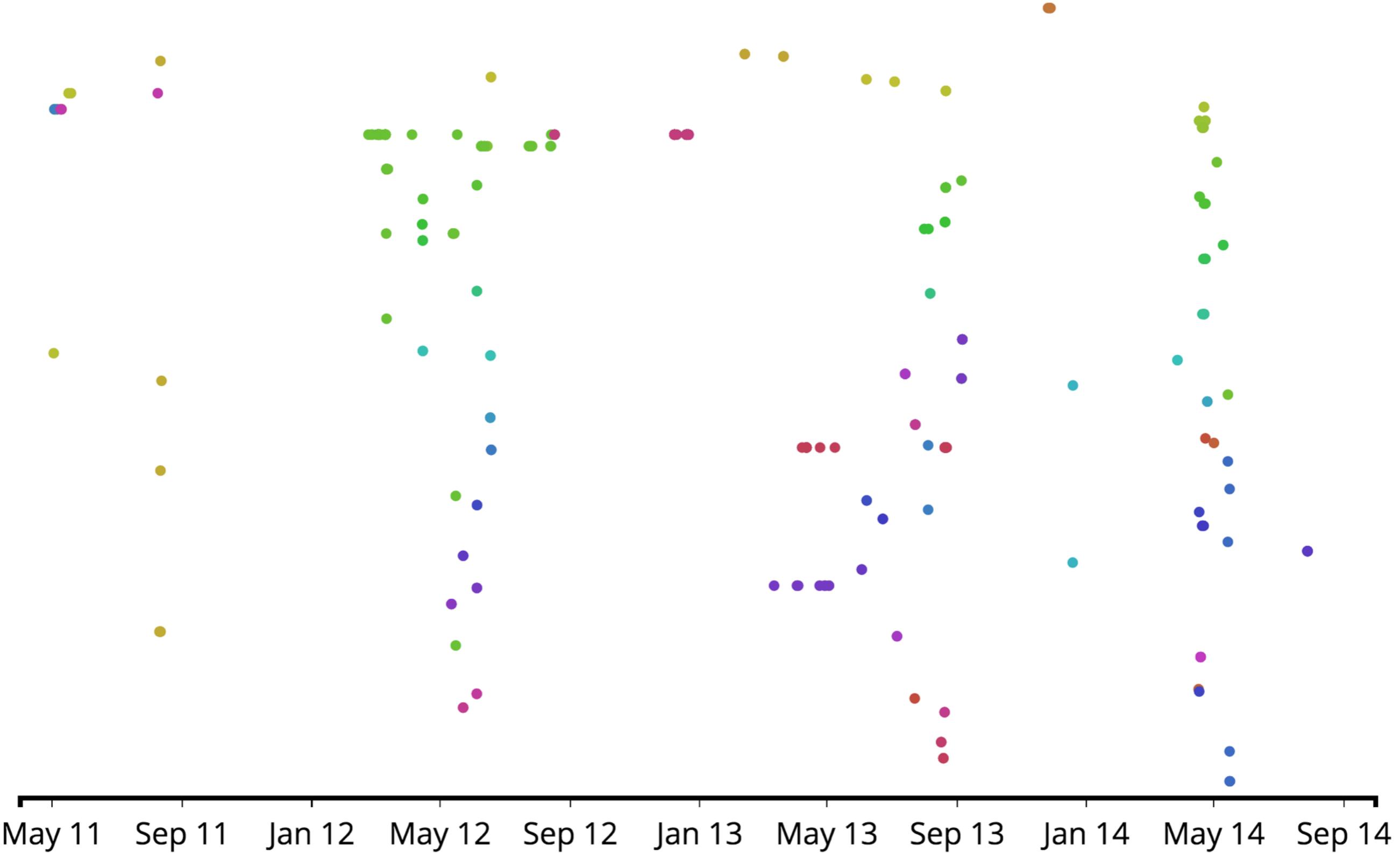
COLOR: R



WEIRD MULTISIGNATURE TRANSACTIONS

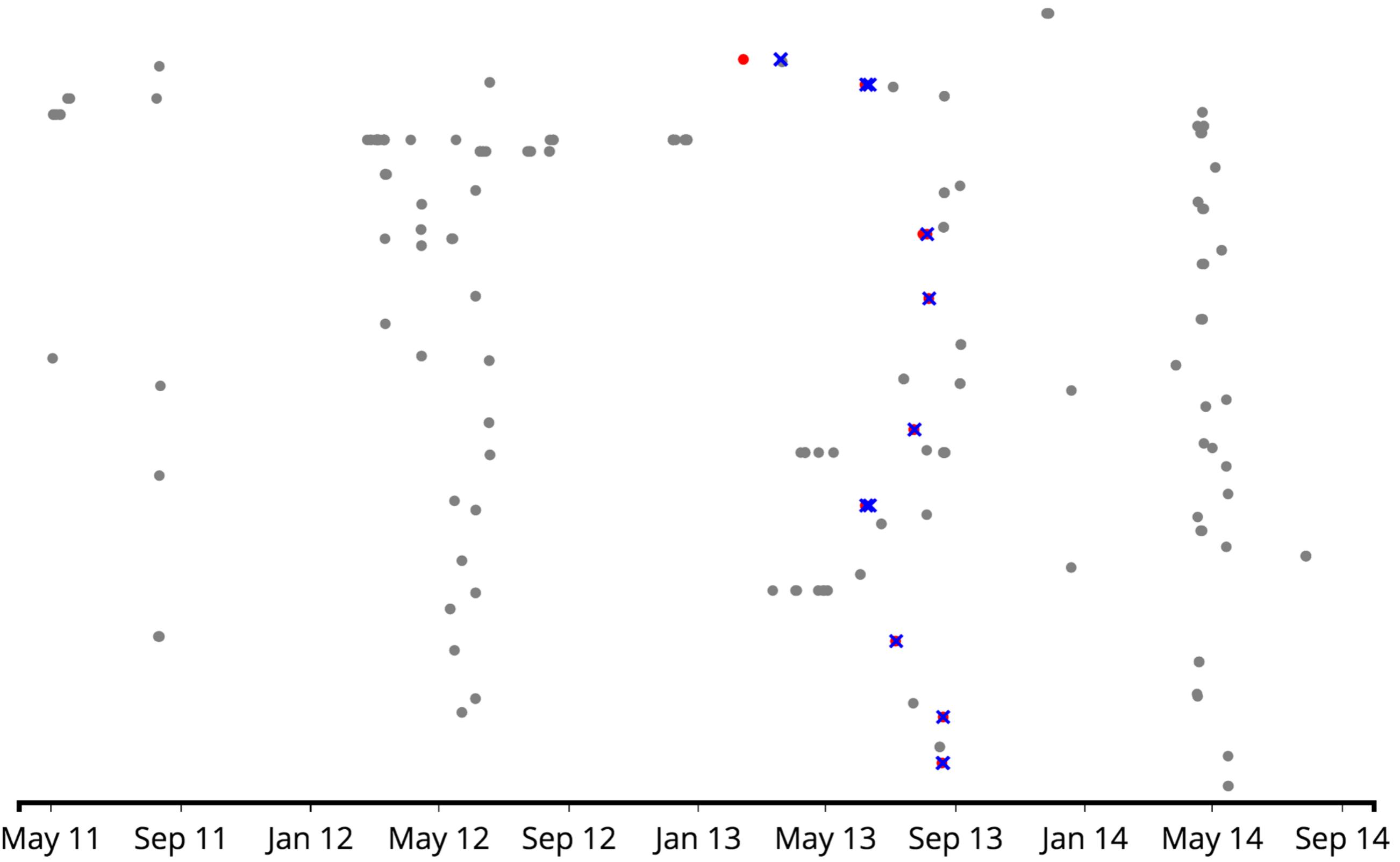


1KTJBE8YDX0QNTSYLG2RE4QTKK19KPVVL
1BKE8TTBRUKVNTJ3LX1EPSW7VVHULZHT



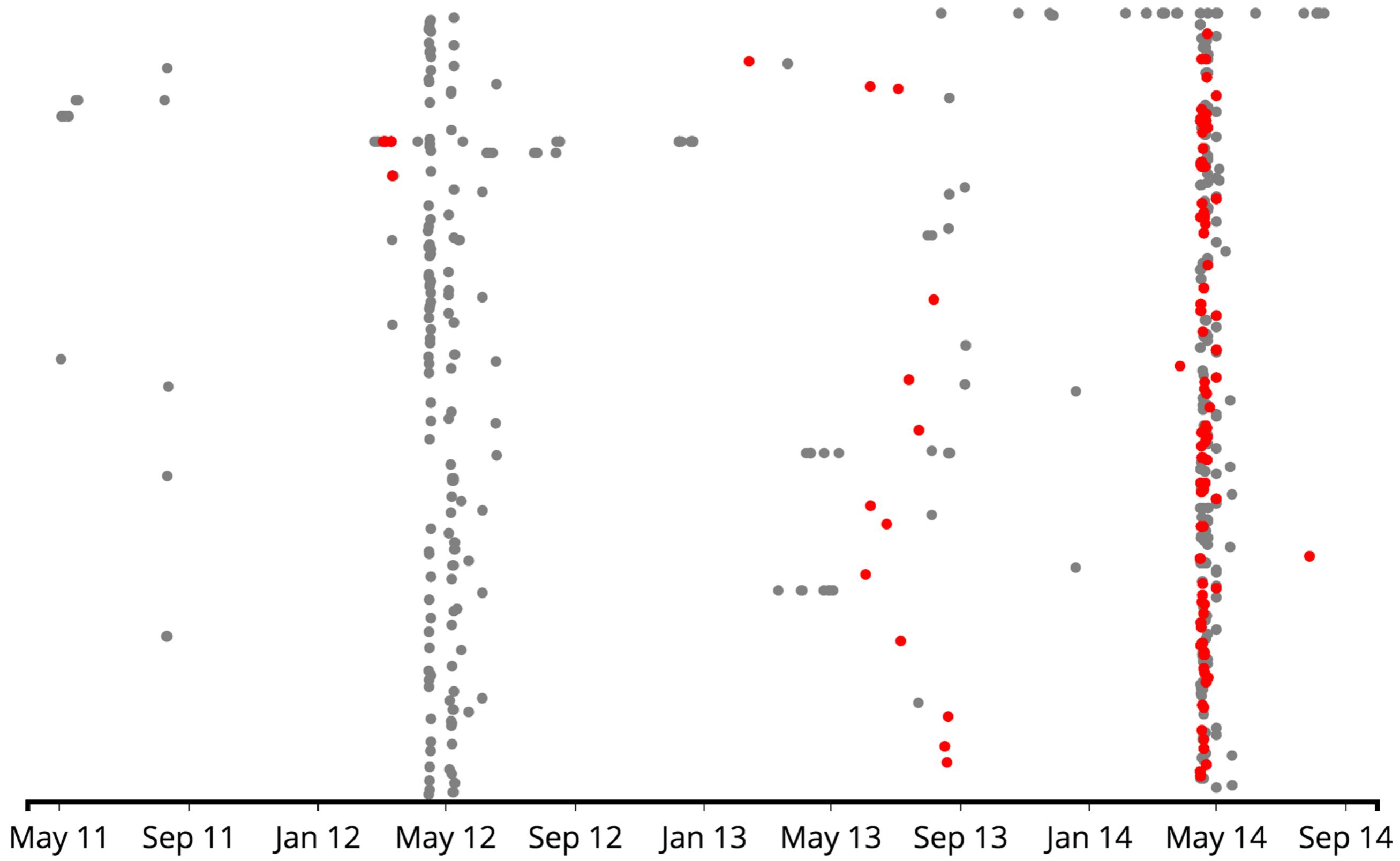
VERTICAL: ADDRESS

COLOR: R



“GOMEZ”

1GOZMCSMBC7BNMVUQLTKEW5VBXBSEG4ERW / 1HKYWXIL4JZIQXRZLKHM6A74MA6KXBSDJ



REPEATED R IN THE SAME TRANSACTION

“BAD SIGNATURES LEADING TO 55.82152538 BTC THEFT (SO FAR)”



Author

Topic: Bad signatures leading to 55.82152538 BTC theft (so far) (Read 33421 times)

BurtW

Legendary



Online

Activity: 1218

I no longer support vanity addresses



Ignore



Bad signatures leading to 55.82152538 BTC theft (so far)

#1

August 10, 2013, 10:53:13 PM

I have only seen this discussed in the newbies section so I thought I would open a thread here for a more technical discussion of this issue.

Several people have reported their BTC stolen and sent to <https://blockchain.info/address/1HKywxiL4JziqXrzLKhmb6a74ma6kxbSDj>

As you can see the address currently contains 55.82152538 stolen coins.

It has been noticed that the coins are all transferred in a few hours after a client improperly signs a transaction by reusing the same random number. As discussed here:

[HTTPS://BITCOINTALK.ORG/INDEX.PHP?TOPIC=271486](https://bitcointalk.org/index.php?topic=271486)

BLOCKCHAIN.INFO SECURITY

[FUNDS STOLEN]



Author

Topic: Blockchain.info security [FUNDS STOLEN] (Read 15842 times)

giantdragor

Legendary



Activity: 1190



Ignore



Blockchain.info security [FUNDS STOLEN]

August 19, 2013, 03:27:16 PM

#1

I used Blockchain.info online wallet for small transactions on my Windows 7 64-bit PC with strong password kept in KeePass. Today I noticed that about 1.8 BTC was stolen from one of the addresses (which used for Anonymous Ads earnings), but funds from other addresses in this wallet were not affected. This leads me on thoughts that **Blockchain.info or Firefox may have some weakness in random number generator** like the vulnerability was recently found in the Android.

TXID with my funds gone:

<https://blockchain.info/tx/975412ecc21a0ad949deba3f47c6ac41e42fb7>

[HTTPS://BITCOINTALK.ORG/INDEX.PHP?TOPIC=277595](https://bitcointalk.org/index.php?topic=277595)

Android Security Vulnerability

11 August 2013

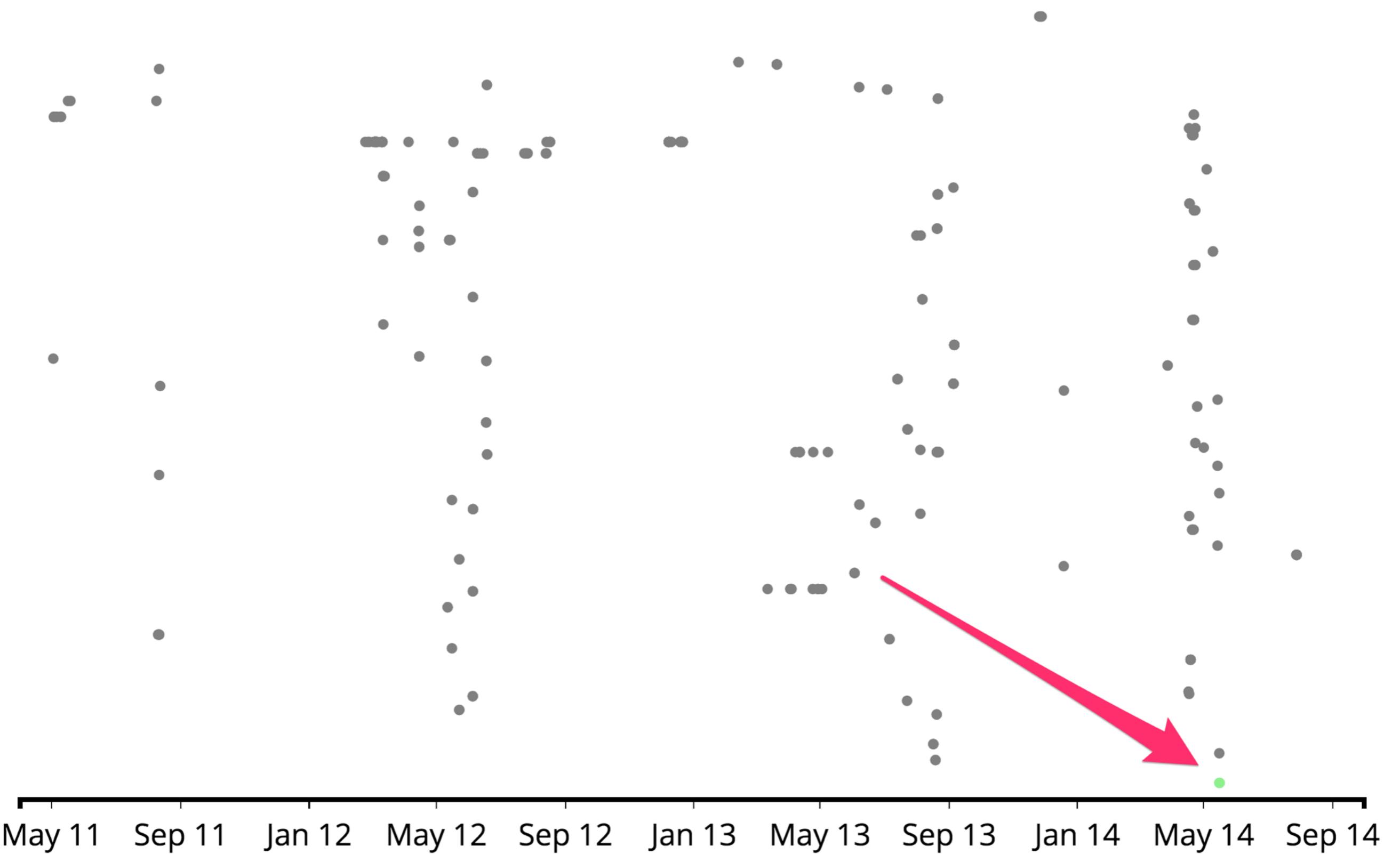
What happened

We recently learned that a component of Android responsible for generating secure random numbers contains [critical weaknesses](#), that render all Android wallets generated to date vulnerable to theft. Because the problem lies with Android itself, this problem will affect you if you have a wallet generated by any Android app. An incomplete list would be [Bitcoin Wallet](#), [blockchain.info wallet](#), [BitcoinSpinner](#) and [Mycelium Wallet](#). Apps where you don't control the private keys at all are not affected. For example, exchange frontends like the Coinbase or Mt Gox apps are not impacted by this issue because the private keys are not generated on your Android phone.

What has been done

Updates have been prepared for the following wallet apps:

- **Bitcoin Wallet:** Update 3.15 can be installed from [Google Play](#) or [Google Code](#). Key rotation will occur automatically soon after you upgrade. The old addresses will be marked as insecure in your address book. You will need to make a fresh backup.
- **BitcoinSpinner:** Update 0.8.3b can be installed from [Google Play](#) or [Google Code](#). On startup it will advise you on how to proceed.
- **Mycelium Bitcoin Wallet:** Update 0.7.0 can be installed from [Google Play](#) or [mycelium.com](#). A wizard will guide you through the process of moving your bitcoins to newly generated addresses, and put the old keys into archive mode.



NICK SULLIVAN “EXPLOITING RANDOMNESS” DEMO

January 28, 2013

Recovering Bitcoin private keys using weak signatures from the blockchain

On December 25th of last year I discovered a potential weakness in some Bitcoin implementations. Have a look at this transaction:

transaction: 9ec4bc49e828d924af1d1029acf709431abbde46d59554b62bc270e3b29

input script 1:

30440220d47ce4c025c35ec440bc81d99834a624875161a26bf56ef7fdc0f5d52f843ad10

 [-] **btcrobinhood** 37 points 1 year ago*

First off, I did not steal these coins. That said, I knew about the flaw. If you're worried your address might be vulnerable, here's a list (albeit compiled as of last month) of all the addresses that are vulnerable. If your address is on this list, expect coins sent to them to be snatched immediately: <https://gist.github.com/anonymous/6204930>

Edit1: I've re-run my little program to find vulnerable addresses. It turns out in all of July/Aug there were only 6. New addresses not in my posted list are
17HHdLh4oXncuTejALwC6fgArVqPUxh2Sr
1BFhrfTTZP3Nw4BNy4eX4KFLsn9ZeijcMm
1FPSVbypWa7rBWbcKHJ983YWcucBn7aUQ

Any developer who suspects this may have something to do with their wallet software, feel free to contact me for more detailed information (i.e. which specific tx inputs / signatures were foobar + k values recovered).

Edit2: To clarify, I did not know about this flaw until now ... I just knew bad signatures existed on the blockchain. This is hella serious ... **any key you previously used with an android wallet should be retired regardless of it's presence on my posted list.**

THE FIX

WHAT'S NEEDED

k must be secret and unique

NOT NECESSARILY RANDOM

RFC 6979

Generate k deterministically,
as a function of private key
and message.

K = HMAC_DRBG (D, H (M))

BITCOIN CORE



UNSAFE: OPENSSL
PATCH BY AGL WAITING ON MASTER

Add secure DSA nonce flag.

This change adds the option to calculate (EC)DSA nonces by hashing the message and private key along with entropy to avoid leaking the private key if the PRNG fails.

by master

 Adam Langley authored on Jan 25, 2013

→ **benlaurie** committed on Jun 14, 2013

Make `safe' (EC)DSA nonces the default.

This change updates [8a99cb2](#) to make the generation of (EC)DSA nonces using the message digest the default. It also reverts the changes to (EC)DSA_METHOD structure.

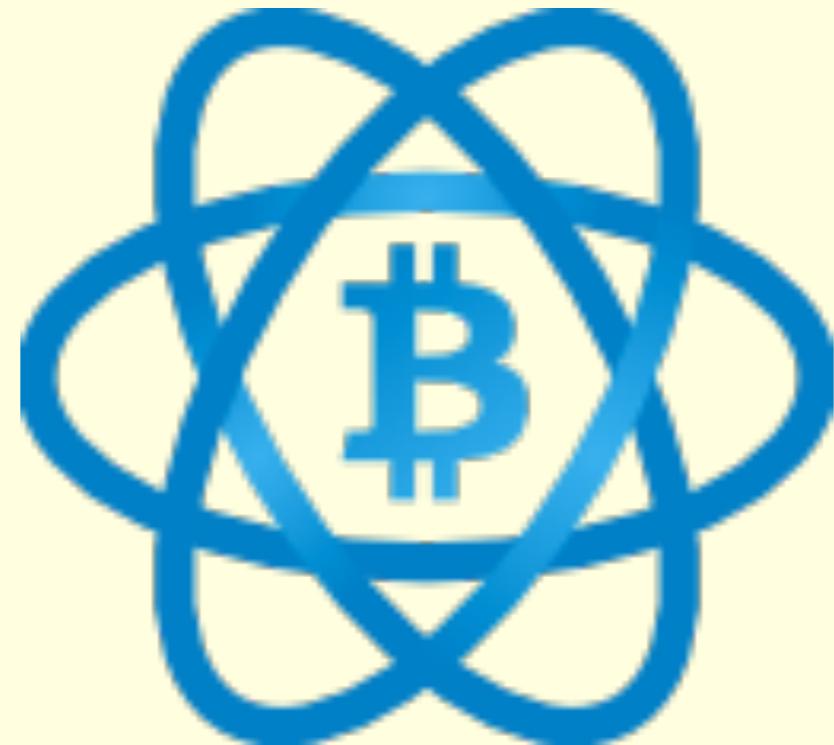
In addition to making it the default, removing the flag from EC_KEY means that FIPS modules will no longer have an ABI mismatch.

by master

 Adam Langley authored on Jul 15, 2013

→ **snhenson** committed on Jul 15, 2013

ELECTRUM



SAFE SINCE V1.9

CORRECT USE OF PYTHON-ECDSA

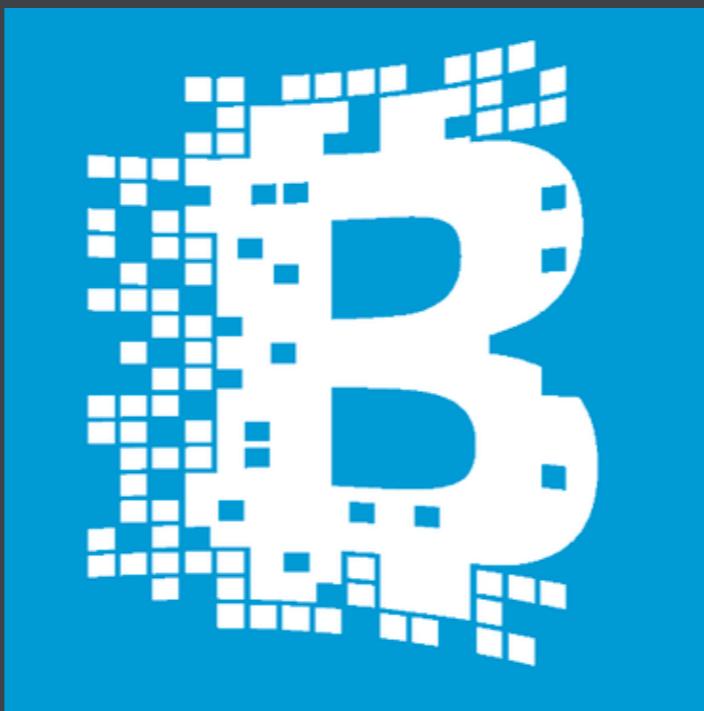
MULTIBIT / BITCOINJ



SAFE

CORRECT USE OF BOUNCYCASTLE

BLOCKCHAIN.INFO



UNSAFE
RELIES ON THE BROWSER RNG (IF ANY!)

```
224 ▼  var ECDSA = {  
225 ▼    getBigRandom: function (limit) {  
226 ▼      return new BigInteger(limit.bitLength(), rng)  
227        .mod(limit.subtract(BigInteger.ONE))  
228        .add(BigInteger.ONE)  
229        ;  
230    },  
231 ▼    sign: function (hash, priv) {  
232      var d = priv;  
233      var n = ecpParams.getN();  
234      var e = BigInteger.fromByteArrayUnsigned(hash);  
235  
236 ▼      do {  
237        var k = ECDSA.getBigRandom(n);  
238        var G = ecpParams.getG();  
239        var Q = G.multiply(k);  
240        var r = Q.getX().toBigInteger().mod(n);  
241      } while (r.compareTo(BigInteger.ZERO) <= 0);  
242  
243      var s = k.modInverse(n).multiply(e.add(d.multiply(r))).mod(n);  
244  
245      return {r : r, s : s};  
246    },
```



BITRATED / BITCOINJS-LIB

```
SHA256(  
  hash  
  .concat(priv.toByteArrayUnsigned())  
  .concat(random),  
 { asBytes: true }));
```

SAFE

HASHES PRIVKEY, MESSAGE AND RANDOM

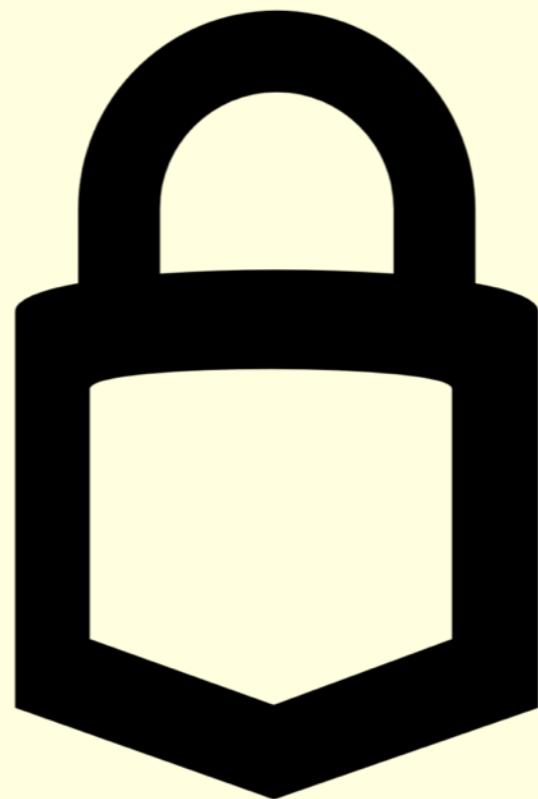
ARMORY



UNSAFE (?) - 90%

CRYPTO++ SEEMS TO USE A RANDOM VALUE

TREZOR



SAFE

IMPLEMENTS RFC 6979

@FILOSOTTILE

FILIPPO.IO/HITB-SLIDES

Q&A