How I checked over 1 trillion mnemonics in 30 hours to win a bitcoin

24 minute read

How I checked over 1 trillion mnemonics in 30 hours to win a bitcoin (https://medium.com/@johncantrell97/how-i-checked-over-1-trillion-mnemonics-in-30-hours-to-win-a-bitcoin-635fe051a752)

By John Cantrell (https://twitter.com/JohnCantrell97)

Posted June 18, 2020

TLDR;

* Alistair Milne tweeted (https://twitter.com/alistairmilne/status/1266037520715915267) that he planned to giveaway 1 Bitcoin in a wallet generated using a 12-word mnemonic.

- With 8 known words there are 2^{40} (~1.1 trillion) possible mnemonics.
- To test a single mnemonic we have to generate a seed from the mnemonic, master private key from the seed, and an address from the master private key.
- I wrote a CPU version in Rust to benchmark performance of a CPU solver. My Macbook was only able to check ~1,250 mnemonics per second which means it would have taken about **25 years** to check all 2^{40} possible mnemonics.
- I ported all necessary code for generating and checking a mnemonic (SHA-256, SHA-512, RIPEMD-160, EC Addition, EC Multiplication) to OpenCL C which is a programming language to run code on a GPU.
- The GPU version was able to check ~143,000 mnemonics per second which means it would take ~83 days to check all 2^{40} possible mnemonics.
- I wrote a server application that would orchestrate the distribution of work into batches of ~16 million mnemonics to a pool of GPU workers. Each GPU worker would ask the server for the next batch of work to do, perform the work, and log the result back to the server.
- I spent ~$350 renting GPUs from vast.ai (plus ~$75 for free from Azure).
- I was worried about other people doing the same and is why I included a .01 BTC miner fee. I didn’t think even this would be enough and thought there could be a ‘race to zero’ where people continually increased the fee trying to get the miners to include their transaction in the next block.
- I have open sourced all the code used to do this. Please see the bottom of the article for the links to the various projects.
- Creating a contest that isn’t won by software is difficult. I’d like to pay-it-forward and try crafting one myself. Follow me on Twitter @johncantrell97 (https://twitter.com/JohnCantrell97) for details.

Full Story

Alistair Milne tweeted (https://twitter.com/alistairmilne/status/1266037520715915267) that he planned to giveaway 1 Bitcoin in a wallet generated using a 12-word mnemonic.

This most likely meant it was generated using a BIP-39 mnemonic (https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki) and was later confirmed (https://twitter.com/alistairmilne/status/1267866768938340352) when he provided the first few seed words and eventually the entire BIP-39 word list in a tweet.

A BIP-39 mnemonic is generated using words from a fixed list of 2048 potential words. Each word is represented by an integer from 0 to 2047 corresponding to its position in the list. In binary you need 11 bits to represent a number up to 2047. Therefore to represent a 12-word mnemonic you would need 12*11 or 132 bits. It turns out BIP-39 uses 128 bits of randomness and then uses SHA-256 to generate a 4-bit checksum that is appended to the end of the original 128-bits to get you the required 132 bits for 12 words.

This means if we wanted to iterate all possible 12-word mnemonics we need to count from 0 to 2^{128}-1, where each number can be interpreted as a single mnemonic, converted to an address, and then checked against the address that holds the 1 BTC. Seems pretty easy at first glance but we quickly learn that this number is way too large to iterate in a timely manner.

Luckily, we do not need to actually check all 2^{128} possibilities because Alistair was going to release 1 seed word every couple of days. Each seed word we collect reduces the possibilities we need to check by a factor of 2^{11} or 2048.

He also mentioned he was going to release the last 3 or 4 words all at once to prevent brute forcing (or so he thought) so this meant I would need to be able to brute force at least the last 4 words in at most a couple of days for this to work in enough time to claim the prize.

With 8 words that means we know 8*11 or 88 bits of the 128 bits we are trying to figure out. It means there are 'only' 2^(128–88) or 2^{40} possible mnemonics we would have to check. This is 1,099,511,627,776 or roughly 1.1 trillion possible mnemonics.

I wasn't sure how long it would take to try 1 trillion possibilities so I wrote a quick program to get a baseline benchmark, so I could have some sense of exactly what I was dealing with, and if I thought it would even be possible to do.

The strategy I was going to use was to calculate a start and end number that I needed to iterate between based on a set of known input words. For each number I would calculate the address corresponding to that number and then check if the address was the one that held the 1 BTC. If it was the address I would then create and sign a transaction to sweep the funds into a wallet I control.

The first version I wrote in Rust using existing libraries (rust-bitcoin, rust-wallet, and ring) to handle all the hashes and elliptic curve math. Doing some quick benchmarking proved that using a CPU for this was not going to be feasible.

My laptop (2.5 GHz Dual-Core Intel Core i7) was only able to perform ~1,250 mnemonic to address checks per second or about 108,000,000 per day. This means it would take my CPU about **25 years** to generate and check the 1 trillion possibilities needed to brute force the mnemonic while only knowing 8 of the words.

In order to achieve this in 1 day I would need to be able to improve the performance by about 9,000x its current speed.

My next attempt was to rent a more powerful machine to see how fast the same CPU-only version could potentially run. I rented a 32-core CPU-optimized machine from Digital Ocean and was able to record a benchmark of ~8,000 per second which was only ~6 times improvement over my laptop.

I would still need about a 1,000 times increase in performance from here in order to do this. I didn't think I was going to get anywhere close to that by trying to optimize the CPU code as it was likely already pretty well optimized in the existing libraries I was using.

Enter the GPU

Over the last decade there has been a rise in utilizing GPU's to perform general purpose programming. In Bitcoin we saw this relatively early on when people started using GPU's to perform the operations required to mine.

So how exactly does a GPU help us solve this problem faster? It turns out that a single GPU core is actually *slower* than its CPU counterpart when used for general purpose programming. The performance gains are typically seen when you are able to efficiently parallelize a program. This is because a single GPU device typically has thousands of cores you can utilize for your computation.

Luckily for us, our problem parallelizes extremely well. Each of the 2^{40} numbers we want to check runs the exact same computation (number -> mnemonic -> seed -> master private key -> address). This means we could give each GPU core 1 number to try and could run thousands of attempts in parallel.

I quickly learned about OpenCL (https://www.khronos.org/opencl/) which is a standard open source programming language for writing software that will run on almost any GPU device. OpenCL C is very similar to the C programming language with a few differences. One of those differences is how memory works. In a GPU you have four main types of memory available to you (Global, Constant, Local, and Private). Global memory is shared across all GPU cores and is very slow to access, you want to minimize its use as much as possible. Constant and Private memory are extremely fast but limited in space. I believe most devices only support 64kB of constant memory. Local memory is shared by a "group" of workers and its speed is somewhere between Global and Constant.

My goal was to fit everything I needed into the 64kB of constant memory and never need to read from global or local memory to maximize the speed of the program. This proved to be a bit tricky because the standard precomputed secp256k1 multiplication table took up exactly 64kB by itself. Luckily, I was able to precompute a smaller table that used only 32kB but ran ~75% slower than the full table. The BIP-39 word list took up another~20kB and the SHA2 hashes took up another ~6kB so I was already using ~58kB of the 64kB available to me right from the start. This left me with about ~6kB of wiggle room to work with.

Ideally I would be able to do all of the computation on the GPU. This meant number -> mnemonic -> seed -> master private key -> address all calculated by the GPU and written in OpenCL.

What exactly needs to be implemented to handle each of those steps? Let's dive into each step we need to take to go all the way from a number to the bitcoin address. If you don't care about these details then just skip ahead in the article to the **Implementing in OpenCL** section.

Convert a Number into a 12-Word BIP-39 Mnemonic

Let's look at an example of how you can convert a number into a 12-word seed.

First let's start with a really big number:

> *34,267,283,446,455,273,173,114,040,093,663,453,845*

From here we need to convert this number into a 128-bit number in binary.

> *00011001110001111010001110000011110100110001011100011001001000100111110101001000101010000011111100000111001010100110001010010101*

We can get the last 4 bits (the checksum) by calculating the SHA-256 hash of this value and taking the first 4 bits of the result. In this case we get a checksum of 0101.

Now we append the checksum to the end and split our 132 bits into groups of 11 bits:

> *|00011001110|00111101000|11100000111|10100110001|01110001100|10010001001|11110101001|00010101000|00111111000|00111001010|10011000101|00101010101|*

We then convert each group of 11 bits into a number representing the index:

> *|206|488|1799|1329|908|1161|1961|168|504|458|1221|341|*

Finally we use these as indices into the BIP-39 english wordlist (https://github.com/bitcoin/bips/blob/master/bip-0039/english.txt) to find each corresponding word:

> *border dial thought plastic immense muffin vivid bench disease deer obvious click*

This is how we can map any number to a 12-word mnemonic. This step only costs us 1 SHA-256 calculation.

Mnemonic to Seed

The next step is to take this 132-bit mnemonic string and use it to generate a 64 byte binary seed. How do you extend a 132 bit string into 64 bytes? BIP-39 does this using a Password-Based Key Derivation Function (https://en.wikipedia.org/wiki/PBKDF2) with HMAC-SHA512 (https://en.wikipedia.org/wiki/HMAC) as the hash function, the string “mnemonic” as the salt, and the 12-word mnemonic as the password. It also uses 2048 iterations and each iteration requires two SHA512 calculations. This means this step will cost in total ~4096 SHA-512 calculations.

This is similar to how a lot of websites store hashed passwords in their database. The main idea is to make it slow to guess lots of passwords when trying to brute force the hash of someone’s password. You can control how long it takes to check a single password by increasing the number of iterations or using a slower hash function like scrypt or bcrypt.

Seed to BIP-32 Master Private Key

Once we have a seed we need to convert it into a BIP-32 (HD) Wallet. You can read the full BIP for all the details (https://en.bitcoin.it/wiki/BIP_0032) but at a high level BIP-32 defines a way to generate a master private key from a seed and then use that master key-pair to generate up to 2^{512} child key-pairs.

It’s a great solution for building wallet software as it makes it easy for a user to backup a single secret (their mnemonic) but to be able to generate nearly endless addresses (for all practical purposes). It has other nice benefits where you can generate child public keys without needing the private keys to be present (great for businesses that need to generate receive addresses without needing to have private keys on their server).

To convert our seed into a master private key according to BIP-32 we need to calculate HMAC-SHA512(“Bitcoin seed”, mnemonic_seed). HMAC-SHA512 produces a 64 byte output. We take the first 32 bytes as the master private key and the other 32 bytes are used later as the ‘chaincode’ to ‘extend’ the key when generating children key-pairs.

To calculate HMAC-SHA512 (https://en.wikipedia.org/wiki/HMAC) of a 132 byte input will take only two SHA-512 calculations.

Master Private Key to Address

This step involves taking our BIP-32 master private key and deriving child key-pairs based on the derivation path required to get to the address that holds the Bitcoin. If we look at the address in an explorer (https://blockstream.info/address/3HX5tttedDehKWTTGpxaPAbo157fnjn89s) we can see that it was generated using BIP-49 P2WPKH-nested-in-P2SH (https://github.com/bitcoin/bips/blob/master/bip-0049.mediawiki).

This means the derivation path is in the format m / 49’ / coin_type’ / account’ / change / address_index.

Figuring out the derivation path was a huge risk for this project. I assumed that Alistair simply generated a new wallet and the only transaction made was to deposit the 1 BTC. With that assumption it means the derivation path for the first address would be m/49’/0’/0’/0/0.

This means we need to take the Master Private Key and generate three hardened private keys and then two normal private keys.

Each hardened private key requires a HMAC-SHA-512 calculation (2 SHA-512 hashes) and one secp256k1 scalar addition.

Each normal private key has the same requirements as a hardened key plus the need to calculate the associated public key from the private key. To calculate the public key we need to perform an elliptic curve multiplication of the scalar represented by the private key by the secp256k1 group generator point G.

The last step is to take the calculated public key and convert it into P2SHWPKH address. This involves building the correct script (https://en.bitcoin.it/wiki/Script) and then using hash160 (RIPEMD-160 followed by SHA-256) to get the address and then SHA256 (twice) to calculate a 4-byte checksum.

In total this step costs us 10 SHA-512’s, 3 SHA-256, and 1 RIPEMD-160 hashes. It also costs us 5 EC scalar additions and 3 EC multiplications.

Total Cost

We have to do all of these steps for EACH mnemonic we want to try:

Number to Mnemonic — 1 SHA-256

Mnemonic to Seed — 4096 SHA-512

Seed to Private Key — 2 SHA-512

Private Key to Address — 10 SHA-512, 3 SHA-256, 1 RIPEMD-160, 5 EC Additions, 3 EC Multiplications

At a glance it looks like the seed generation step will be the slowest though it's hard to know how to compare SHA-512 hash to EC operations in terms of cost without some benchmarks. It will turn out that they both are relatively slow compared to the other steps but the seed generation is at least an order of magnitude more costly than the others.

Implementing in OpenCL

So in order to implement this entire flow in OpenCL I would need a way to perform SHA-256, SHA-512, RIPEMD-160, EC Addition, and EC Multiplication. I would also need to orchestrate it all together in a way to solve my problem.

My strategy was to find open source C implementations of all of the algorithms and port them to OpenCL C. I started with SHA-256 and SHA-512 because these alone would allow me to calculate number -> mnemonic -> seed and I knew that generating the seed was the slowest part by far.

Eventually I got seed generation working in OpenCL and my first benchmark using a nVidia 2080Ti showed me that I could generate 142,857 seeds per second. Wow! Now we were getting somewhere.

This meant a single nVidia 2080Ti could generate ~12 billion seeds per day. This still meant it would take **83 days** to generate all trillion seeds. That was a *lot* better than the 25 years my CPU was going to take. However, this was still only generating the seeds and a seed wasn't enough to know if the mnemonic was correct or not. I would still need to complete the process of converting the seed all the way to an address to be able to verify if it was the correct mnemonic.

I then went back and benchmarked my CPU version of the seed to address generation to see how long that would take. The 32-core Digital Ocean machine was able to process about 52,000 seeds per second. This was pretty decent but after the OpenCL seed generation improvements it was now the bottleneck and would still take over 221 days to complete. Additionally, I would need to coordinate moving the seeds generated from GPU back to the CPU to finish their processing. This coordination would take time and be a more complicated program to write.

My goal was to move the entire calculation into the GPU.

This meant I needed to be able to perform EC math (addition and multiplication) in OpenCL. I took the open source libsecp256k1 implementation (https://github.com/bitcoin-core/secp256k1) that Bitcoin uses and ported it to OpenCL C. It required me to first understand how the libsecp256k1 library was structured and what exactly I needed to port in order to generate an address from a master private key.

It turned out to be about 2,000 lines of code that I needed to port. Luckily, OpenCL C and standard C are similar enough that there wasn't a lot of changes required. The changes involved me implementing some memory management functions like memcpy, memset, and memzero that OpenCL does not support. It also involved me removing the blinding that is done when performing EC Multiplication and precomputing a 32kB table instead of the default 64kB one.

After all is said and done I ran some sanity tests and was surprised to discover that my OpenCL implementation was actually working correctly.

I then re-ran my benchmarks using the 2080Ti and saw that the time added to calculate the address from the seed using the GPU was negligible. It only added a few hundred milliseconds per 1 million seeds.

I was now able to run the entire process 100% on a GPU but it was still going to take ~80 days to enumerate the 1 trillion possible mnemonics using a 2080Ti.

I then tried running it on a bunch of different video cards (1080, 1080Ti, 2070, 2070Ti, Tesla K80, Tesla P100, and Tesla V100). To my surprise the performance didn't change that much across GPUs. Even the top of the line Tesla V100 was only about 15% faster but cost almost 4x as much to rent.

On the lower end the 1080/1070s were roughly 3 to 4 times slower than the 2080Ti but were only about half the cost. The 2080Ti seemed to be the most cost efficient card to use for this problem. How many could I get and how would I orchestrate all of this?

If I was to solve this in 24 hours I would need the power of about 80 2080Ti's.

Orchestrating a GPU Pool

If I wanted to distribute this problem across multiple GPUs the simple answer is to just break down the 2^{40} numbers we need to iterate into 80 equal parts and run each part on a single GPU. Unfortunately, it would not be this simple.

I first needed to figure out how I was going to get access to all of these machines. My first thought was to use the major cloud providers (AWS, Google Cloud, and Microsoft Azure). I quickly learned that these companies have strict quotas on the number of GPUs you can provision (some of them ZERO!) with a new account.

Luckily, I came across a GPU marketplace (https://vast.ai/) (Vast.ai) that let people who had unused GPUs rent them out to anyone who wanted access to GPUs. They had large inventory of 1080's and 1080Tis but not as many 2080Ti's as I needed. The inventory fluctuated all the time depending on how many people were renting them, how much they were willing to pay, and how many providers were online at the time.

This meant I wasn't going to easily be able to allocate exactly 80 2080Ti's and evenly distribute my workload across them all.

I ended up building a simple centralized server that would act as the distributer of work. It is pretty similar to how a mining pool works. Each GPU worker would make a request to the centralized server for a batch of work to perform, perform the work, and then log the work(and a solution if it found one) back to the server. Each worker would continue to do this in a loop until there was no more work to do.

This meant I could easily spin up as many cards as I wanted as fast as I wanted and the central server would be able to keep track of what the next batch of work was when one was requested. Each worker instance didn't need to know anything about what part of the work it needed to perform, it could just blindly ask the server for the next batch to work on.

This solution does add more time in terms of network latency. I needed to make the batch size large enough so that the added network latency was a small percentage of the total time. I ended up using a batch size of 16,777,216 which means there would be 65,536 batches of work to compute all 2^{40} possible mnemonics.

A 16,777,216 batch took the 2080Ti slightly less than 2 minutes to compute. This means the less than 1 second network latency was adding less than 1% additional computation time.

Testing the System

This ended up being a more complicated system than I originally envisioned and I was worried there would be a bug or that it wouldn't work as expected when the time came. I ended up going through a full end to end test to make sure everything was actually working.

I created a wallet with a new BIP-39 mnemonic and transferred 0.0001 BTC into it. I then initiated my system with 9 known seed words (so it wouldn't take as long or cost me as much). I rented a couple 2080Ti's on vast and let it rip. Within 20 minutes it had found the solution and swept the 0.0001 BTC to a Trezor wallet I controlled. I felt like I was ready even though I was unsure I'd actually be able to get enough compute power in time to perform the task when it was needed.

I felt like there was still a way to optimize the SHA-512 code I was using by trying to port the version of SHA-512 that hashcat (https://github.com/hashcat/hashcat) was using as they have an extremely optimized version. Since SHA-512 was the most used method and the current bottleneck any improvements made here could drastically reduce the number of GPUs needed. I was about halfway through this implementation when Alistair released the 8th word (https://twitter.com/alistairmilne/status/1272568234588483590).

The Big Day

I immediately threw away the optimized SHA-512 I was working on and went back to the version I knew was working from earlier. I started renting as many 2080Ti, 2080, 2070, 2070Ti, 1080Ti's as I could from vast.ai. In the meantime I had been able to get my gpu quota increased on Azure (+a $200 free credit for new accounts) to rent up to 40 GPUs over there. Unfortunately, the machines I had access to over there were only roughly ~50% as powerful as a 2080Ti. However, this meant I was able to get roughly 20 2080Ti's for free from Azure alone.

At the peak I was testing about 40 billion mnemonics per hour. This means it should have taken around 25 hours to test the 1 trillion mnemonics. I knew that on average it should only take 50% of the time (depending on what the 9th word actually was). If the word started with A then it would finish in 1 hour and if it started with a Z then it would take the full 25 hours. Of course my testing rate was not steady at 40 billion per hour as machines on vast.ai come and go as people outbid me or go offline. I had to continually scan the list of available machines and rent more as they became available.

As the day went on without finding the solution I became worried it wouldn't work because there were a lot of ways that my approach could fail:

- I assumed the words he was releasing were in the correct order. If they were not in order there would have been 8! (factorial) more possibilities (making 8 words basically impossible to brute force) and my code wasn't trying the different permutations anyway.
- I assumed I had all 8 words correct. While most were obvious there were a couple where I felt like there could be other options. I was not trying any of those other options, only the 8 I thought were correct.
- I assumed he was using the first address of the first account of the HD wallet derived at m/49'/0'/0'/0/0. If he used any other derivation path (second or later address) I would not have found it. I was *REALLY* nervous about this one because there was no way I could know for sure what derivation path he had used. Luckily, he used a brand new wallet without generating extra addresses before depositing the 1 BTC.

After a full day of running my work server status showed that it was about 85% of the way done with testing all possibilities and I had largely given up hope that it would work. I literally almost turned it off at this point to implement a version that tested more than just the first address because I was convinced that assumption was wrong.

I couldn't get myself to actually stop it at that point as I had come so far so I just let it continue. To my surprise a little while later that evening (at 91%) and after almost 30 hours and exactly 1 trillion checks (1,000,710,602,752) it had found a solution!

I couldn't believe it had worked. I nervously plugged in my Trezor to check the balance to make sure I hadn't screwed up the code that generated and signed the transaction to sweep the BTC. To my relief the 0.99 BTC was there!

I then nervously waited for a confirmation to arrive. I was worried that another person (or eve Alistair) would try to steal back the Bitcoin by continually increasing the miner fee so that a miner would include their transaction over mine. This is one of the reasons I started with a relatively high fee (0.01 BTC). I didn't have time to implement a tool that watched the mempool for competing transactions and automatically bumped my fee but I thought it would be a good idea.

However, after a few minutes my transaction was included in a block. Wow, it really had worked.

Final Thoughts

I had a lot of fun and learned a lot working on this problem. I have been thinking of ways to run a similar type of giveaway while avoiding the possibility of someone writing software to win. It's not so easy actually.

Even if he had decided to release the last 5 words all at once to prevent brute forcing I could still have software running that was waiting for me to enter each word as I figured them out, assuming it was some kind of puzzle and automatically sweep the coins faster than any human could.

I was thinking he could have used a deeper derivation path and while my initial version of software would have missed this it would be trivial and not that much slower to check lots of derivation paths for the used address. I think the issue here is even for humans, the wallet software that handles the recovery process only scans for a small number of unused addresses before stopping so he couldn't have put it too deep if he wanted users to be able to use a normal wallet to recover it.

Another idea is to give the words out of order which would have made it impossible to brute force early but still doesn't stop me from using software to enumerate all possible orderings and sweep the btc faster than a human could ever hope to do it once enough words were released.

At the end of the day software will always be faster than humans at this kind of task. I think the only real way is to make the discovery of the actual words the difficult part. Some kind of puzzle where solving it provides you with all of the words at once so the race is more about solving the puzzle than it is about how fast you can enter the words.

The Software

I have made all of the projects I used to solve this problem open source for anyone interested in learning exactly how it was done. Hopefully it will be useful for someone learning about GPU programming or to help someone recover some lost BTC.

BIP39-Solver-CPU (https://github.com/johncantrell97/bip39-solver-cpu): This is the CPU benchmark tool I wrote in Rust to get an idea of how long it will take do solve on a CPU for certain number of unknown words

BIP39-Solver-GPU (https://github.com/johncantrell97/bip39-solver-gpu): This is the actual GPU version I ran on each worker GPU to solve this problem.

BIP39-Solver-Server (https://github.com/johncantrell97/bip39-solver-server): This is the actual server I ran that handled distributing the work to all the workers.

Tags: 2020 Q2 Azure BIP39 Brute Force GPU John Cantrell Mnemonics Open Source Security Seed Words Software

Categories: posts

Updated: June 18, 2020