Speed Optimizations in Bitcoin Key Recovery Attacks

Nicolas Courtois
University College London
n.courtois@ucl.ac.uk

Guangyan Song
University College London
g.song@cs.ucl.ac.uk

Ryan Castellucci
White Ops
pubs@ryanc.org

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 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 have we have cracked, showing that brain wallets are not secure in practice even for quite complex passwords.

Keywords

Bitcoin, Elliptic Curve Cryptography, Crypto Currency, Brain Wallet

1. INTRODUCTION

Bitcoin is a cryptocurrency, an electronic payment system based on cryptography. It was created by Satoshi Nakomoto[1] in 2008 [13]. 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 cosponsoring private key. Once the private is lost, the bitcoin network will not recognize any other evidence of ownership.

Bitcoin uses digital signature protect the ownership bitcoin and private key is the only evidence of owning bitcoin. Thus it is very important to look at the technical details of the digital signature scheme used in bitcoin.

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

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[11] and Victor Miller [12] in 1985. It is a public-key cryptography protocol where each of the participant 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 [14].

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 includes all points (x, y) which satisfy the elliptic curve equation modulo p (where x and y are numbers in $\mathbb{F}_p$). It is typically defined in the short Weierstrass form:

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

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 and together with a special point ∞ call the point at infinity [2].

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 [10]. 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 [10]

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 chose 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 turn 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.

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

3. RELATED WORK

We are not the first ones try to crack Bitcoin brain wallets, a lot of other security researchers are doing it. 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 hyper-threaded cores. 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 et al. published their cybercrime analysis results on brain wallets addresses cracked using Ryan's software implementation in FC 2016. Their work were 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 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 a laptop with the following specifications:

- CPU: Intel i7-3520m 2.9GHz
- RAM: 4G
- OS: 64-bit Windows 8

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

4.1 Fixed Point Multiplication Methods

The most basic and naive method for point multiplication $Q = kP$ with a unknown point P is double-and-add method [10]. The idea is to use binary representation for k:

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

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

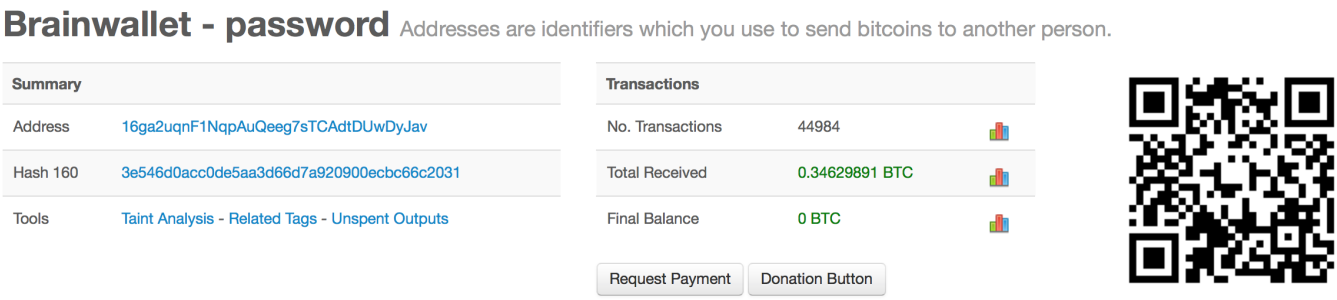

Figure 1: Brain wallet generated by password "password"

Algorithm 2 double-and-add method for point multiplication of unknown points[10]

1: $Q :=$ infinity
2: **for** i from 0 to m **do**
3: if $k_i = 1$ then $Q := Q + P$ (using point addition)
4: $P := 2P$ (using point doubling)
5: **end for**
6: **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 precomputing some data that depends only on P. For example if the points $2P, 2^2P, \ldots, 2^{m-1}P$ are precomputed, 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 precomputing step stores every multiple 2^iP. Let $(K_{d-1}, \ldots, K_1, K_0)_{2^w}$ 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

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

By reviewing the literature and checking some other existing methods in [10] 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 [10].

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, 10], but expected running time for each point multiplication will reduce to approximately $(d-1)A$

Algorithm 3 Our implementation of windowing method with larger precomputation table

INPUT: Window width w, $d = [m/w]$, $k = (K_{d-1}, \ldots, K_1, K_0)_{2^w}$
OUTPUT: kP

1: Precompute $P_{i,j} = 2^{wi}jP, 0 \leq i \leq d-1$ and $1 \leq j \leq 2^w - 1$
2: $A \leftarrow$ infinity
3: **for** i from 0 to $d-1$ **do**
4: $A \leftarrow A + P_{i,j}$ where $j = K_i$
5: **end for**
6: **return** A

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

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 for calculating point addition need 2 multiplications, 1 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 Oct 2012), and the Pieter Wuille's implementation on Github [15] [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.

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

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

Table 1: Time cost for different window width w, point addition method secp256k1 library [15] secp256k1_gej_add_ge

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

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 us	0.15 us	0.13 us	0.15 us	1.8 us
openssl	0.08 us	0.43 us	0.06 us	0.43 us	18.0 us
secp256k1	0.049 us		0.039 us		1.1 us

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 coordinate are represented in the following format $(X : Y : Z)$, $Z \neq 0$, 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.

We refer the reader to [10, 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:

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

where

$$A = Z_1^2,\ B = Z_1 \cdot A,\ C = X_2 \cdot A,\ D = Y_2 \cdot B,\ E = X_1 + C$$

$$F = Y_1 + D, G = F^2, H = E \cdot G, I = E^2, J = X_1 \cdot C, K = I - J$$

4.2.4 Bernstein-Lange point addition formulas

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

$$\begin{aligned} X_3 &= r^2 - J - 2V \\ Y_3 &= r \cdot (V - X_3) - 2Y_1 \cdot J \\ Z_3 &= (Z_1 + H)^2 - Z_1^2 - H^2 \end{aligned} \tag{3}$$

where

$$U2 = X_2 \cdot Z_1^2,\ S2 = Y_2 \cdot Z_1^3$$
$$H = U2 - X_1,\ I = 4H^2$$
$$J = H \cdot I,\ r = 2(S2 - Y_1),\ V = X_1 \cdot I$$

4.3 Detailed Field Operation Benchmarks

From the results of table 2 we saw that Wuille's secp256k1 library [15] 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, 10] in method *secp256k1_gej_add_ge_var*, which uses 8 multiplications, 3 squares and 12 multiply integer / addition / negation. Equation 2 is implemented in another method called *secp256k1_gej_add_ge*, which uses 7 multiplications, 5 squares and 21 multiply integer / addition / negation. We have implemented equation 3 which takes 7 multiplication, 4 squares and 22 multiply integer / addition / negation.

It is important to notice the squaring and multiplication differences we discussed in table 2. In [9] Bernstein listed the best operation counts based on different assumptions: S = 0M, S = 0.2M, S= 0.67M, S=0.8M and S=1M. 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

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

Table 3: Operation counts for point addition and doubling. A = affine, P = standard projective, J = Jacobian [10, 5]

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

very similar to S = 0.8M. In [1], other field operations are considered as 0M, in table 4 our benchmark results shows field addition and other operations have approximately 0.1M cost.

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"[15]

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 might have different results. One should choose between our code and secp256k1_gej_add_ge method. Detailed benchmark results are given in table 5

DEF CON attack [6] published code on github in August 2015 uses a faster version of secp256k1 library [5], and the results is 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.

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 better than 7M+4S.

In order to compare the results with DEF CON attack, we also benchmark our implementation and the DEF CON released software 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 18,000 passwords using this tool. Some sample passwords, including some quite difficult ones are listed in appendix A.

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

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

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 electro-magnetic 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 2 months ago. 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.

6. REFERENCES

[1] D. J. Bernstein and T. Lange. Explicit-formulas database, 2007.

[2] D. J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In *Advances in cryptology–ASIACRYPT 2007*, pages 29–50. Springer, 2007.

[3] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation. In *Advances in Cryptology-EUROCRYPT'92*, pages 200–207. Springer, 1993.

[4] E. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. In *Public Key Cryptography*, pages 335–345. Springer, 2002.

[5] M. Brown, D. Hankerson, J. López, and A. Menezes. *Software implementation of the NIST elliptic curves over prime fields*. Springer, 2001.

[6] R. Castellucci. Cracking cryptocurrency brainwallets. https://www.defcon.org/html/defcon-23/dc-23-index.html.

[7] S. Certicom. Sec 2: Recommended elliptic curve domain parameters. *Proceeding of Standards for Efficient Cryptography, Version*, 1, 2000.

[8] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *Advances in Cryptology ASIACRYPT 98*, pages 51–65. Springer, 1998.

[9] T. L. Daniel J. Bernstein. Explicit-formulas database. https://www.hyperelliptic.org/EFD/.

[10] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to elliptic curve cryptography*. Springer Science &

Table 4: Field operation counts and benchmark results

	#Multiplication 1M	#Square ≈ 0.8 M	#add/neg/*int ≈ 0.1 M	#fe_cmov ≈ 0.2 M	total time cost
secp256k1_gej_add_ge	7	5	15	6	≈ 0.681 us
secp256k1_gej_add_ge_var	8	3	12	0	≈ 0.562 us
7M + 4S code	7	4	21	0	≈ 0.594 us

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
number 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 us	22.16 us	15.35 us	11.23 us	9.23 us
secp256k1_gej_add_ge_var	**37.37 us***	17.86 us	12.21 us	8.89 us	**7.16 us**
7M + 4S code	39.01 us	18.79 us	12.77 us	9.23 us	7.48 us
covert Jacobian to Affine	≈ 10 us				
Benchmark on my laptop i7-3520m 2.9 GHz CPU	≈ 42 K guesses / sec (single thread)				
DEF CON Attack** i7-2600 3.5 GHz CPU	≈ 130 K guesses / sec				
Improved DEF CON attack**	≈ 315 K guesses / sec				

* DEF CON attack [6] is equivalent to this result

** Results are reported by Ryan Castellucci running his DEF CON code and our improved code on 8 threads with linux gcc compiler.

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

Business Media, 2006.

[11] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[12] V. S. Miller. Use of elliptic curves in cryptography. In *Advances in CryptologyâĂŤCRYPTOâĂŹ85 Proceedings*, pages 417–426. Springer, 1985.

[13] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1(2012):28, 2008.

[14] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[15] P. Wuille. bitcoin secp256k1 library, version 2015/08/11. https://github.com/bitcoin/secp256k1.

APPENDIX

A. CRACKED PASSWORD SAMPLES

Our open source tool was given to students for our code breaking competition, more than 100 new passwords were found by master students: Iason Papapanagiotakis, 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