`cuckoo-http`

Using the Cuckoo Cycle Proof-Of-Work problem to harden web servers against denial-of-service attacks

Srinivas Kaza

github.com/AnimatedRNG/cuckoo-http

1. Abstract

Denial-of-Service attacks incur billions of dollars in damages every year, and have caused outages for major services. `cuckoo-http` is a front-end web proxy server and framework which issues clients a Cuckoo Cycle proof-of-work problem, and uses the solution as an access token to guard computationally intensive tasks. The Cuckoo Cycle problem has several desirable properties for this application – it is memory latency-bound (given a certain set of parameters), has no known *practical* time-memory trade-off, and is trivial to verify. To that end, the front-end proxy processes a small subset of HTTP by representing the parser state as a simple finite state machine and drops TCP connections as soon as possible.

2. Introduction

In a denial of service attack, the attacker attempts to make a service unavailable by flooding the underlying system with superfluous requests [1]. Generally, the aim is to prevent legitimate users from accessing the service. A distributed denial-of-service attack involves using more than one IP address. Collections of private computers that have been previously compromised by malware, also known as botnets, can be used to perform large denial of service attacks. Some of these are more than able to incapacitate small websites, and weaken larger ones [2].

In the last few years, there have been a few examples of large-scale DDoS attacks that have managed to disable popular services for hours on end. In late 2016, the Mirai botnet attacked the DNS service provider Dyn and disabled several high-profile websites, such as Netflix, AirBnB, Twitter, and Github [3]. This attack was particularly devastating due to the sheer scale of requests received by incoming servers – the highest peak was estimated to be around 1 terabit

[1]https://www.incapsula.com/ddos/denial-of-service.html
[2]https://www.incapsula.com/ddos/botnet-ddos.html
[3]https://www.wired.com/story/mirai-botnet-minecraft-scam-brought-down-the-internet/

per second. Mirai was also unique in that it used poorly-secured IoT (Internet of Things) devices to accomplish its ends. Given how many unique devices participated in the attack, traditional DDoS prevention mechanisms were unable to throttle this traffic. Earlier this month, a record 1.3 terabit per second attack was attempted on Github, which was fortunately able to redirect the incoming packets to a CDN (Akamai) [4].

Blockchain-based distributed consensus protocols typically use a proof-of-work algorithm to generate blocks. A proof of work is a solution to a problem which is easy to verify (i.e prove that a given solution is correct) but hard to solve (i.e generate a solution solely given the problem). Bitcoin uses a PoW system based on the SHA-256 hashing algorithm. The general gist of the algorithm is that the miner (i.e the person who wishes to solve the PoW problem) must find a string, given some set of constraints, that has a hash with a certain number of leading zeros. Given a well-designed hash function, the process of obtaining this string is very computationally difficult. However, determining that the solution string is correct is quite simple – just check that the hash of the string has the correct number of leading zeros [5].

Unfortunately, the SHA-256 hash function is not memory-intensive. Despite the author's original intent for most Bitcoin mining to occur on a miner's CPU, mining software was quickly developed for GPUs (graphics processing units), which were orders of magnitude faster. Soon after, FPGAs (field programmable gate arrays) were designed to mine the coin, and then dedicated ASICs (application-specific integrated circuits) became a viable option. Mining pools became dominated by a handful of powerful miners. And to that end, these miners have used their clout to influence development of Bitcoin. As of 2018, cryptocurrency mining uses as much power as the country of Greece.

Over the years, other cryptocurrencies have emerged that use better PoW algorithms. Litecoin used the memory-intensive scrypt KDF (key derivation function) to prevent the proliferation of custom mining hardware [6]. While this endeavor failed, more recent currencies such as Ethereum (based on Ethash), Zash (based on Equihash), and Monero (based on CryptoNight), have all been somewhat successful at providing ASIC-resistant proof of work algorithms that avoid Bitcoin's tragedy of the commons scenario.

In late 2014, the Cuckoo Cycle algorithm was published by John Tromp. Unlike other proof-of-work algorithms, Cuckoo Cycle is graph-theoretic, based on the idea of finding length 42 cycles in massive random graphs [7]. The siphash function is used to determine the edges of the graph [8]. Unlike some other cryptocurrencies (Ethereum or Monero), the verification algorithm (check whether the list of edges

[4]https://nakedsecurity.sophos.com/2018/03/05/worlds-largest-ddos-attack-thwarted-in-minutes/

[5]https://en.bitcoin.it/wiki/Proof_of_work#Example

[6]https://www.tarsnap.com/scrypt.html

[7]Cuckoo Cycle: a memory bound graph-theoretic proof-of-work

[8]SipHash: a fast short-input PR

is a cycle) is not at all computationally heavy, Cuckoo Cycle was designed to be memory latency bound, making it a much less power-intensive algorithm given that the CPU/GPU is spending most of its time waiting on memory fetches. A smartphone charging overnight would be a suitable miner. Better miners are memory bandwidth bound rather than latency bound, but the same principles apply, except that the CPU/GPU would likely be more busy.

3. Motivation

Most botnets are capable of staging effective attacks because they are limited by network bandwidth rather than any other computational resource. As evidenced by the case study above, this is an unfortunate property. To that end, a "challenge" posed to the user should use a scarce computational resource, such as processing time, memory, or storage; or a human resource, such as a CAPTCHA.

We intend to use Cuckoo Cycle to introduce this challenge, by requiring clients to solve a Cuckoo Cycle problem in order to access certain resources on a website. We use a set of parameters for the Cuckoo Cycle problem that ensure that it is memory latency bound and requires a substantial amount of memory.

It might not seem obvious why these constraints greatly limit the number of requests that a client can send to a server. To emphasize this point, consider a simple example. Imagine a simple botnet which has 10000 computers each with 4 GB of memory and a 1 GHz CPU with 2 cores. That would imply that the entire botnet has 40000 GB of memory total (ignoring the possibility of swapping to disk). If each request requires 100 MB of memory and takes 2 seconds to compute, then the botnet is capable of issuing at most 200000 requests per second. Despite the sheer volume of requests in the above example, it is actually not that much – well under 1 gigabit, given the largest request size that this protocol supports. Also, during an attack the memory requirements can be increased. This paper assumes that it is reasonable to expect that users of a web site will tolerate a 1-2 second page load delay and the brief use of 100 MB of memory.

Another way to look at this solution is from the perspective of total work performed by the server operator versus the botnet. As mentioned earlier, the Cuckoo Cycle verification algorithm is computationally lightweight – it merely involves verifying that the provided nonces (edges in the random bipartite graph) are a cycle. However, finding such a cycle is several orders of magnitude more difficult. If *verifying* the result of some computation requires performing n operations (i.e for the server), ideally performing the computation (i.e the client/botnet) should be on the order of $10000n$.

Faced with the difficulty of computing the cuckoo problem, an attacker may instead decide to exploit other flaws in the system, such as the behavior of the front-end proxy itself. They may decide to spam the server with garbage HTTP requests that do not include valid solutions to the cuckoo problem, in hopes of exhausting the server's resources. An easy way to mitigate this problem is to simplify the front-end proxy as much as possible, parsing the enough amount of information from the request needed to determine whether it ought to be cuckoo-verified and whether it is already cuckoo verified, and then forward it to the primary server. A simple finite state machine will do.

Finally, the memory-latency bound of the Cuckoo Cycle problem makes the technique friendlier to less powerful internet-connected devices. The majority of memory accesses in the Cuckoo Cycle miner (given certain problem parameters that will be detailed in the next section) are cache misses (as they are edge lookups on a large random graph), which implies that the CPU will be stalled waiting for data to become available. Indeed, some analysis of core temperatures during mining suggest pipeline stalling does occur. A natural consequence of this behavior is that the device will use less of the CPU's functionality, running cooler and drawing less power. Given that memory latency in smartphones (LPDDR*) is comparable to its laptop/desktop counterparts (DDR*), the mining algorithm runs roughly as fast on mobile (e.g requests take 2-3 seconds rather than 1-2 seconds) as it does on a powerful desktop.

4. Design

`cuckoo-http` requires a certain handshake between the client and the proxy server before the request is forwarded to the web application. This handshake demonstrates that the client has performed a certain amount of work before it interacts with the server.

For the purposes of this project, the model specified in 4.1 has been implemented. Section 4.2 covers a more flexible model, which was not implemented for this proof-of-concept.

4.1 Wrapped Request Model

This request model offers the greatest level of abstraction around the `cuckoo-http` service. The web application need not be designed in a way that takes cuckoo hardening into effect. Unfortunately, this model constrains the kinds of requests that can be handled by the application.

In the Wrapped Request Model, clients start by requesting a webpage from the proxy server. The proxy server generates a random 32-byte bytestring, which represents the header of the cuckoo problem to solve, as well as the other parameters of the cuckoo problem (easiness and hash difficulty, see section 4.3 for details). It then responds to the client's request for the webpage with a page solely consisting of a script tag containing the JavaScript code which solves this problem, as well as the parameters of the problem embedded in script data tags.

This page also contains a script data tag containing the original message request. At runtime, the miner examines these tags to determine the problem parameters. Once the problem is solved, the client responds back to the server with another message, containing the solution to the cuckoo problem, as well as the original request. The server verifies the solution, and if it is correct, it forwards the result to the web application.

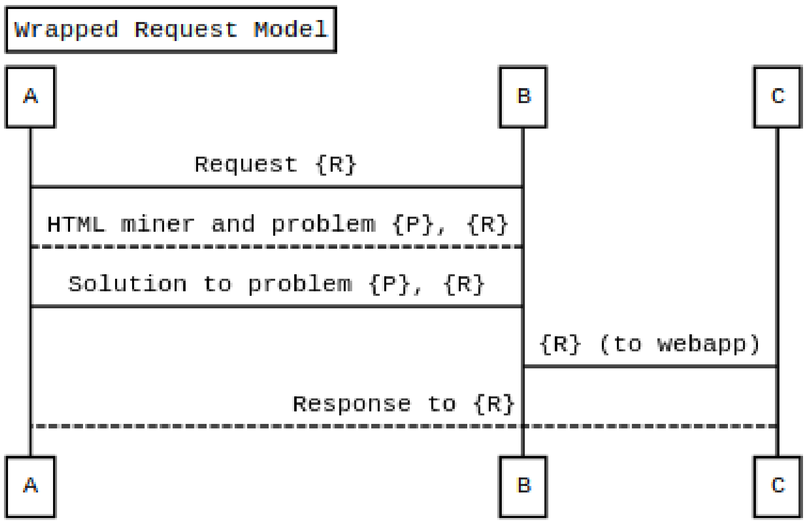

Figure 1: Wrapped Request Model – A is the client, B is the proxy server, C is the web application

Admittedly, this is an inflexible and crude solution to performing the handshake. However, the operation of the proxy server is completely opaque to the web application, simplifying the process of cuckoo hardening web applications that already exist.

4.2 Access Token Model

The request model in section 4.1 is a plug-and-play solution designed to guard HTTP requests (usually GET requests). Unfortunately, it constrains the web application developer and requires processing within iframes in order to properly handle POST requests. Integrating cuckoo-hardened requests into the front-end logic gives the web app more control over the cuckoo handshake and simplifies reasoning about asynchronous requests.

In the Access Token Model, clients begin the handshake by requesting a cuckoo header from the server. This request includes a HTTP header that indicates this intention. The server replies with a 32-byte bytestring, which represents the

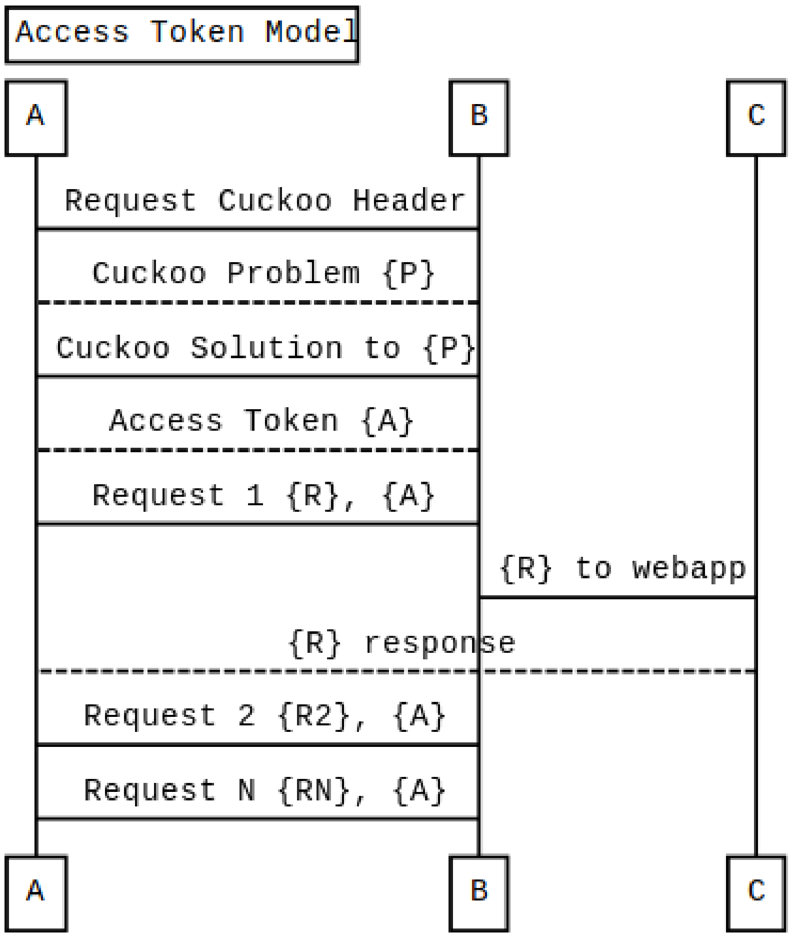

Figure 2: Access Token Model Diagram

header of the cuckoo problem to solve, as well as the other parameters of the cuckoo problem. Unlike the response from 4.1, the response is not an HTML page with embedded JavaScript; it just has the header, the easiness, and the hash difficulty. The client reads the problem parameters, solves it, and then requests an access token (including the solution as a header). Upon receiving this request, the server issues a temporary access token to the client, which expires after a certain amount of time (usually less than 30 seconds). Any request sent to the server which contains a valid access token is forwarded to the web application.

The short lifespan of the access token is important, as it guarantees that an attacker cannot effectively share access tokens among its botnet. Even though the proxy server does verify that each access token is used by only one client, a botnet operator can circumvent this measure by spoofing individual client IP addresses. A short access token timeout and IP checks should make spoofing more difficult. Alternatively, the access token could be limited by use rather than time. Perhaps access tokens can only be used a limited number of times before they expire.

The access token model was not implemented for this project, although it would be easy to add given that all the requisite components have already been developed.

4.3 Cuckoo Cycle Parameters

Small modifications have been made to the Cuckoo Cycle problem to improve control over the difficulty of the problem, as well invalidate mining strategies which attempt to trim edges (and thus are no longer memory-latency bound). As mentioned before, the Cuckoo Cycle solver attempts to find a length 42 cycle in a random bipartite graph. Despite the author's original intent to make the problem memory-latency bound, a mining strategy known as edge trimming was developed that could solve certain kinds of Cuckoo Cycle problems much more efficiently than the memory-latency solver. From then on, much (but not all) of the work on this problem included edge-trimming, which is undesirable for our use-case.

Edge trimming attempts to remove vertices from the graph which have a degree of 1. In a graph in which the ratio of edges to vertices is less than or equal to $\frac{1}{2}$, this strategy is easy due to the large number of degree 1 vertices. This ratio, somewhat confusingly, is referred to as the "easiness" of the problem. Increasing the easiness to 70% makes edge trimming less effective, and also practically guarantees finding at least one length 42-cycle (i.e the probability of *not* finding a solution is less than 10^{-5}).

The original implementation of the simple, memory-latency bound miner included a strategy for parallelizing the memory reads. Unfortunately, this approach has a race condition in the cuckoo hash table code which requires an hacky workaround to circumvent. One could solve this problem by performing that particular code in a dedicated thread, but the entire notion of having a multi-threaded miner for a single problem was not considered because WebAssembly (see Section 5) does not currently support threading. Having a parallelizable solution is ideal, if it can improve performance – there is no point leaving a CPU core idle if it could be used to accelerate the computation. The chosen approach was to add an additional "hash difficulty" problem constraint, and issue additional problems. A solution is only considered valid if the hash of all of its nonces are less than the hash difficulty. Thus, the probability of finding a solution at all in a given graph is set to be around 0.4, and the miner likely has to solve multiple problems if it wants to find a valid solution at all. The ideal way to approach this problem is to run multiple miners at once in different threads (i.e the new Web Workers feature).

We opt to parallelize the problem in this manner to not only consume all available CPU resources, but to also gain more control over the CPU-memory tradeoff required to solve it. See Section 6 for details.

5. Implementation Details

With the exception of the Cuckoo Cycle problem itself, all of the requisite web features required to perform the steps mentioned in Section 4 were developed many years ago. That said, this implementation uses a distinctively modern language and framework, which hopefully should reduce the number of security-related bugs in the front-end proxy server and also greatly improve performance, both on the client and server side.

5.1 Why Rust is the best language for security-oriented applications

Almost all of the code for this project was written in Rust, from the (re)-implementation of the Cuckoo Cycle miner and verifier, to the proxy server, to even the majority of the client-side scripting that executes the miner and parses the HTML response. Rust is an ideal language for this project for several reasons:

- Rust is safe. For security applications, safety is far more important than performance. By design, many of the bugs that occur in memory-managed languages *cannot* occur in safe Rust. Rust disallows derferencing a dangling pointer, prevents mutable pointer aliasing, and enforces a stronger form of const-correctness. Rust even prevents some problems in languages that are not described as memory-managed. Rust cannot have null pointer exceptions – functions that may not return a value return an Option type, and Rust forces the programmer to take account of this property. Memory is not default initialized; the programmer must set the memory before it can be read – eliminating problems from default constructors and from uninitialized memory. Rust's powerful type system also enforces safe casting between types and throws exceptions on overflow (in debug mode). The ownership semantics make data races over a variable impossible, thus making race conditions more difficult to accidentally introduce. Perhaps most importantly, there is almost no undefined (or platform-specific) behavior. These protections prevent many security bugs from ever being compiled in the first place.

- Rust is fast. The language is primarily focused on zero-cost abstractions – there is no garbage collection. Rust compiles directly into LLVM, and can benefit from all of the optimizations that LLVM performs. Benchmarks have shown that Rust code has comparable performance to analogous C code. Preventing a DDoS attack requires that the proxy server keep up with the unending barrage of fake requests. To that end, the default hash map implementation in Rust is designed to be DoS-resistant.
- Rust can be *easily* compiled to WebAssembly. Given WebAssembly's near-native performance, we opt to compile our Rust miner (which the client should execute to solve the problem issued to it by the proxy server) directly to WebAssembly. No alterations were made to the Rust mining code in order for it to run in the browser, other than the wrapper code which executed it.
- Easy-to-use tools/frameworks for concolic testing and fuzzing. While a more traditional approach to testing was used for this project, it would be easy to test our code this way.

5.2 Front-end Proxy Implementation

Neither the model mentioned in 4.1 nor the model mentioned in 4.2 require the entire HTTP header to be parsed, just certain headers. As a result, we can implement a far simpler HTTP parser. We also want the parser to quickly determine whether a request is malformed. Thus, a simple finite state machine was implemented to extract the relevant parts of the HTTP request. Because this HTTP parser has to do much less work than a full fledged parser, it runs *much* faster than most HTTP parsers.

Unfortunately, Rust only support UTF-8 strings – there is no support for string operations upon simple ASCII strings. Oddly enough, converting a string from ASCII to UTF-8 is slow in Rust, despite the fact that pure ASCII is encompassed in UTF-8! Additionally, the string processing operations in Rust were somewhat slower than expected. Given that I only need a find and find/replace operation, I decided to reimplement this functionality upon byte-strings.

Notably, the front-end proxy server ignores the request header field "Connection," and opts to always drop requests after it responds. This implementation detail was motivated by DoS attacks that exploit the idle/active resource consumption of open TCP sockets.

There were other design decisions which improved the performance of the front-end proxy, such as buffering the randomly generated cuckoo headers; however, there are still opportunities to micro-optimize the parser (as section 6 would suggest).

5.3 WebAssembly

I had few issues getting my Rust code to properly compile into WebAssembly, although I should say that performance varied a lot across different compiler backends. Emscripten created large WebAssembly files that took a long time to load and even longer to execute. I also tried compiling to asm.js, but that performed even worse than the Emscripten backend. The Rust nightly backend worked quite well though; the miner took only a ~20% performance penalty.

Currently, the client requests both the WebAssembly file and the script that loads the WebAssembly file from the proxy server. That task could be offloaded to a CDN, taking even more load off of the proxy server.

6. Evaluation

The performance of the proxy server can be measured in several ways. One evaluation metric is to merely determine how many requests it can serve per second. The goal of this test is to spam the HTTP server with requests, ignoring the actual cuckoo problem itself. A test was performed using the `wrk` utility with 12 threads running for 30 seconds, with Python's `simple-http`, `NodeJS`'s example HTTP server, and `lighttpd`, a popular front-end proxy. All applications were serving a blank page, all of which were larger than 50 KB in size.

I suspect that Lighttpd's incredible performance comes from aggressive caching. Given the relatively immature state of `cuckoo-http`'s optimization, I am confident that it can eventually reach `Lighttpd`'s performance and exceed it.

I also benchmarked the time to solve a Cuckoo Cycle problem given different hash difficulties on my Intel i7-4712HQ. All tests have an easiness of 70%, and a graph size of 2^{23} (and thus, consume around 2^{23} bytes). Note that larger hash difficulty values are easier to solve, and smaller hash difficulties are more difficult to solve.

One interesting side-effect of the memory latency-bound behavior is that the CPU temperature does not increase very much. Running the cuckoo miner on a large random graph of 2^{29} vertices only increased the core temperature by ~20 degrees Celsius to ~70 degrees Celsius. Most computationally intensive applications raise the core temperature to at least 80 degrees Celsius.

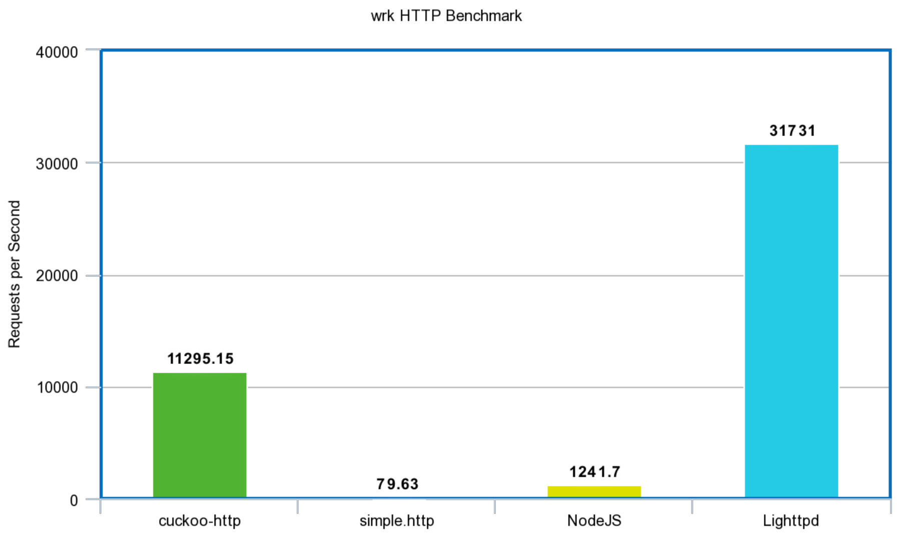

Figure 3: wrk HTTP Benchmark

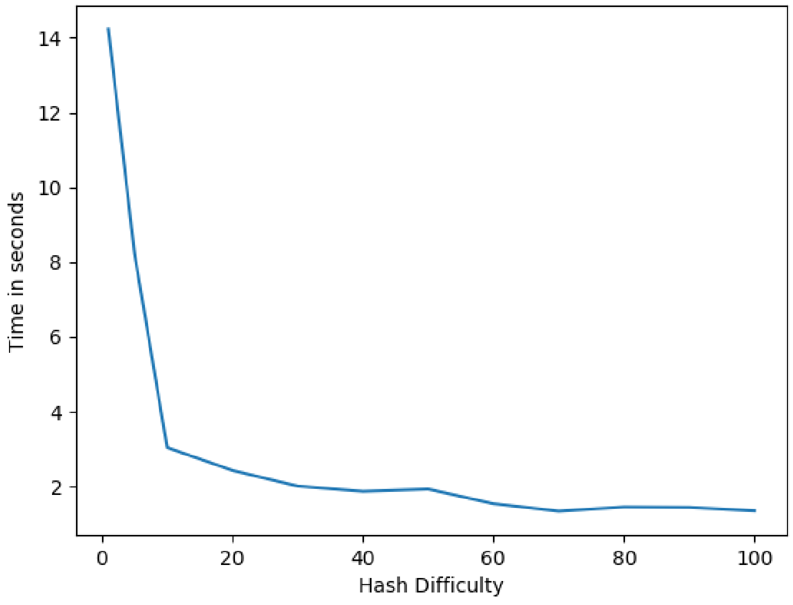

Figure 4: Hash Difficulty Benchmark

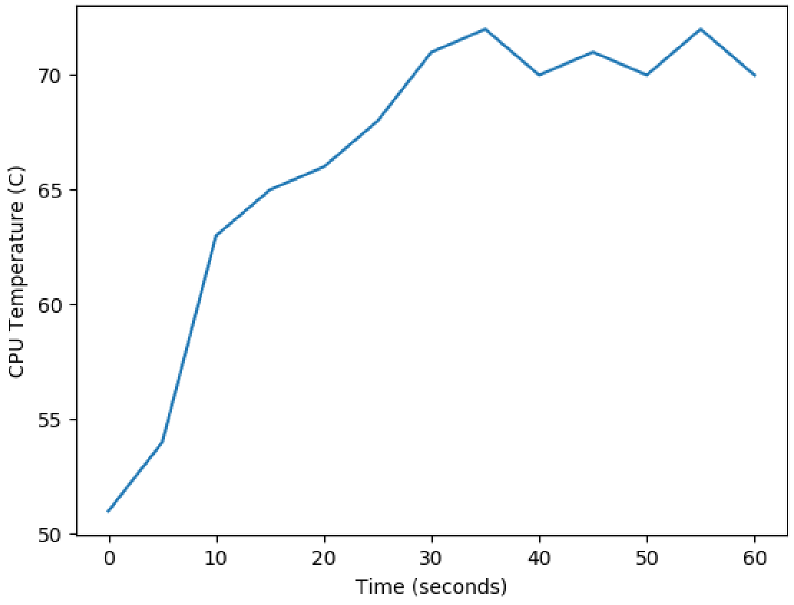

Figure 5: CPU Temperature Benchmark

7. Conclusion

Over the course of this project, I have implemented an limited HTTP server from the TCP socket level. While reinventing the wheel is a fun experience, I strongly suggest using already-existing components to perform HTTP parsing *if possible*. Most of the time spent on this project was writing the HTTP server, including some of the TCP plumbing.

Transitioning from unsafe languages like C++ and C to Rust was somewhat difficult, but once my code compiled, there were very few bugs. I was also impressed by the Rust-WebAssembly ecosystem. With the presence of “standard web” libraries like `stdweb`, one could create entire web applications without writing more than a few lines of JavaScript.

If a protocol like `cuckoo-http` were adopted on a larger scale, custom hardware (i.e an ASIC) could be used to operate the front-end proxy server, given the limited requirements. A NetFPGA board could be a good choice [9]. There is something to be said about the notion of using proof-of-work problems to gain access to computational resources – one day it might even be feasible to incorporate the `cuckoo-http` handshake into the TCP handshake itself.

[9]https://netfpga.org/site/#/