Satoshi's Genius: Unexpected Ways in which Bitcoin Dodged Some Cryptographic Bullets

by **Vitalik Buterin** 10.28.13 5:49 PM

As far as open-source protocols go, one area in which Bitcoin is unique is the sheer difficulty of making any changes to the protocol. Unlike most other protocols, where features can be added, modified or deprecated at a moment's notice, in the world of Bitcoin even the slightest change requires the simultaneous cooperation of the vast majority of the entire Bitcoin network. The reason for this is simple: in Bitcoin, and in Bitcoin alone, absolute consensus is required. On internet protocols like HTML and CSS, if a web browser interprets some style elements incorrectly, the worst that can happen is that the webpage renders incorrectly. In Bitcoin, on the other hand, a single transaction being incorrectly valid or invalid makes the entire block invalid, potentially causing the entire network to split in half as it did in March 2013. As a result, most of the decisions that Satoshi Nakamoto made in 2008 we are essentially stuck with. Although Satoshi's choices were by no means perfect, fortunately it appears that he has been right more often than not; in fact, there are several instances in which we are all better off for the choices Satoshi made for reasons that even he did not imagine.

Addresses as hashes of public keys

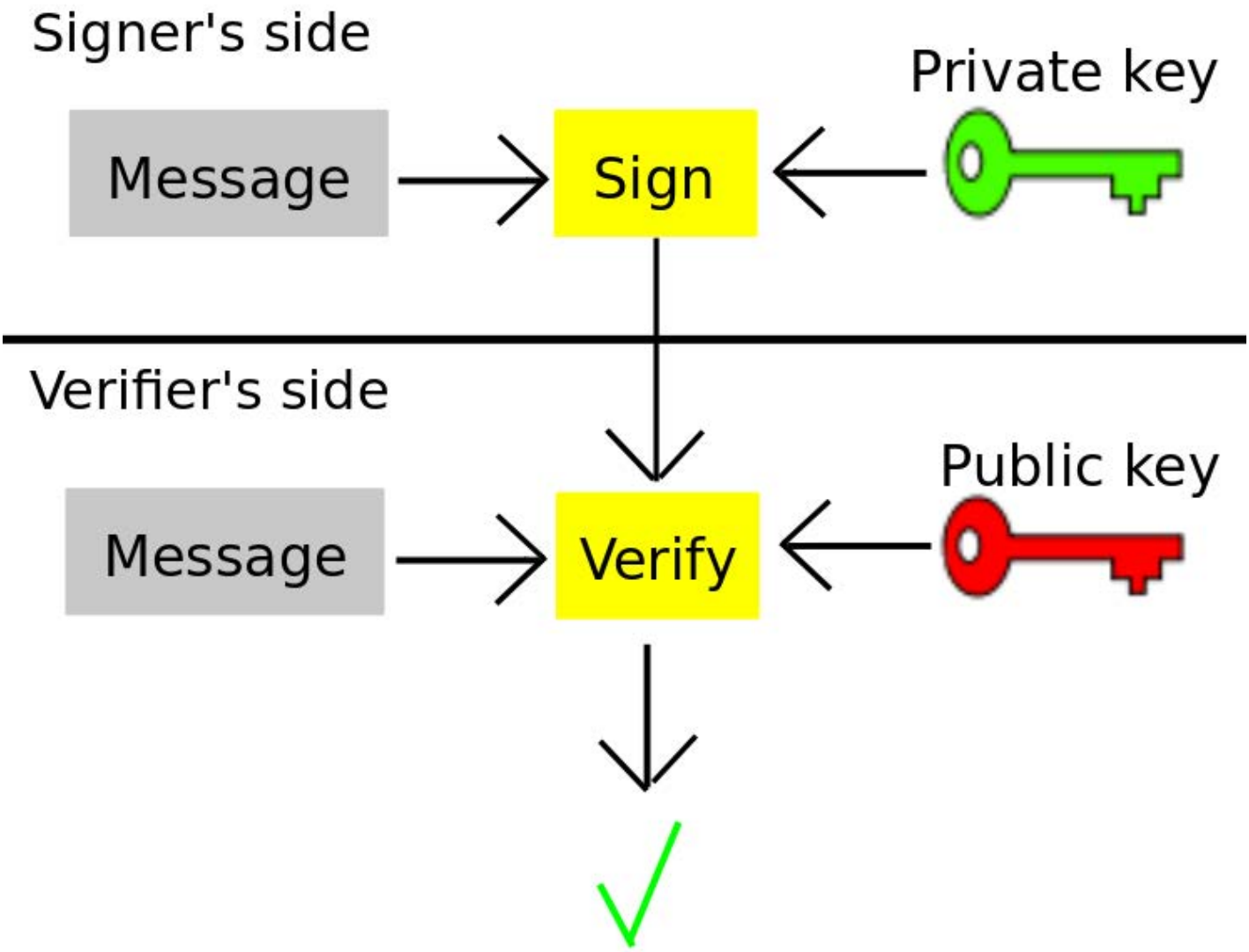
One of the interesting, and to many at first slightly confusing, aspects of Bitcoin is the precise relationship between private keys and Bitcoin addresses. It is widely understood that you send transactions to a "Bitcoin address", and claiming money sent to your address requires you (or rather, your Bitcoin wallet) to create a transaction containing a digital signature made with the corresponding "private key". Bitcoin addresses can be safely handed out publicly, whereas private keys need to be stored securely in an encrypted Bitcoin wallet. For example, here is a randomly generated private key:

```
5JexEhxcSeADA4MmpHq426zC2935JHKc2de1nphv75u8TRhAqqP
```

And its associated address:

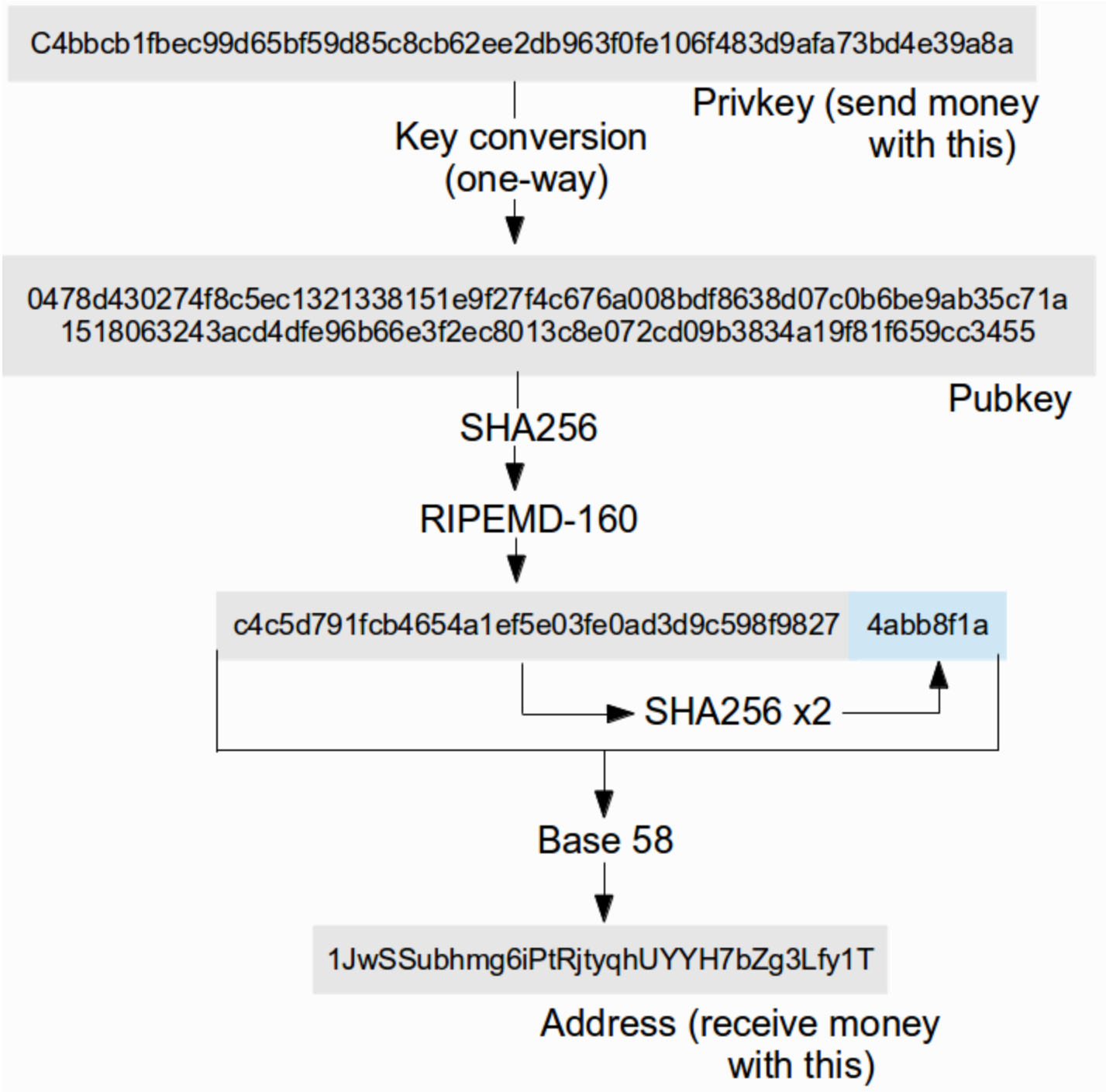
```
1MEMPfCcqatuWxHWFsFcyhDTq3eV61xLB4
```

But what is the relationship between these two values? The answer lies in a branch of mathematics known as public key cryptography. Although the term cryptography is usually associated with the art of securely sending secrets, public key cryptography is actually even more often employed for another purpose: authentication. For example, suppose that a software company wants to send an update to all of its users, but wants to do so securely, so that others cannot publish fake updates containing computer viruses. With public key cryptography, the software company can create a *key pair*, consisting of a private key and a public key, and include the public key in the software. When the company sends out an update, it *digitally signs* the update using a cryptographic digital signature algorithm with its private key, and the clients, upon receiving the update, can verify the validity of the signature using their copy of the public key. If an attacker, without knowledge of the software company's secret private key, tried to make a fake update, the signature would fail to verify, and even if the attacker modified a legitimate update in transit the signature would also fail to verify.



One can see how this can be used in a Bitcoin-like currency: everyone publishes their public keys, and sending from A to B requires A to sign a message containing B's public key with his private key. From this, the protocol would be able to infer that A authorized a transaction sending some money to B.

Bitcoin, however, is more complicated. A Bitcoin address is not the public key; rather, the Bitcoin address is the hash of the public key. A hash is a function that can take anything as an input, and produces a fixed-size output, with the property that it is nearly impossible to invert. That is, given a message M, it is easy to calculate hash(M), but given hash(M) it would take until beyond the heat death of the universe to find M. With Bitcoin, the relationship between private keys and addresses is as follows:



So how do transactions work? Aside from the obvious information about the transaction itself, a Bitcoin transaction contains two things: the spender's public key, and a signature made with the spender's private key. Anyone verifying the transaction checks (1) that the hash of the public key is the spender's address, and (2) that the signature verifies with the public key. Taken together, these facts are a proof that the transaction was made (or authorized) by the owner of the private key corresponding to the spender's address.

The point of this is surprisingly mundane: under the elliptic curve DSA cryptosystem that Bitcoin uses, a public key is 512 bits long, meaning that it would take close to a hundred characters to represent. For example, here's what a public key looks like:

```
04b52fd5a616a8f08ccad58469102f86fc7891e5aa4262ab8d43e41767c17d45b80850044a62af51783609176
```

Meanwhile, the corresponding Bitcoin address in hexadecimal form is just:

```
4b463093e6fc3135a4de2ff577c4b658198777a9
```

And in its more familiar base58 form:

```
1obodiqhAZ3GD9onBXRZ9v7hshkuBreCu
```

In reality, however, this does not accomplish nearly as much as Satoshi thought it did. As it turns out, there is a way of encoding public keys in a much more compact way, taking up only 257 bytes:

```
03c5c9833d00bed3211a5f3733316ecf6ebc407806d70caa14862f1e2e8c2f852d
```

And if we had decided to put this into base58 form:

```
15sqRCowBDTfyuxPQD3ba8sN3wBB8MwGbo6gsBEGeKmUbNQADGh
```

Not really that much longer than the addresses we use today. So did Satoshi's choice simply introduce unnecessary complexity and waste? As it turns out, the answer is no. There is another very good reason to use the hash-of-public-key address construction: quantum cryptography. Quantum computers are capable of breaking elliptic curve DSA (ie. given a public key, a quantum computer can very quickly find the private key), but they cannot similarly reverse hash algorithms (or rather, they can, but it would take one 2^{80} computational steps to crack a Bitcoin address, which is still very much impractical). Thus, if your Bitcoin funds are stored in an address that you have not spent from (so the public key is unknown), they are safe against a quantum computer - at least until you try to spend them. There are theoretical ways to make Bitcoin fully quantum-safe, but the fact that an address is simply a hash of a public key does mean that once quantum computers do come out attackers will be able to do much less damage before we fully switch over.

The 21 Million BTC limit

One somewhat controversial property of Bitcoin is its fixed currency supply. There are currently 25 BTC being generated every 10 minutes, and this amount cuts in half every four years. All in all, there will never be more than 21 million BTC in existence. On the other hand, each bitcoin can be split into 100 million pieces (called "satoshis"), so it will not become difficult to use Bitcoin if its value goes up the same way it would become problematic to trade with dollars if each penny was enough to buy a car. Thus, all in all, the total number of currency units that will ever exist stands at 2,100,000,000,000,000: 2.1 quadrillion, or about $2^{50.899}$. In choosing this figure, Satoshi was much luckier, or wiser, than most people realize. First of all, the number is considerably less than $2^{64} - 1$, the largest integer that can be stored in a standard integer on a computer - go above that, and the integers wrap around to zero like an odometer.

Second, however, there is another, lower threshold that the total satoshi count manages to fall just below: the largest possible integer that can be exactly represented in floating point format. Integers are not the only kind of number that computers can store; to handle decimal numbers, computers use a format known as floating point representation. Floating point representation is essentially a binary version of scientific notation. For example, here are some values that you may be familiar with if you studied any physics:

- Mass of the Earth: $5.972 * 10^{24}$ kg
- Mass of the Sun: $1.989 * 10^{30}$ kg
- Speed of light: $2.998 * 10^{8}$ m/s
- One lightyear: $9.460 * 10^{15}$ m
- Mass of a proton: $1.672 * 10^{-27}$ kg
- Planck length: $1.616 * 10^{-35}$ m

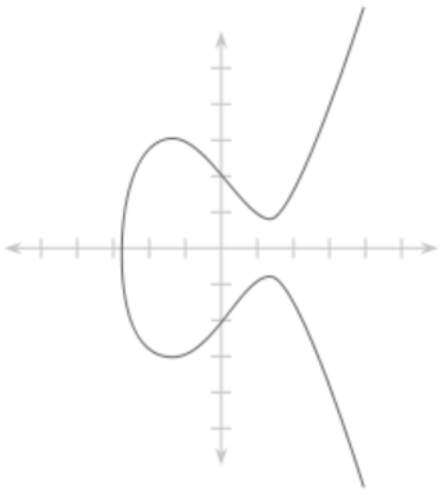
Notice how scientific notation allows you to express all these values with reasonable accuracy despite their wildly varying scales. Floating point notation is essentially scientific notation in binary; when you store the number 9.625, your computer stores "$1.001101 * 10^{11}$" (or rather, it stores 01000000 00100011 01000000 00000000 00000000 00000000 00000000 00000000, which is the same thing in high-precision serialized form). In this high-precision form, the "significand" (the part that's not the exponent) has 52 bits. What this means is that high-precision (more precisely, "double precision") floating point numbers are good enough to exactly store integers up to 2^{53}, but not higher - if you go higher, you start lopping off digits at the end. Bitcoin's $2^{50.9}$ satoshis are, in exponential terms, *just below* this maximum.

Why do we care about floating point values if we have integers? Because many higher-level programming languages (eg. Javascript) do not expose the low-level "floating point" and "integer representations", instead providing the programmer with only the concept of "number" - represented in floating point form, of course. If Satoshi had chosen 210 million instead of 21 million, Bitcoin programming in many languages would be considerably harder than it is today.

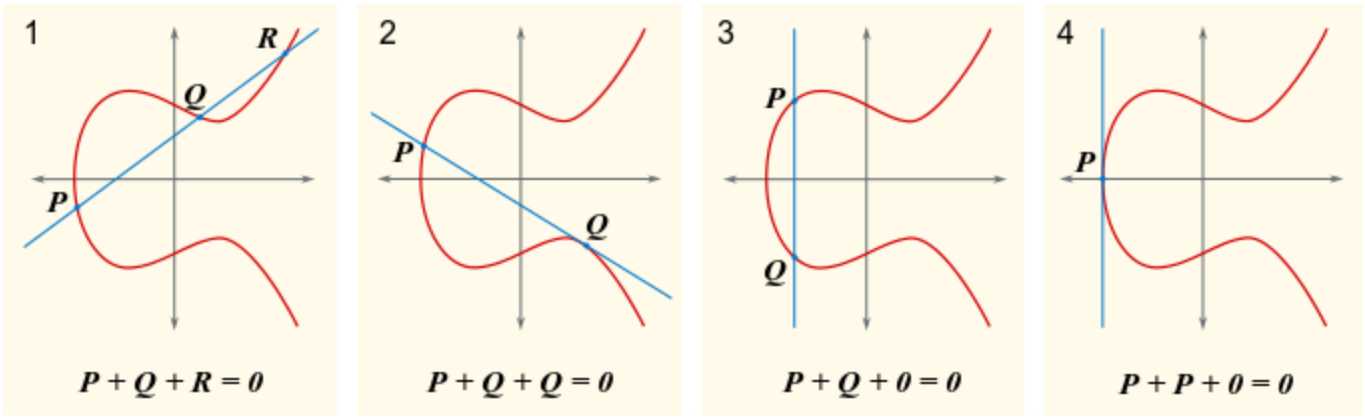
Note that Stefan Thomas in his BitcoinJS library did not take advantage of this, so that library uses a specialized "big number" object instead of a plain number to store transaction output values. When asked about this, Thomas replied that he realized that using regular numbers was possible, but BitcoinJS needed to include the "big number" library regardless, since elliptic curve arithmetic requires numbers up to 2^{512}, so the choice was arbitrary. My own BitcoinJS fork (which also adds other improvements) does use plain numbers to store the number of satoshis, a decision motivated largely by the desire to be compatible with external sources of transaction output data such as sx and pybitcointools.

Choosing the right elliptic curve

Elliptic curve cryptography, the kind of cryptography used by Bitcoin to digitally sign and verify transactions, is not a single standardized way of signing messages; there are in fact many different "curves" to choose from. To understand what different "curves" are, it first helps to have a basic understanding of how the math behind ECC works. In general, an elliptic curve is a set of points (x,y) on a two-dimensional plane such that the equation $y^2 = x^3 + ax + b$ holds, where a and b are parameters of the curve. Here is what one elliptic curve looks like:



Elliptic curve cryptography relies on operations called "point addition" and "point doubling" on such a curve, operations which are best described in diagrams:



Essentially, to add two points P and Q, trace a line between them, locate the other point on the curve that the line intersects, and trace a vertical line from that point to get your answer. However, for cryptographic purposes these ordinary elliptic curves have a weakness: they're imprecise. If you do very many point additions, floating point rounding errors will slowly accumulate, and eventually the result will be entirely meaningless noise. Thus, elliptic curve cryptography uses an elliptic curve with two modifications. First, the equation is now $y^2 = x^3 + ax + b + kp$, where k can be any integer and p is some large prime number (a parameter of the curve alongside a and b). Second, x and y must be integers. Although the resulting set is hardly a "curve", surprisingly enough the same math still works, and the restriction to integers avoids rounding errors.

There are many different curve parameters that could be used; the SEC2 document provides the standard ones. In general, however, the curves fall into two categories: "pseudorandom" curves and Koblitz curves. In a pseudorandom curve, the parameters a and b are chosen by a specified algorithm (essentially a hash) from a certain "seed". For secp256r1, the standard 256-bit pseudorandom curve, the seed is c49d360886e704936a6678e1139d26b7819f7e90, giving rise to the parameters:

```
p = 115792089210356248762697446949407573530086143415290314195533631308867097853951a

  = 115792089210356248762697446949407573530086143415290314195533631308867097853948b

  = 4105836372515214212932612978004726840911444101599372555483525631403946740129 1
```

The obvious question is this: where did the seed come from? Why was the seed not chosen to be some more innocent-looking number, like 15? In light of recent revelations regarding the US National Security Agency subverting cryptographic standards, an obvious concern is that the seed was somehow deliberately chosen in order to make the curve weak in some way that only the NSA knows. Thankfully, the wiggle room is not unlimited. Because of the properties of hash functions, the NSA could not have found one "weak" curve and then gone backward to determine the seed; rather, the only avenue of attack is to try different seeds until one turns out to generate a curve that is weak. If the NSA knows of an elliptic curve vulnerability that affects only one specific curve, the pseudorandom parameter generation process would prevent them from standardizing it. However, if they knew of a weakness in one in every billion curves, then the process offers no protection; for all we know, c49d360886e704936a6678e1139d26b7819f7e90 could have been the billionth seed that the National Institute for Standards in Technology tried.

Fortunately, Bitcoin does not use pseudorandom curves; Bitcoin uses Koblitz curves. In Bitcoin's secp256k1, the parameters are:

```
p = 115792089237316195423570985008687907853269984665640564039457584007908834671663a = 0b
```

That's it. And even p is quite simple to generate; it's only $2^{256} - 2^{32} - 977$ (in the interests of fairness, p and a in secp256r1 are also fairly simple; it's b that's problematic). The simplicity of these parameters gives the NSA

/ NIST very little wiggle room to create a deliberately bad curve. And even the specific values of 0, 7 and 977 can be justified by security and efficiency constraints, so the chance that Bitcoin's elliptic curve parameters were chosen with any malicious intent is very low indeed. When Dan Brown, the current chairman of the Standards for Efficient Cryptography Group, was asked about this, he replied: "I did not know that BitCoin is using secp256k1. Indeed, I am surprised to see anybody use secp256k1 instead of secp256r1." If secp256r1 is actually compromised, then since Bitcoin is one of the few applications that is using secp256k1 instead of secp256r1, Bitcoin has truly dodged a bullet.