Polynonce: An ECDSA Attack and Polynomial Dance

Nils Amiet
Marco Macchetti

August 12, 2023 - DEF CON 31

KUDELSKI SECURITY

Who are we?

- Nils Amiet
  - Security researcher @ Kudelski Security
  - Privacy
  - Data processing at scale
  - Linux enthusiast

- Marco Macchetti
  - Cryptographer @ Kudelski Security
  - Applied cryptography
  - Hardware design
  - Cryptanalysis

KUDELSKI SECURITY

Table of contents

Introduction

Digital signatures - the basics

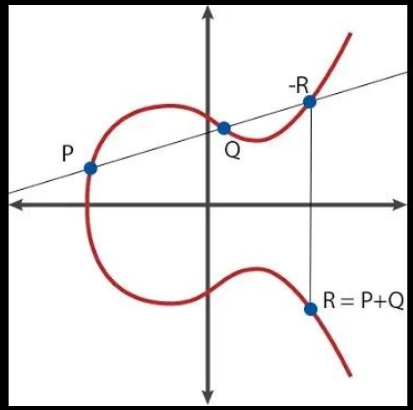
- What is a digital signature?
  - Proof of knowledge of private key
  - Verified with public key
  - Private key = identity
- On blockchains, transactions are digitally signed
  - Private key = 
- Anyone can try to steal your key. Wherever it is stored.
  - Backups
  - Paper notes / metal plates
  - Brain wallets
  - A secret place
  - Sometimes it’s on storage and encrypted with a weak password…

Digital signatures - not so basic

- Signature algorithm = {keygen, sign, verify}
- Randomness?
  - Keygen :  => keypair
  - Sign : private key + message +  => signature
  - Verify : signature + message + public key => pass/fail
- Signature => private key ???
  - Yes, under some assumptions.
  - For example, in keygen if the key is generated with low entropy
  - … or using a brain wallet method
  - Small keys / easy to guess keys
- But provided the key is properly generated?
  -  during sign -> more attack surface!

ECDSA

- ECDSA is probably the most used and deployed signature algorithm
- “EC” stands for elliptic curve
- An elliptic curve is a curve…
  - $y^2 = x^3 + ax + b$ => {G, a, b, N, P,}
  - secp256k1, “the Bitcoin curve”
- And a set of points…
- We can perform operations with points and remain in the group
  - R=P+Q
  - R=[5]P=P+P+P+P+P
- Why EC?
- Because of the EC discrete logarithm problem (ECDLP)
  - k, G ⇒ [k]G EASY
  - [k]G, G ⇒ k HARD
- They can be used to implement digital signatures! -> ECDSA



ECDSA {keygen, sign, verify}

- Keygen
  - pick integer d in [1, n-1] uniformly at random
  - Congratulations : d is your private key
  - Q = [d]G is your public key
- Sign
  - Hash the message h = H(M)
  - pick integer k in [1, n-1] uniformly at random => the “nonce”
  - r = x of R = [k]G
  - $s = k^{-1}(h + rd)$
  - Signature is (r, s)
- Verify
  - Hash the message h = H(M)
  - Compute $u1 = hs^{-1}$ and $u2=rs^{-1}$
  - Compute R = [u1]G + [u2]Q
  - If x of R == r then PASS else FAIL

- Takeaway: a random value **k** (the nonce) needs to be generated for each ECDSA signature
  - And it should be unique and non-biased

Nonces, nonces....

- Keys are generated once... nonces once per signature!
- If bits of the nonces can be predicted or are known...
  - Lattice attacks
- Let's ignore (remote) side channel attacks
  - But they can be used in combination with lattice attacks
- Let's pick our dice!
  - Good ones: CSPRNGs, HMAC, AES, Yarrow, etc...
  - Bad ones: LCGs, QCGs, LFSRs, etc... -> NIST 800-22 criticism
  - Awful ones: rand, fixed secret constants (incremented), dice roll, playstation3, etc...
- We still have to seed it... with enough entropy
- Power-on attacks? Do we have to seed every time??

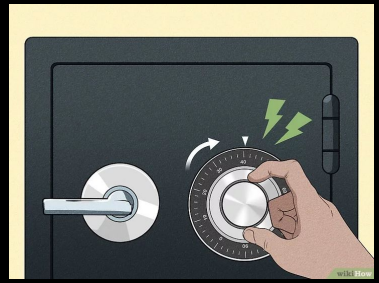
Bad RNGs - let's pick one

- Linear congruential generator (LCG)
- GMP, other libs, NIST 800-22
- Random numbers are obtained as $k_1 = a*k_0 + b$ with known a, b (usually b=1)
  - k_1 depends linearly on k_0
- No bit bias but…
- As shown by Google Paranoid Crypto project
  - Lattice attacks can be used to recover the private key
  - 22 signatures needed to attack an {128, 64} LCG generating nonces for the Bitcoin curve (256 bits)
- And what about full-state (256-bit) LCG modulo N?
  - No results published w.r.t. ECDSA
  - What about increasing degree to quadratic (QCG) or cubic (CCG)? -> lattice attacks are not possible
  - with UNKNOWN coefficients???!?!?
- It should be safe, right...?
- NO! -> Polynonce -> private key can be obtained in a matter of fractions of a second.

Introducing Polynonce: A novel ECDSA attack

Intuition behind the attack

- One (non working) way to break ECDSA is to solve the discrete logarithm problem
  - r = x of R=[k]G
- And what about the other half of each signature, s?
  - Rewriting the expression of s => **k**=h/s+(r***d**)/s
  - linear relationship between the nonce and the static private key **d**!
- So, there is only one secret behind all your signatures!
- For instance, we can write:
  - R_1=[a]R_0+[b]G with known a and b
  - You are not supposed to write this if you don't know the discrete logs of R_0 and R_1…
- What does this all mean?
  - The value of s brings additional information!
- How can we use it to attack the private key?



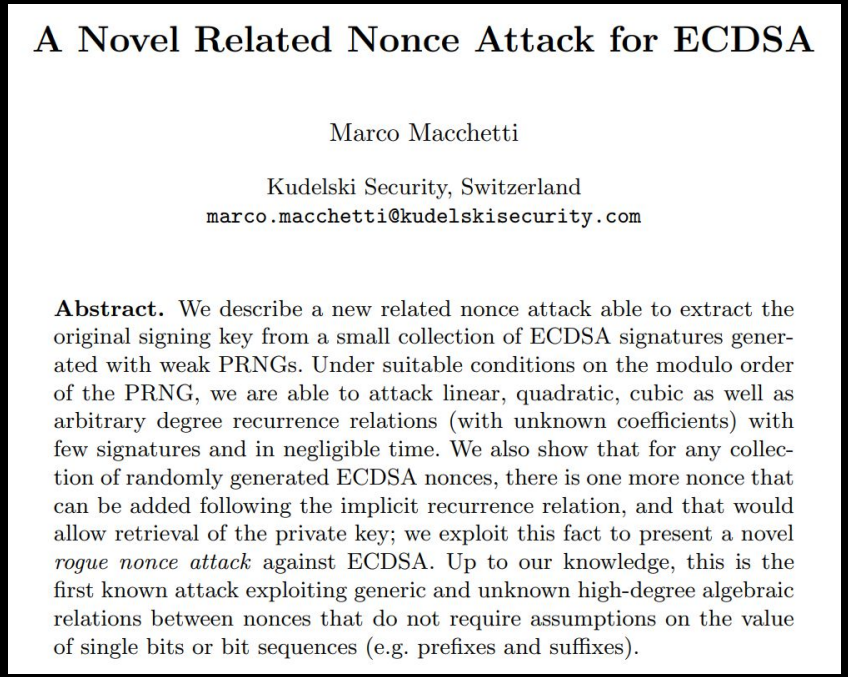
Polynonce on LCGs

- We can break unknown-coefficients LCGs with full state with 4 signatures and 100% success
- Suppose that nonces are generated as follows:
  - $k_1 = a_1 k_0 + a_0$
  - $k_2 = a_1 k_1 + a_0$
  - $k_3 = a_1 k_2 + a_0$
  - And we don't know a1, a0
- We can subtract the second equation from the first, and the third from the second
  - $(k_1 - k_2) = a_1(k_0 - k_1)$
  - $(k_2 - k_3) = a_1(k_1 - k_2)$
- We got rid of a_0.... and then we can write (we skip a few steps here):
  - $(k_1 - k_2)(k_1 - k_2) = (k_0 - k_1)(k_2 - k_3)$
- We also got rid of a_1!
- We got a polynomial in k_i... but we can substitute each k_i with its expression in d!
- We got a polynomial with only unknown **d**!
- We find its roots... and among them we find **d**, the private key!

Remember:
- k=h/s+(r*d)/s
- (r,s): signature
- h: msg hash
- d: private key

Extensions

- In the eprint paper, we extend all this to n-degree polynomial relations
- This is possible using a recursive algorithm that eliminates all unknown coefficients
- It is:
  - Fast
  - Generic
  - Not using lattice algorithms
  - No need for pre-computations
- https://eprint.iacr.org/2023/305



A Novel Related Nonce Attack for ECDSA

Marco Macchetti

Kudelski Security, Switzerland
marco.macchetti@kudelskisecurity.com

Abstract. We describe a new related nonce attack able to extract the original signing key from a small collection of ECDSA signatures generated with weak PRNGs. Under suitable conditions on the modulo order of the PRNG, we are able to attack linear, quadratic, cubic as well as arbitrary degree recurrence relations (with unknown coefficients) with few signatures and in negligible time. We also show that for any collection of randomly generated ECDSA nonces, there is one more nonce that can be added following the implicit recurrence relation, and that would allow retrieval of the private key; we exploit this fact to present a novel *rogue nonce attack* against ECDSA. Up to our knowledge, this is the first known attack exploiting generic and unknown high-degree algebraic relations between nonces that do not require assumptions on the value of single bits or bit sequences (e.g. prefixes and suffixes).

How to attack things with all of that?

Polynonce - Requirements in practice

- In practice, what do we need to run the attack?
  - At least 4 signatures generated by the same private key
    - More signatures are needed for higher degree relations between nonces
  - The associated public key
  - The message hash associated with each signature
  - Signatures must be ordered by generation time
- What's the output and impact of the attack when it succeeds?
  - If the nonces follow the relation:
    - We retrieve the private key that was used to generate the vulnerable signatures

Bitcoin

- A block contains multiple transactions
  - Transactions can be of multiple types
- Each transaction contains inputs and outputs
- Each input is signed with ECDSA and curve secp256k1, SHA-256 hashed
- What do we need to attack it?
  - Public key
    - Contained in the input
  - Signature (r, s)
    - Also contained in the input
  - Message
    - Must be computed

Bitcoin - Computing the message

- Message not only depends on fields of the current transaction
- Also depends on fields from previous transactions
- Procedure to compute the message is error-prone and under documented
- After lots of trial and error, we computed the right message
- How to check we got it right?
  - Verify the signatures

Remember:
- ECDSA-Verify = f(msg, signature, pubkey)

Bitcoin - Obtaining the data

- Sync with Bitcoin Core
  - Takes about 24 hours to sync
  - 430 GB disk space
- All blocks until September 5, 2022
- Dump the signatures, messages and public keys
  - Forked rusty-blockparser, written in Rust
  - Read block files from disk
  - Takes 24 hours (on SSD)
  - Dump size: 271 GB

- 763 million signatures dumped
- Only P2PKH (Pay-to-Public-Key-Hash) transactions
  - Most common transaction type (55% of all transactions)
- Our open source code is available at:
  - https://github.com/kudelskisecurity/ecdsa-dump-bitcoin

Ethereum

Ethereum

- Transactions are signed with ECDSA
  - Same curve and hash function as Bitcoin
  - Many other blockchains use ECDSA
  - 3 out of 4 of the top blockchains use ECDSA
- What do we need to attack it?
  - Signature (r, s)
    - Contained in the transaction
  - Public key
    - Not directly in the transaction
    - Can be recovered from the signature and message hash (ECDSA-Recover)
  - Message
    - Must be computed

Ethereum - Computing the message

- Multiple Ethereum protocol versions
  - Protocol version applies based on block number
  - Examples:
    - Ethereum version “Spurious Dragon” (blocks 2’675’000 to 4’369’999)
    - Ethereum version “Berlin” (blocks 12’244’000 to 12’964’999)
- Message is computed differently based on protocol version
- No need to refer previous transactions, no inputs and outputs
- How to check we got it right?
  - Compute the message
  - Recover the public key from message and signature
  - Verify signature
  - Pubkey => Wallet address, then check wallet address == source address (in tx)
- Our open source code is available at:
  - https://github.com/kudelskisecurity/ecdsa-dump-ethereum

Ethereum - Obtaining the data

- Install + run
  - geth (execution client)
  - lighthouse (consensus client)
- Let the chain sync
  - Takes about 3 weeks
  - geth takes 1.6 TB disk space
  - lighthouse takes 120 GB disk space
- All blocks until October 28, 2022
- Dump the signatures, messages and public keys
  - Written in Python
  - Use geth JSON-RPC API to get blocks in JSON
  - Takes about 3 days
  - Output file size: 628 GB
- 1.7 billion signatures dumped

Other datasets we explored (not only blockchains)

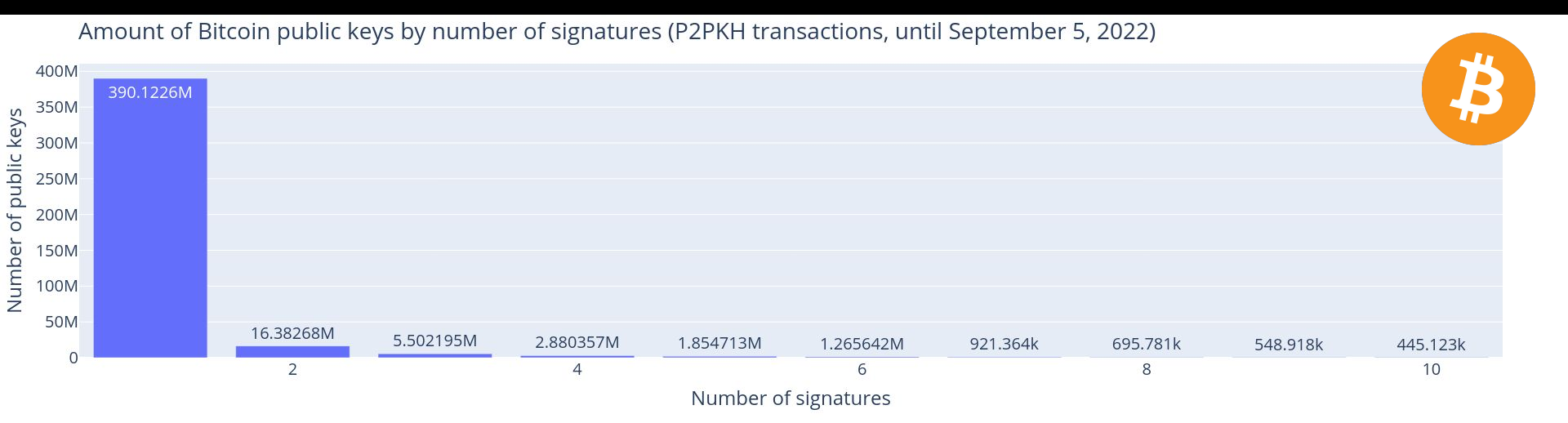
Sample of TLS servers

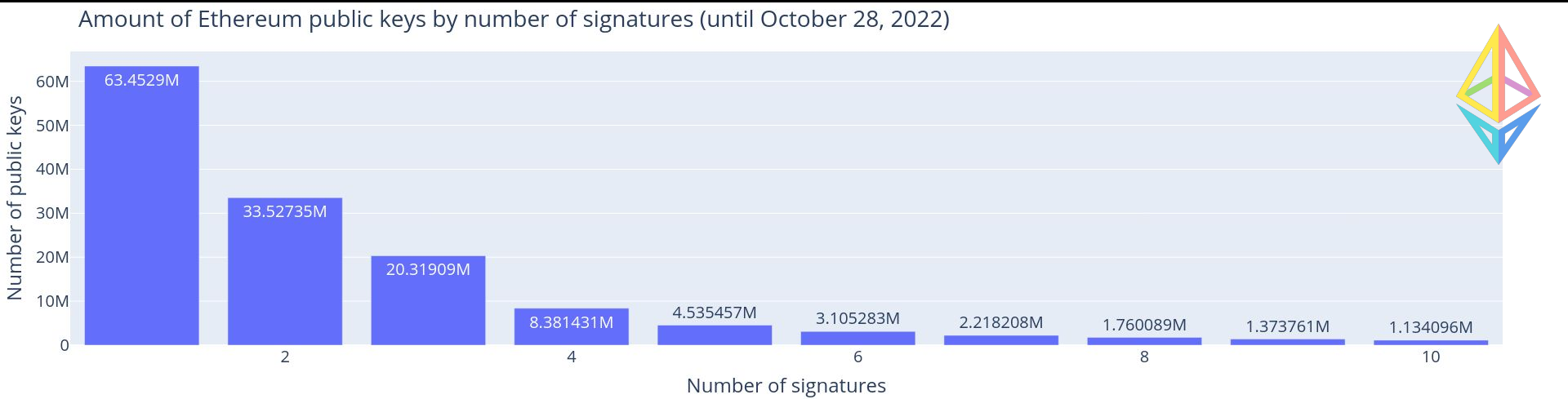
- TLS handshake
  - Signatures are in ServerKeyExchange TLS messages
- Used list of domains that receive the most traffic
  - Cisco Umbrella 1 Million domains
- 3 network scans on TLS servers
  - Sample of ~10k targets
  - Cipher suite: ECDHE-ECDSA-AES128-SHA256
  - Make sure server uses an ECDSA key
  - Total scan time, less than 24 hours
- About 6k unique signatures dumped in total
  - Just a small sample of what’s out there

Minerva datasets

- Public datasets of ECDSA signatures
- We covered these datasets (1 smartcard, 1 TPM, 4 software libraries)
  - Athena IDProtect
  - TPM-FAIL
  - libgcrypt
  - MatrixSSL
  - WolfCrypt
  - Simulated
- Each dataset contains 50k signatures, except TPM-FAIL (383k)
  - Signatures in a dataset are all generated with the same private key and the same message
- Only sort by timestamp is needed
  - Datasets ready for use
- https://github.com/crocs-muni/minerva

Statistics


Amount of Bitcoin public keys by number of signatures (P2PKH transactions, until September 5, 2022)
Number of public keys
400M
350M
300M
250M
200M
150M
100M
50M
0
390.1226M
16.38268M
5.502195M
2.880357M
1.854713M
1.265642M
921.364k
695.781k
548.918k
445.123k
2
4
6
8
10
Number of signatures


Amount of Ethereum public keys by number of signatures (until October 28, 2022)
Number of public keys
60M
50M
40M
30M
20M
10M
0
63.4529M
33.52735M
20.31909M
8.381431M
4.535457M
3.105283M
2.218208M
1.760089M
1.373761M
1.134096M
2
4
6
8
10
Number of signatures

Stats

- Bitcoin (P2PKH)
  - 424 M unique public keys
  - 97% generated less than 4 signatures
  - The rest: 12M pubkeys we can try to attack
- Ethereum
  - 151 M unique public keys
  - 77% generated less than 4 signatures
  - The rest: 34M pubkeys we can try to attack

Attack setup

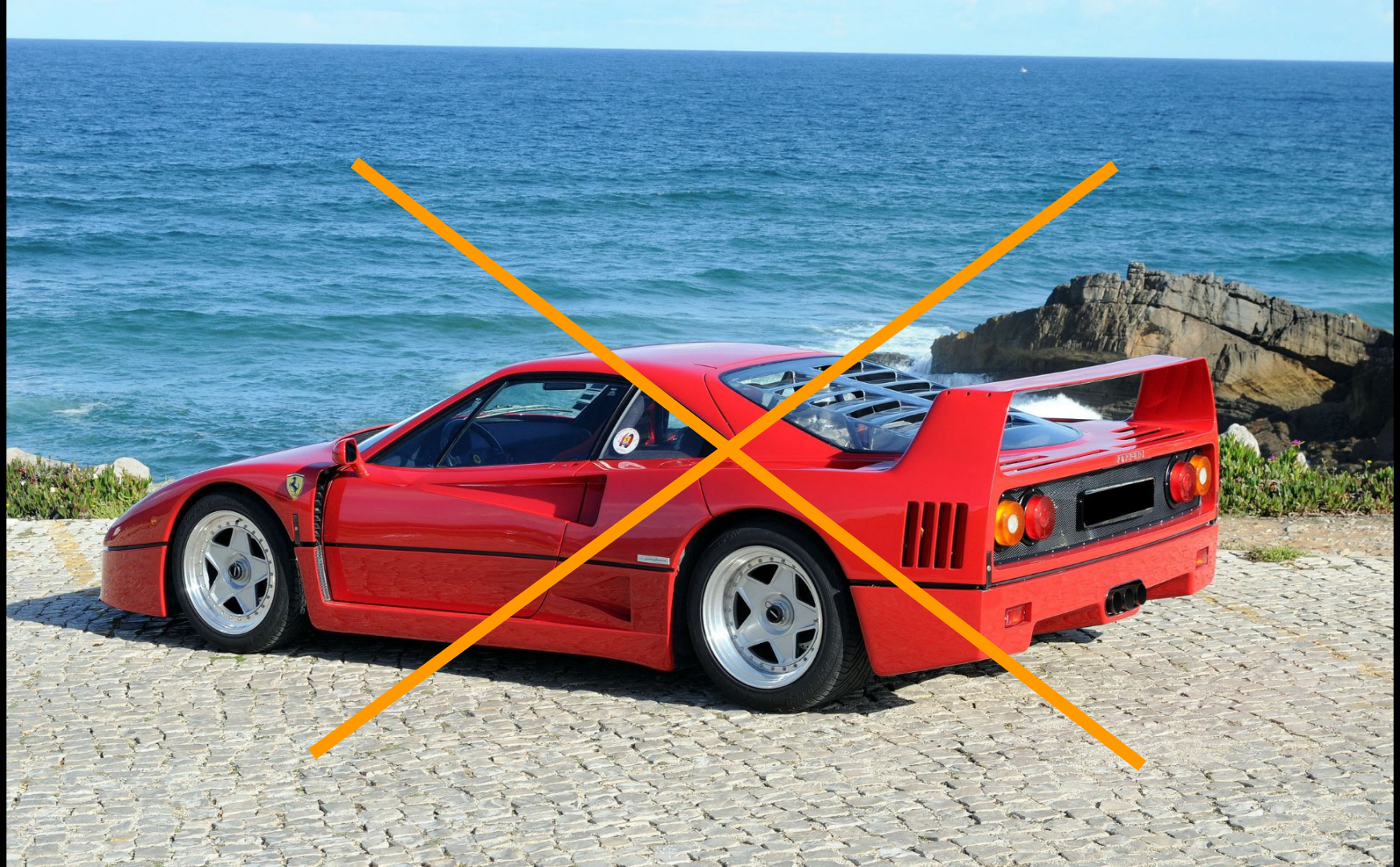
Implementing the attack

- Implemented in Sagemath (Python)
- Sliding window of size N signatures of the same public key
- Multithreaded implementation
- Our open source code is on Github:
  - https://github.com/kudelskisecurity/ecdsa-polynomial-nonce-recurrence-attack

Demo

- Attacking a vulnerable TLS server
  - Establish multiple TLS connections to server
  - Dump signatures, messages and public keys
  - Run the attack
  - Get the private key

Results



Results - Bitcoin and Ethereum

- 128-core VM
- N=5 signatures (quadratic)
- Sort signatures by block time
- Bitcoin
  - Runtime: 2 days and 19 hours
  - Retrieved private keys of 773 unique wallets
  - All had a zero balance
  - Estimated cost: 285 USD
- Ethereum
  - Did 22% in 48 hours
  - Retrieved private keys of 2 unique wallets
  - Both had zero balance
  - Estimated cost (for 100%): 900 USD
- Turns out they all had signatures with repeated nonces
- Polynonce found those but they could also have been found in a different way

Results - TLS, Minerva

- 4-core laptop
- N=4,5,6
- TLS
  - Runtime: just a few seconds
  - Cost: cheap
  - Out of our small sample, no successful attacks
  - Difficult to find a target where consecutive signatures can be triggered
- Minerva
  - Runtime:
    - 50k datasets: about 10 minutes each
    - TPM-fail: about 70 minutes each
    - Total: less than 6 hours
  - Cost: cheap
  - No successful attacks on the 6 datasets tested

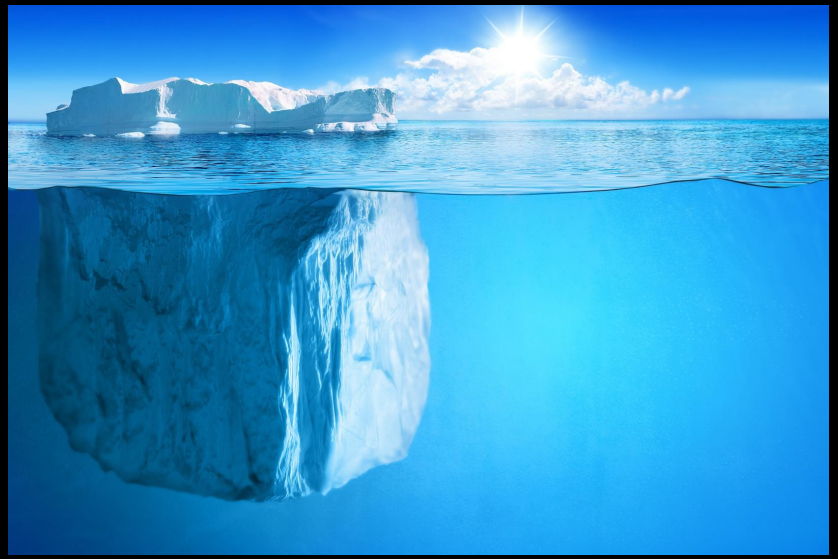
Fun facts

- Where did the stolen Bitcoin go?
  - 466 different wallets that received stolen funds
    - Top wallet received 75 BTC
  - Total 144 BTC stolen (9.4M USD at Bitcoin peak)
  - Vanity addresses used
    - 1idiot, 1Kgift, 1dust, 1Hack, etc.
  - 1idiot => 1andreas in early 2018
    - Andreas Antonopoulos, received over 100 BTC in donations
  - Mostly happened between 2014 and 2017
- Bitcointalk forums
  - Public talk about stolen funds (2014, 2016) due to repeated nonces
  - “So far I have collected about 7 BTC.” - forum user, April 2016

Conclusions

Conclusions

- Make sure nonce generation is not biased
- Safer alternatives to ECDSA that use deterministic nonce generation
  - Deterministic ECDSA (RFC 6979)
  - EdDSA
- We barely scratched the surface
  - Email signatures
  - Signed executables
  - TLS servers out there
  - Rest of Ethereum and Bitcoin (non-P2PKH)
    - + Higher degree relations
  - Other blockchains
  - Smartcards, constrained devices
  - etc.
- It's easy for you to check your own wallet
  - No cloud needed



Thank you

- For more research from our team
  - Research blog
    - https://research.kudelskisecurity.com
    - More exciting recent research on this topic
- We hope to make people more secure by releasing our code
  - https://github.com/kudelskisecurity
- Questions?

KUDELSKI SECURITY