DE GRUYTER
OPEN

Tatra Mountains
Mathematical Publications
DOI: 10.1515/tmmp-2016-0030
Tatra Mt. Math. Publ. **67** (2016), 55–68

SPEED OPTIMIZATIONS IN BITCOIN KEY RECOVERY ATTACKS

Nicolas Courtois — Guangyan Song — Ryan Castellucci

ABSTRACT. In this paper, we study and give the first detailed benchmarks on existing implementations of the secp256k1 elliptic curve used by at least hundreds of thousands of users in Bitcoin and other cryptocurrencies. Our implementation improves the state of the art by a factor of 2.5 with a focus on the cases, where side channel attacks are not a concern and a large quantity of RAM is available. As a result, we are able to scan the Bitcoin blockchain for weak keys faster than any previous implementation. We also give some examples of passwords which we have cracked, showing that brain wallets are not secure in practice even for quite complex passwords.

1. Introduction

Bitcoin is a cryptocurrency, an electronic payment system based on cryptography. It was created by Satoshi Nakomoto[1] in 2008 [12]. In 2009, Bitcoin was launched as open-source software. Bitcoin is designed to be a fully decentralised peer-to-peer network—self-governing without support from trusted entities such as banks or governments. Bitcoin transactions are like checks but signed cryptographically instead of using ink. Transactions are broadcast to the peer-to-peer network and verified by each node. A public ledger called a "blockchain" records transactions pseudonymously.

Ownership of bitcoins implies that a user can spend bitcoins associated with a specific address (equivalent to a bank account). In order to spend the coins, a payer must digitally sign the transaction using their private key. The signed transaction is then broadcast to the peer-to-peer network. Everyone on the network can verify the signature that has been sent out. Anyone can spend all the bitcoin in a bitcoin address as long as they hold the corresponding private key.

2010 Mathematics Subject Classification: 62K05.
Keywords: Bitcoin, Elliptic Curve Cryptography, Crypto Currency, Brain Wallet.

[1] It is not known whether Satoshi Nakomoto is a real or pseudonym name or if it represents one person or a group

Once the private is lost, the bitcoin network will not recognize any other evidence of ownership.

Bitcoin uses digital signature to protect the ownership of coins and private keys are the only way of owning bitcoins. Thus it is very important to look at the technical details of the digital signature scheme used in bitcoin.

1.1. Structure of the paper

In this paper we study and give the first detailed benchmarks on existing secp256k1 elliptic curve implementations used in Bitcoin. Section 2 introduces background knowledge about elliptic curve cryptography and brain wallets. Section 3 reviews previous research work in this area. Section 4 gives detailed benchmark for existing method and our own implementation. Our implementation improves the state of the art by a factor of 2.5. Section 5 is the conclusion of this paper.

2. Background

2.1. Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) was independently proposed by Neal Koblitz [10] and Victor Miller [11] in 1985. It is a public-key cryptography protocol where each of the participants has a pair of keys. There is one private key which is kept as a secret by the owner and one public key which can be shared with anyone. In the past 10+ years ECC has been increasingly used in practise since its inclusion in standards by organisations such as ISO, IEEE, NIST, NSA etc. Elliptic curves are more efficient and offer smaller key sizes at the same security as other widely adopted public key cryptography schemes such as RSA [13].

An Elliptic Curve over finite field $\mathbb{F}_p$ where p is a large prime, can be formed by choosing the variables a and b within the field $\mathbb{F}_p$. The elliptic curve primarily includes all points (x, y) which satisfy the elliptic curve equation modulo p (where x and y are numbers in $\mathbb{F}_p$). The equation is typically written in the short Weierstrass form

$$y^2 = x^3 + ax + b \bmod p,$$

where $a, b \in F_p$ satisfy $4a^3 + 27b^2 \bmod p$ is not 0, which guarantees $x^3 + ax + b$ contains no repeated factors and then the elliptic curve can be used to form a group. The elliptic curve contains all points $P = (x, y)$ for $x, y \in F_p$ that satisfy the elliptic curve equation with addition of a special point ∞ known as the point at infinity[2].

[2] In code implementation, ∞ is normally represented as point (0,0), but not always, as (0,0) might be on the curve.

The elliptic curve used in Bitcoin is called secp256k1. Secp256k1 curve is proposed in Certicom [7] in addition to NIST curve for 256 bits prime. It is defined over prime field $\mathbb{F}_p$, where

$$p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1.$$

The curve equation E is $y^2 = x^3 + ax + b$, where $a = 0$ and $b = 7$.

2.1.1. Key Pair Generation

An elliptic curve key pair is associated with a particular set of valid domain parameters [9]. Let E be an elliptic curve defined over a finite field $\mathbb{F}_p$. Let P be a point in $E(\mathbb{F}_p)$, and suppose that P has prime order n. Then the cyclic subgroup of $E(\mathbb{F}_p)$ generated by P is

$$\langle P \rangle = \{\infty, P, 2P, 3P, \ldots, (n-1)P\}.$$

The prime p, the equation of the elliptic curve E, the point P and its order n are the public domain parameters. A private key is an integer d that is selected uniformly at random from the interval $[1, n-1]$, and the corresponding public key $Q = dP$.

Algorithm 1 Key pair generation [9]

```
Input: Domain parameters $D = (p, E, P, n)$.
Output: Public key $Q$, private key $d$.
1: Select $d \in_R [1, n-1]$.
2: Compute $Q = dP$.
3: Return $(Q, d)$.
```

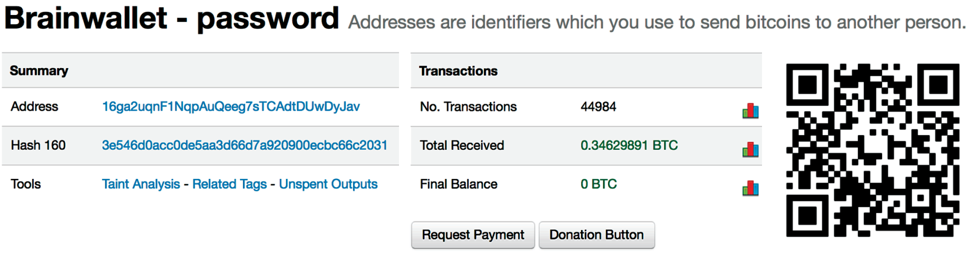
Note that the process of computing a private key d given public key Q is exactly the elliptic curve discrete logarithm problem (ECDLP). Hence it is very important to choose a set of domain parameters so that the ECDLP is hard to solve. In addition the number d should be **random** in the sense that it should have large entropy AND there should be no way to distinguish a source which produces these values from a source which generates them uniformly at random. In particular the min-entropy should also be high and there should be no efficient guessing strategy of any sort.

2.2. Brain Wallet

A Bitcoin wallet is a collection of Bitcoin addresses and stores the corresponding keys for those addresses. Bitcoin wallets come in different forms, including desktop software, mobile apps, online services, hardware, smart card and paper.

As we discussed earlier in Section 2.1.1, the private key is a number which we presume to be totally random. Normally the private key will be a long hex string which is very hard for a person to remember and store safely. No matter what form of wallet you are using, there always exists a chance that you might lose your wallet in a cybersecurity breach.

Brain wallets are another solution, which do not need the users to keep anything in safe and still be able to recover their private key. Instead of storing the private key and protecting it, one can store it in a human mind. A brain wallet creates private key from a (typically) human chosen password or a passphrase, using the SHA-256 hash algorithm and converts it into a 256-bit number. As SHA-256 is deterministic method, users can always use the same password to recreate their private key. Note that since brain wallets use the hash directly as the private key, the security of storing private keys now depends only on how unpredictable the passwords are.



Brainwallet - password Addresses are identifiers which you use to send bitcoins to another person.

Summary	
Address	16ga2uqnF1NqpAuQeeg7sTCAdtDUwDyJav
Hash 160	3e546d0acc0de5aa3d66d7a920900ecbc66c2031
Tools	Taint Analysis - Related Tags - Unspent Outputs

Transactions	
No. Transactions	44984
Total Received	0.34629891 BTC
Final Balance	0 BTC

FIGURE 1. Brain wallet generated by password “password”.

3. Related Work

We are not the first to try to crack Bitcoin brain wallets, some hackers have been there before. Many victims have found their money stolen and posted it in forums. The first ethical/research brain wallet cracker was announced publicly in a recent hacking conference DEF CON 23 (Aug 2015). Ryan Castellucci, a whitehat hacker presented his research on cracking brain wallets, and also published his software [6]. Ryan’s attack was done on an Intel i7 PC with 4 physical cores and 8 logical cores due to Hyper-Threading. The attack speed can reach approximately 16,250 password per second on each thread and he had cracked more than 18,000 brain wallet addresses.

The software Ryan has published uses an existing open source secp256k1 bitcoin elliptic curve implementation mainly written by Pieter Wuille, one of Bitcoin core developers. This implementation is widely used in Bitcoin clients and is considered the current best in terms of code level optimisation (detailed benchmarks are given in Table 2).

Later Vasek, Bonneau, Keith, Castellucci and Moore published their cybercrime analysis results on brain wallets addresses cracked using Ryan's software implementation in FC 2016 [15]. Their work was more focused on brain wallets usage measurements and did not try to improve the speed of the attack.

4. Special designed point multiplication method for attack

The process of cracking Bitcoin brain wallets is to repeatedly generate public keys using guessed passwords. Key generation method as we described in Section 2.1.1, is to compute $Q = dP$. Here d is a SHA256 hash of the generated password, P is a fixed point which is the base point G for secp256k1. We first benchmark the current best implementation, libsecp256k1. All benchmark results are running on an Intel i7-3520m 2.9 GHz laptop (win8 x 64).

The time cost for computing one public key given a random private key takes: **47.2** μs.

4.1. Fixed point multiplication methods

The most basic and naive method for point multiplication $Q = kP$ with an unknown point P is double-and-add method [9]. The idea is to use binary representation for k:

$$k = k_0 + 2k_1 + 2^2k_2 + \cdots + 2^m k_m,$$

where $[k_0 \dots k_m] \in \{0, 1\}$ and m is the length of k, in bitcoin elliptic curve, $m = 256$.

Algorithm 2 double-and-add method for point multiplication for input points P not known in advance. [9]

```
INPUT: $k = (k_m, \dots, k_1, k_0)_2$, $P \in E(\mathbb{F}_q)$.
OUTPUT: $kP$.
Q := infinity
for $i$ from 0 to $m$ do
    if $k_i = 1$ then $Q := Q + P$ (using point addition)
    $P := 2P$ (using point doubling)
end for
return $Q$
```

The expected number of ones in the binary representation of k is approximately $\frac{m}{2}$, so double-and-add method will need $\frac{m}{2} + mD$ computations in total. However, if the point P is fixed and some storage is available, then the point multiplication operation $Q = kP$ can be accelerated by pre-computing some data that depends only on P. For example if the points $2P, 2^2P, \dots, 2^{m-1}P$ are pre-computed, then the double-and-add method (algorithm 2) has expected running time $(\frac{m}{2})A$, and all doublings are eliminated.

In [3] the authors introduced a new method for fixed point multiplication. The pre-computing step stores every multiple 2^iP. Let $(K_{d-1}, \dots, K_1, K_0)_{2^w}$ be the base-2^w representation of k, where $d = [m/w]$, and let $Q_j = \sum_{i:K_i=j} 2^{wi}P$ for each j, $1 \leq j \leq 2^w - 1$, Then

$$\begin{aligned} kP &= \sum_{i=0}^{d_1} K_i(2^{wi}P) = \sum_{j=1}^{2^w-1} (j \sum_{i:K_i=j} 2^{wi}P) = \sum_{j=1}^{2^w-1} jQ_j \\ &= Q_{2^w-1} + (Q_{2^w-1} + Q_{2^w-2}) + \cdots \\ &\quad + (Q_{2^w-1} + Q_{2^w-2} + \cdots + Q_1). \end{aligned}$$

By reviewing the literature and checking some other existing methods in [9] we noticed they are all memory friendly implementations which do not take a lot of memory space for precomputation. However, we are working on a different task and aim to repeatedly run point multiplication method for great many times. We have implemented an extreme version of window method which will take much more precomputation space than methods introduced in [9].

In our implementation, the precomputation step will compute $P_j = jP$, where $1 \leq j \leq 2^w - 1$, then for each P_j we compute $P_{i,j} = 2^{wi}P_j$, which will cost $2^w - 1$ times more memory space than [3], [9], but expected running time for each point multiplication is reduced down to approximately $(d - 1)A$.

Algorithm 3 Our windowed method with larger precomputation table

```
Input: Window width $w$, $d = [m/w]$, $k = (K_{d-1}, \dots, K_1, K_0)_{2^w}$.
Output: $kP$.
Precompute $P_{i,j} = 2^{wi}jP, 0 \leq i \leq d-1$ and $1 \leq j \leq 2^w - 1$
$A \leftarrow$ infinity
for $i$ from 0 to $d - 1$ do
    $A \leftarrow A + P_{i,j}$ where $j = K_i$
end for
return $A$
```

We have implemented a code that can take any window width w. Results and corresponding memory usages based on different window size are shown in Table 1.

Table 1. Time cost for different window width w, point addition method secp256k1 library [14] secp256k1_gej_add_ge.

	w=4	w=8	w=12	w=16	w=20
d	64	32	22	16	13
# of additions	63	31	21	15	12
memory	81.92 KB	655.36 KB	7.21 MB	83.89 MB	1.09 GB
time cost	46.36 μs	22.76 μs	15.35 μs	11.23 μs	9.23 μs

Table 2. Benchmarking openssl and MPIR library for field multiplication, square and modular inverse in affine coordinate.

	multiplication	mod p	square	mod p	mod inverse
MPIR	0.07 μs	0.15 μs	0.13 μs	0.15 μs	1.8 μs
openssl	0.08 μs	0.43 μs	0.06 μs	0.43 μs	18.0 μs
secp256k1	0.049 μs		0.039 μs		1.1 μs

4.2. Point representation

Representing a point as an affine coordinate $P(x, y)$ on an elliptic curve over $\mathbb{F}_p$, the field operations required for one EC point addition are: two multiplications, one square and one modular inverse (for short, 2M+1S+1I). Modular inverse is more expensive operation compared to multiplication and square. We list our benchmarks using different packages in C to demonstrate the difference for modular inverse computation compared to multiplication and square. The packages we have benchmarked are: openssl-1.0.2a (released in March 2015) and mpir-2.5.2 (released in October 2012), and Pieter Wuille's implementation on Github [14] [3] .

The results are shown in Table 2. The benchmarking shows modular inverse is much more expensive than multiplication and squaring. It is also important to notice, for MPIR big number library, the square operation is more expensive than multiplication, and for openssl library, 1 square = 0.75 multiplication. As modular inverse is more expensive than multiplication, it may be advantageous to represent points using other coordinates.

[3] with the following configuration:
USE_NUM_GMP USE_FIELD_10x26 USE_FIELD_INV_NUM USE_SCALAR_8x32 USE_SCALAR_INV_BUILTIN

4.2.1. Projective coordinates

For elliptic curve over $\mathbb{F}_p$, where the curve equation is $y^2 = x^3 + ax + b$. The standard projective coordinates represent elliptic curve points as $(X : Y : Z)$, $Z \neq 0$, correspond to the affine point $(\frac{X}{Z}, \frac{Y}{Z})$. The projective equation of the elliptic curve is

$$Y^2Z = X^3 + aXZ^2 + bZ^3.$$

The point at infinity ∞ corresponds to $(0 : 1 : 0)$, where the negative of $(X : Y : Z)$ is $(X : -Y : Z)$.

4.2.2. Jacobian coordinates

Elliptic curve points in Jacobian coordinates are represented in the following format $(X : Y : Z)$, $Z \neq 0$, which corresponds to the affine point $(\frac{X}{Z^2}, \frac{X}{Z^3})$. The projective equation of the elliptic curve is

$$Y^2 = X^3 + aXZ^4 + bZ^6.$$

The point at infinity ∞ corresponds to $(1 : 1 : 0)$, while the negative of $(X : Y : Z)$ is $(X : -Y : Z)$.

The field operations needed for point addition and doubling are shown in Table 3. We see that Jacobian coordinates yield the fastest point doubling, while mixed Jacobian-affine coordinates yield the fastest point addition.

Table 3. Operation counts for point addition and doubling. A = affine, P = standard projective, J = Jacobian [5, 9].

Doubling		General addition		Mixed coordinates*	
2A → A	1I,2M,2S	A+A → A	1I,2M,1S	J+A → J	8M,3S
2P → P	7M,3S	P+P → P	12M,2S		
2J → J	4M,4S	J+J → J	12M,4S		

* Here mixed coordinates means Jacobian-Affine mixed coordinates

We refer the reader to [9], [5] for other detailed equations in different coordinates. Here we only interested in point addition functions using mixed coordinates.

4.2.3. Secp256k1 point addition formulas

In the latest version, secp256k1 point addition formulas are based on [4] which introduced strongly unified addition formulas for standard projective coordinates. Bitcoin developers implemented a mixed coordinate formula (Jacobian-Affine) version based on [4].

Let $P = (X_1 : Y_1 : Z_1)$ be a Jacobian projective point on elliptic curve $y^2 = x^3 + ax + b$, and $Q = (X_2 : Y_2 : 1)$ be another point on the curve, suppose that $P \neq \pm Q$, $P + Q = (X_3 : Y_3 : Z_3)$ is computed by the following equations:

$$\begin{aligned} X_3 &= 4(K^2 - H), \\ Y_3 &= 4\left(R(3H - 2K^2) - G^2\right), \\ Z_3 &= 2FZ_1, \end{aligned} \tag{1}$$

where

$$\begin{array}{llll} A = Z_1^2, & B = Z_1 \cdot A, & C = X_2 \cdot A, & D = Y_2 \cdot B, \\ E = X_1 + C, & F = Y_1 + D, & G = F^2, & H = E \cdot G, \\ I = E^2, & J = X_1 \cdot C, & K = I - J. & \end{array}$$

4.2.4. Bernstein-Lange point addition formulas

In [2], Bernstein introduced the following method which takes 7M+4S using Jacobian-Affine coordinate, the explicit formulas are given as follows [1]

$$\begin{aligned} X_3 &= r^2 - J - 2V, \\ Y_3 &= r \cdot (V - X_3) - 2Y_1 \cdot J, \\ Z_3 &= (Z_1 + H)^2 - Z_1^2 - H^2, \end{aligned} \tag{2}$$

where

$$\begin{array}{llll} U2 = X_2 \cdot Z_1^2, & S2 = Y_2 \cdot Z_1^3, & H = U2 - X_1, & \\ I = 4H^2, & J = H \cdot I, & r = 2(S2 - Y_1), & V = X_1 \cdot I. \end{array}$$

4.3. Detailed field operation benchmarks

From the results of table 2 we saw that Wuille's secp256k1 library [14] has much faster field multiplication and square speed than openssl and mpir library. Wuille's field implementation is optimised based on the special prime used in secp256k1 curve. Libsecp256k1 has 5x52 and 10x26 field implementations for 64 bits and 32 bits integers [4]. Here we use the 10x26 representation and each 256 bit value is represented as a 32 bit integer array with size of 10. We refer readers to file *field_10x26_impl.h* in libsecp256k1 for more details. Libsecp256k1 already implemented the equation from [1], [9] in method *secp256k1_gej_add_ge_var*, which uses 8 multiplications, 3 squares, 1 multiply integer, 6 additions and 5 negations. Equation 1 is implemented in another method called *secp256k1_gej_add_ge*, which uses 7 multiplications, 5 squares and 5 multiply integer, 7 additions, 3 negations and 6 fe_cmov operations. We have implemented equation 2 which takes 7 multiplications, 4 squares, 4 multiply integer, 9 additions and 8 negations.

[4] Depends on whether compiler and target support 64 bit integers.

It is important to notice the squaring and multiplication differences we discussed in Table 2. In [1] Bernstein listed the best operation counts based on different assumptions: S = 0 M, S = 0.2 M, S= 0.67 M, S=0.8 M and S=1 M. In [8], the author showed that the ratio S/M is almost independent of the field of definition and of the implementation, and can be reasonably taken equal to 0.8. Our benchmark results is very similar to S = 0.8 M. In [1], other field operations are considered as 0 M, in Table 4 our benchmark results shows field addition and other operations have approximately 0.1 M cost.

Table 4. Field operation counts and benchmark results.

	#Mul 1M	#Square ≈ 0.8 M	#add/neg/*int ≈ 0.1 M	#fe_cmov ≈ 0.2 M	Total cost
secp256k1_gej _add_ge	7	5	15	6	≈ 0.681 μs
secp256k1_gej _add_ge_var	8	3	12	0	≈ 0.562 μs
7M + 4S code	7	4	21	0	≈ 0.594 μs

The secp256k1_gej_add_ge method which is also the default method for key generation, uses 6 secp256k1_fe_cmov operations which has a cost approximately 0.2 M. The rationale for writing code in this way is stated by Wuille in the following comment:

"This formula has the benefit of being the same for both addition of distinct points and doubling" [14]

The purpose of making addition and doubling using the same function is to prevent side channel attacks, as point doubling is otherwise much cheaper than point addition. Our experiments are done based on the benchmark results of S/M ratio with specified machine setting (earlier in Section 4) and specific library configuration (footnote in Section 4.2). Different operating systems or library configurations lead to different results. One should choose between our code and secp256k1_gej_add_ge method. Detailed benchmark results are given in Table 5.

The code published as part of DEF CON attack on github in August 2015 [6] uses a faster version of secp256k1 library[5], and the results are marked as "*" in Table 5. Our best result using 1.09 GB precomputation memory gives ≈ **2.5 times speed up** for key generation process than the current known best attack.

[5] Also written by Pieter Wuille one year ago, this version is performance focused and using 8M+3S.

Table 5. Time cost for different window width w for EC key generation.

	w=4	w=8	w=12	w=16	w=20
d	64	32	22	16	13
# of additions	63	31	21	15	12
precomputation memory	81.92 KB	655.36 KB	7.21 MB	83.89 MB	1.09 GB
secp256k1_gej _add_ge	45.85 μs	22.16 μs	15.35 μs	11.23 μs	9.23 μs
secp256k1_gej _add_ge_var	**37.37** μs*	17.86 μs	12.21 μs	8.89 μs	**7.16** μs
7M + 4S code	39.01 μs	18.79 μs	12.77 μs	9.23 μs	7.48 μs
Jacobian to Affine	$\approx$ 10 μs				
Benchmark on my laptop	$\approx$ 42 K guesses/sec (single thread) on i7-3520m 2.9 GHz CPU				
DEF CON Attack**	$\approx$ 130 K guesses/sec on i7-2600 3.2 GHz CPU				
Our improved DEF CON Attack	$\approx$ 315 K guesses/sec				

* The main results reported DEF CON attack [6] are equivalent to these **37.37** μs.

** These are reported by Ryan Castellucci running his DEF CON code,which are then compared to our improved code on 8 threads with linux gcc compiler.

In theory the best point addition method is 7M+4S introduced in [2]. However in practice, when field multiplication and square are well optimised, other field operations (such as addition, negation) become more significant than theoretical value, see Table 4. Our results show that for our laptop specification, 8M+3S method is overall better than 7M+4S.

In order to compare the results with DEF CON attack, we also benchmark our implementation against the DEF CON software release on Amazon server. Experiments are done on an m4.4xlarge Amazon EC2 instance[6]. Results are shown in Table 6. The results confirm a 2.5 times improvement. Note that when running on Amazon EC2 (Intel Haswell CPU), the theoretical best method (7M+4S) performs a little bit better than 8M+3S.

Based on the current price for Amazon EC2 service, we observe the following cost for implementing such brain wallet attack: 17.9 billion passwords check per US dollar; 55.86 dollars to check a trillion passwords. We have found more than

[6] https://aws.amazon.com/ec2/instance-types/

Table 6. Benchmark with DEF CON results on Amazon EC2 instance.

	processes	passwords per second
brainflayer (DEF CON)	16	219,460
win size 20 8M+3S	16	533,196
win size 24 8M+3S	16	556,294
win size 24 7M+4S	16	558,449

18,000 passwords using this tool. Some sample passwords, including some quite difficult ones are listed in Appendix A.

5. Conclusion

In this paper we have analysed and improved the state of the art on the implementation of the secp256k1 elliptic curve and similar curves. We provide the first benchmarks on existing implementations and provide a faster implementation for specific applications where private keys are not manipulated or there exist other protections against side channel attacks (e.g., physical and electromagnetic isolation) and when larger amounts of RAM are available. For example we are able to examine passwords in brain wallets 2.5 times faster than the state of the art implementation presented at DEF CON few months earlier. We have released our source code.

As an example application of this research, we have been able to crack thousands of passwords including some quite difficult ones. Our research demonstrates again that brain wallets are not secure and no one should use them.

REFERENCES

[1] BERNSTEIN, D. J.—LANGE, T.: *Explicit-formulas database*, 2007, `https://hyperelliptic.org/EFD/`

[2] BERNSTEIN, D. J.—LANGE, T.: *Faster addition and doubling on elliptic curves*, in: Advances in cryptology–ASIACRYPT '07, Lecture Notes in Comput. Sci., Vol. 4833, Springer-Verlag, Berlin, 2007, pp. 29–50.

[3] BRICKELL, E. F.—GORDON, D. M.—MCCURLEY, K. S.—WILSON, D. B.: *Fast exponentiation with precomputation*, in: Advances in Cryptology-EUROCRYPT '92, Springer-Verlag, Berlin, 1993, pp. 200–207.

[4] BRIER, E.—JOYE, M.: *Weierstraß elliptic curves and side-channel attacks*, in: International Workshop on Public Key Cryptography—PKC '02, Springer-Verlag, 2002, pp. 335–345.

[5] BROWN, M.—HANKERSON, D.—LÓPEZ, J.—MENEZES, A.: *Software implementation of the NIST elliptic curves over prime fields*, in: Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA, April 08–12, 2001, CT-RSA '01, London, UK; Lecture Notes in Comput. Sci., Vol. 2020, Springer-Verlag, 2001. pp.250–265.

[6] CASTELLUCCI, R.: *Cracking cryptocurrency brainwallets*, `https://www.defcon.org/html/defcon-23/dc-23-index.html`

[7] CERTICOM RESEARCH: *Sec 2: Recommended elliptic curve domain parameters*, in: Proceeding of Standards for Efficient Cryptography, Version 1, 2000. `www.secg.org/SEC2-Ver-1.0.pdf`

[8] COHEN, H.—MIYAJI, A.—ONO, T.: *Efficient elliptic curve exponentiation using mixed coordinates*, in: Advances in Cryptology, ASIACRYPT '98 (Beijing), Lecture Notes in Comput. Sci., Vol. 1514, Springer-Verlag, Berlin, 1998, pp. 51–65.

[9] HANKERSON, D. —MENEZES,A . J. —VANSTONE, S.: *Guide to Elliptic Curve Cryptography*, Springer Science & Business Media, 2006.

[10] KOBLITZ, N.: *Elliptic curve cryptosystems*, Mathematics of computation, **48** (1987), no.177, 203–209.

[11] MILLER, V. S.: *Use of elliptic curves in cryptography*, in: Proc. Advances in Cryptology––CRYPTO '85 (Santa Barbara, Calif., 1985), Lecture Notes in Comput. Sci., Vol. 218, Springer-Verlag, Berlin, 1986, 417–426.

[12] NAKAMOTO, S.: *Bitcoin: A peer-to-peer electronic cash system*, `https://bitcoin.org/bitcoin.pdf`, 2008.

[13] RIVEST, R. L.—SHAMIR, A.—ADLEMAN, L.: *A method for obtaining digital signatures and public-key cryptosystems*, Commun. ACM **21** (1978), no. 2, 120–126.

[14] WUILLE, P.: *bitcoin secp256k1 library, version 2015/08/11*, `https://github.com/bitcoin/secp256k1`

[15] VASEK, M.—BONNEAU, J.—KEITH, C.—CASTELLUCCI, R.—MOORE, T.: *The Bitcoin brain drain: A short paper on the use and abuse of Bitcoin brain wallets,* Financial Cryptography and Data Security, Lecture Notes in Comput. Sci., Springer--Verlag, Berlin, 2016.

Appendix A. Cracked Password Samples

More than 100 new passwords never discovered before were found (using our open source tool) by UCL M.Sc. Information Security students during our "code breaking competition" run as a part of GA18 Cryptanalysis course in March 2016. These students are Iason Papapanagiotakis-Bousy, Ilyas Azeem, Jeonghyuk Park, Ellery Smith, Weixiu Tan and Wei Shao.

(1) say hello to my little friend
(2) to be or not to be
(3) Walk Into This Room
(4) party like it's 1999
(5) yohohoandabottleofrum
(6) dudewheresmycar
(7) dajiahao
(8) hankou
(9) {1summer2leo3phoebe
(10) 0racle9i
(11) andreas antonopoulos
(12) Arnold Schwarzenegger
(13) blablablablablablabla
(14) for the longest time
(15) captain spaulding

Received December 2, 2016

Department of Computer Science
University College London
Gower Street
London
WC1E 6BT
UNITED KINGDOM

Department of Optimization
Satalia (NPComplete Ltd)
97 Tottenham Court Rd
London
W1T 4TP
UNITED KINGDOM

Department of Computer Science
University College London
Gower Street
London
WC1E 6BT
UNITED KINGDOM

White Ops
11 8th Avenue
New York
NY 10011
USA

E-mail: n.courtois@ucl.ac.uk
g.song@cs.ucl.ac.uk
pubs@ryanc.org