Bachelor's Thesis
(Course Code: XB_40001)

Fairness in Hashcash through Memory Hard Functions

by

Hendrik Hermann Werner Joel
(Student Number: 2722498)

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science
in
Computer Science
at the
Vrije Universiteit Amsterdam

July 25, 2024

Certified by .....
First supervisor's name
Sabine Oechsner
First Supervisor

Certified by .....
Second reader's name
Kristina Sojakova
Second Reader

Fairness in Hashcash through Memory Hard Functions

Hendrik Hermann Werner Joel
Vrije Universiteit Amsterdam
Amsterdam, NL
h.h.w.j@student.vu.nl

ABSTRACT

Within the field of computer security, the Hashcash proof-of-work algorithm has played a crucial role, especially in its application within the Bitcoin network. However, concerns about the fairness and equitability of this algorithm have been raised, as there are clear advantages that specialized hardware, such as application-specific integrated circuits (ASICs), have over regular computers. Often specialized hardware is capable of finding the proof of work solution several thousand times faster than a regular desktop. This results in the original concept of a decentralized network to fall apart, as it provides an unfair advantage to certain groups and allows for potential attacks, when the majority of the network is owned by a few individuals. This thesis examines how memory-hard functions, specifically Argon2d, can be implemented into the Hashcash function, to restore fairness. Argon2's memory hard properties can minimize the computational gap between general hardware and dedicated mining equipment, and similar hash rates can be achieved between different devices. Through comprehensive analysis and experimental validation, this paper concludes that memory-hard functions, such as Argon2 can effectively be used within the Hashcash framework and present a viable, fair, alternative to the currently used SHA-256 implementation.

1 INTRODUCTION

The Hashcash proof of work function was originally proposed in 1997 as a means of combating junk mail and spam [4, 10]. Since then it has been used as a consensus algorithm in a variety of different applications, most notably within the cryptocurrency Bitcoin. Due to the rise of application-specific integrated circuits (ASICs) and FPGAs, it has become increasingly more difficult, if not impossible, for regular hardware to participate in solving the Hashcash proof of work equation within cryptocurrencies such as bitcoin. This advantage held by those with access to specialized hardware, not only creates an "unfair" environment, but it allows for theoretical attacks on the Bitcoin network, when a party controls more than 50% of the network power [5]. This inequality arises from the nature of the cryptographic puzzle employed by Hashcash, mainly relying on the SHA-256 hashing function. ASICs, designed to specifically perform SHA-256 computations, have created a barrier of entry for individuals and entities without access to such specialized equipment. To overcome these concerns and issues this thesis investigates the use of so-called Memory Hard Functions (MHFs), specifically the popular function Argon2, as a replacement for SHA-256 within the Hashcash algorithm, to test whether a "fair" environment can be created. Memory-hard functions (MHF) are cryptographic functions designed to require a substantial amount of memory for their computation, making them more resistant to attacks using specifically designed hardware such as ASICS or GPU cracking attempts. Due to memory cost being platform-independent, MHFs play a significant role in a variety of security applications such as password hashing. Memory-intensive computations were first proposed in 2003 as a way to combat the increasing computational power of GPUs and specialized hardware [3, 9]. Using memory as a way to limit computational power has several advantages. Memory latency is similar across different platforms, as long as the same technology is used and memory chips are large and expensive in production. Due to these properties, memory-intensive computation remains an effective way of combating the increasing use of GPUs / FPGAs and ASICs as long as a certain memory-time trade-off remains [5]. MHFs ensure that the overall performance cannot be easily optimized by specialized hardware, thereby maintaining security and fairness, making them ideal for Proof-Of-Work concepts. The primary research question guiding this thesis is: Can Argon2 enhance the fairness and accessibility of the Hashcash Proof-Of-Work algorithm? In order to answer this question, the study will:

- Provide an overview of a basic Hashcash implementation
- Provide an overview of Argon2's properties and internal mechanisms
- Analyze how Argon2 parameter adjustments affect hashing rates and determine parallel computation feasibility limits
- Compare the performance of Argon2 with SHA-256 in a GPU computing environment
- Identify how the obtained results influence a Argon2-Hashcash implementation and whether this creates a fairer environment

This paper's contributions can be summarized as follows:

⋆ Conducting an empirical evaluation to compare the performance of Argon2 with the traditional SHA-256 hashing function in a GPU computing environment, assessing their effectiveness in resisting optimization through specialized hardware

⋆ Provide theoretical insights into the feasibility of implementing Argon2 within Hashcash, as an alternative to SHA256

⋆ Backup the claims made by Argon2 regarding its memory hard properties in a parallelized setting

By addressing these aspects, this research aims to contribute towards establishing methods for creating fairer proof of work mechanisms, as well as validating the memory hard properties of the Argon2 algorithm.

2 BACKGROUND

The upcoming sections will explore the fundamentals of Memory Hard Functions (MHF) and the Hashcash algorithm. This includes an overview of their operations and design principles. Additional insights into GPU execution using the NVIDIA CUDA environment will be covered in section 1, to provide a fundamental understanding. This knowledge is integral for a deeper comprehension of the

experimentation performed and the conclusions drawn at the end of this research paper.

2.1 Memory Hard Functions

An MHF is a hash function that uses a substantial amount of memory to compute its solution, thus limiting the total number of hashing operations that can be performed at the same time within a given system. Memory latency will be similar across systems, as long as the same memory technology is being used, resulting in the parallelization of the algorithm to quickly become infeasible or at least highly inefficient, independent of the system itself [5]. The resulting time-memory trade-off is what memory-hard functions refer to as their memory-hard property. The password-based key derivation function Scrypt is an example of such a Memory Hard Function, and so is Argon2 [18]. Scrypt has also been used in a proof of work setting within the cryptocurrency Litecoin [1]. However it has been shown to not be ASIC resistant as various ASIC architectures and concepts have appeared throughout recent years [14], making it less suitable for use in a proof of work implementation.

To achieve memory hardness most MHFs will operate in the following manner. A given input I is split into several segments, which will then be processed by an underlying, often memoryless, hashing function H, to create a series of memory blocks starting with M_1. Once M_1 has been generated, an indexing function F is then used to generate the subsequent blocks until a final number of blocks M_T has been reached. The output of the MHF is then the hash H of the final computed block M_T. The following equation described in [5] summarizes the process discussed above:

$$\begin{aligned} M_1 &\leftarrow H(\text{Input}), \\ M_j &= F(M_{\phi_1(j)}, M_{\phi_2(j)}, \ldots, M_{\phi_k(j)}), \quad 1 < j \leq T, \\ \text{Output} &\leftarrow H(M_T), \end{aligned} \tag{1}$$

The generation of new memory blocks M_j through the indexing function F forces the system to reiterate over the stored memory blocks, making it necessary for a system to keep the previous blocks actively in memory, as it is unknown which block will be used for the next generation. In turn, a Time-space trade-off is created as the computation increases in complexity the more blocks are stored in memory. Several passes over function F to compute a single block can take place as well, however, this is dependent on the design of the MHF [5].

Relevant to the performance and security of a memory-hard function is the choice of using a data-dependent or a data-independent indexing function F. Algorithms using data-independent indexing functions are often denoted by an appended i, such as Argon2i, whilst data-dependent schemes are often denoted by an appended d (Argon2d). In memory-independent data indexing schemes, the computation of M_j through F is not dependent on the previously computed memory blocks, usually resulting in a faster execution of the MHF itself. However, data-independent schemes open up the possibility of pre-computing attacks, where an attacker attempts to pre-compute all memory blocks simultaneously to gain a time advantage. In data-dependent indexing schemes the output of function F depends on the content of the previously computed memory blocks, for example, the derived index of the new block M_i can be based on the previous block or the current block itself. This unpredictable nature of data-dependent indexing schemes theoretically forces a system to compute every block individually, making the overall computation more complex. However, it must be pointed out that data-dependent indexing schemes are prone to cache-timing attacks [6], where an attacker analyses cache behavior, to leverage certain access patterns that reveal information about how the indexing function F behaves in order to pre-compute blocks and thus shorten the algorithms computational time. It is important to note that the topic of how these hypothetical attacks on an MHF would impact an overlying Hashcash proof of work algorithm and its "fairness" lies outside the scope of this thesis.

2.2 The Hashcash Algorithm

The Hashcash proof of work algorithm was originally proposed as a solution to combating junk and spam mail in 1997 [4], and is used by a system to prove that a certain computational effort has been performed. This computational effort, often referred to as "proof of work", ensures that the sender of an email or a request has expended a certain amount of resources, making it costly for malicious actors to abuse the system. It is important to note that while minor differences between algorithm implementations exist, the core working concept of Hashcash always remains the same, the implementation this paper refers to is the Python implementation written by David Mertz [15].

The core idea behind Hashcash is to require the sender of a packet to compute a hash value that meets certain criteria before the sent packet can be considered valid. This is achieved through the following steps:

- Stamp Creation: The sender generates a "stamp" which can include several pieces of information such as a resource string, a date, or otherwise. The stamp is then used as the main input to a hash function, in this case, SHA256 is used.
- Proof of Work: The sender must then find an arbitrary number (nonce) that, when included in the stamp and hashed, produces a hash value with a specific number of leading zeroes. This part of the process is computationally expensive as it requires a significant amount of trial and error, ensuring that the sender has expended a certain amount of computational effort.
- Verification: Once a hash value with a specific number of leading zeroes has been found its validity can easily be verified by a receiving party. To verify the integrity of a hash the recipient can simply hash the received stamp and check if the resulting hash value has the required amount of leading zeroes. This process is not computationally intensive.

The difficulty of finding a solution to the Hashcash puzzle can be adjusted simply by varying the number of leading zeroes required in the hash value. A higher difficulty level requires more computational effort, making the system adaptable [3, 4]. The difficulty parameter is often divided by four in order to get the number of leading zeroes that will be required in the proof of work. However, tuning the difficulty level to a given system requirement is rather complex, as the overall computational effort required increases in a somewhat exponential manner with the number of leading zeroes required.

2.3 Argon2

Argon2 is a password-hashing function that was selected as the winner of the 2015 Password Hashing Competition (PHC). Its design is aimed to replace older, less secure hashing algorithms and provides a range of security features against brute-force attacks and side-channel attacks [5]. It is especially resistant to GPU and ASIC attacks making it an ideal candidate for the application intended in this thesis. Argon2 was designed to address the shortcomings of older algorithms such as Bcrypt, PBKDF, and Scrypt. These older algorithms were more susceptible to attacks utilizing modern hardware such as GPUs and ASICS [5]. The argon2 algorithm works as following which is outlined in the Argon2 white paper [5]:

- Initialization: The Argon2 function initializes a memory matrix with a size defined by the user through the memory cost parameter. It is worth mentioning that the underlying hash function, H as referenced in equation 1 used by Argon2 is Blake2b.
- Argon2 then fills the matrix through a series of iterations, with each block of memory depending on the previous blocks. A similar process has been outlined above in section 2.1 about MHFs.
- Finalization: After the matrix has been filled, Argon2 produces the final hash value by compressing the matrix into a fixed-length output.

Furthermore, Argon2 is very configurable as it offers the user a variety of parameters to set and adjust the difficulty and memory consumption of the computation. The essential parameters include:

- Memory cost (measured in Kilobytes): This parameter indicates how much memory is taken up by the memory matrix. In essence, this is the limiting factor to how many instances of Argon2 can run on a system.
- Time cost: How often the memory matrix gets iterated over by the indexing function F mentioned in equation 1 in section 2.1. Adjusting this parameter inflates the computational cost of running one iteration of Argon2.
- Parallelism: How many threads are allowed to hash a single instance of Argon2. Increasing this value results in a faster runtime, however, it also increases the system's core usage, again effectively limiting how many instances of Argon2 can be run simultaneously.

Moreover, Argon2 comes in three different variants:

- Argon2d: This variant is optimized to resist GPU cracking attacks by maximizing memory access, making it more difficult for attackers to use hardware with high levels of parallelism. The data-dependent indexing scheme is suitable for applications where potential side-channel attacks are not a major concern.
- Argon2i: The data-independent indexing variant of Argon2, specially designed to be more resistant to side-channel attacks.
- Argon2id: A hybrid version of both the data-dependent and independent indexing schemes, which allows for a versatile option in a lot of different applications.

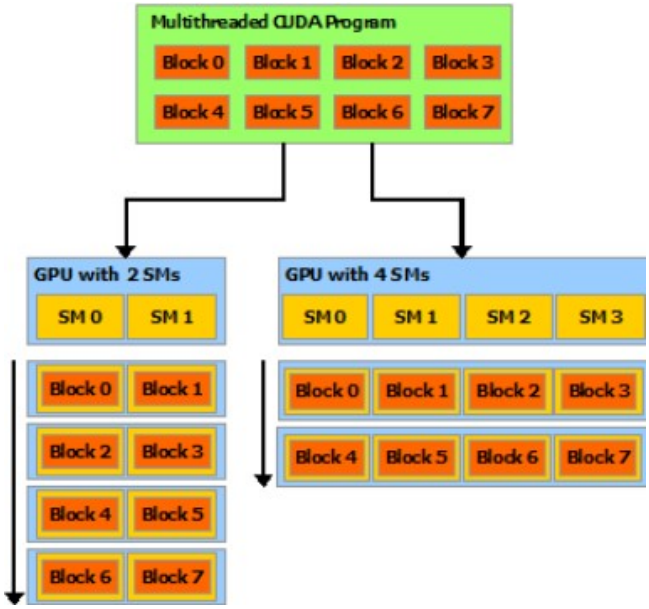


Figure 1: ***Shows the scalability of how a CUDA program uses the architecture of a GPU to divide the workload of an application between several threads [17].***

As a major concern of the Hashcash algorithm is that of increased GPU and ASIC usage, Argon2d is the chosen option to be implemented throughout the experimentation phase of the paper.

2.4 Cuda Implementation

To fully understand the hardware specifications outlined in table 1, this section will provide a brief overview of NVIDIAs CUDA interface. CUDA which stands for Compute Unified Device Architecture, is a parallel computing interface (API) created by the NVIDIA corporation. CUDA enables developers to use NVIDIA graphics processing units (GPUs) for general-purpose processing [17]. As GPUs are generally more capable of high levels of parallelism when compared to regular CPUs, they are often utilized to compute vast numbers of hashing functions in parallel, allowing them to achieve significantly higher hashing rates than a CPU ever could. CUDA provides developers with the ability to more easily access the power of GPU hardware. Due to this, CUDA implementations of the algorithms discussed throughout this thesis were used.

Figure 1 illustrates how a CUDA application uses the cores of a GPU to distribute a workload efficiently. In essence, every multiprocessor of the GPU has several separate CUDA-capable cores, as can be seen in table 1. These cores are referred to as Streaming Multiprocessors (SM). Every SM is then split into different blocks, where each block contains a total of up to 1024 threads [17]. In the context of the Argon2 implementation, every thread is capable of running a single Argon2 instance. Furthermore, SM processors share the same memory space, a property memory-hard functions attempt to utilize as a constraint, to limit the GPU's speed and total throughput.

3 EVALUATION

3.1 Experiment Design

The following overarching sections delve into detail regarding the experimental setup and methodology used to evaluate the feasibility and effectiveness of implementing the Argon2 hash function into the Hashcash Proof-Of-Work algorithm. The experiments, mainly consist of bench-marking tests, which are aimed at assessing the performance of the Argon2 function with the now commonly employed SHA256 hash function, particularly focusing on GPU resistance. Furthermore, the effect of varying Argon2s parameters to achieve more suitable execution times in a Proof-Of-Work setting is investigated. The main goal of these experiments is to demonstrate that within a Proof-Of-Work setting, Argon2 is a more suitable alternative to use than SHA256, as it mitigates the performance differences between varying system architectures. This in turn creates an overall fairer environment for users who lack access to specialized hardware.

3.2 Expected Trends

Throughout the experimentation, several trends are expected to be found within the resulting data. Based on the theoretical properties of Argon2d covered in section 2.3 as well as external research regarding SHA256 [11, 19], several hypotheses can be postulated. If Argon2 is a viable alternative to SHA256 in a Hashcash Proof-Of-Work setting the following trends should be reflected in the gathered data:

- SHA256 should exhibit massive performance boosts when run in a parallelized environment, such as when it is run on GPU hardware.
- In a parallelized environment SHA256 hashing performance should not slow down with an increased level of parallelism
- It is expected for a parallelized SHA256 implementation to be theoretically capable of solving the Hashcash puzzle several thousand times faster when compared to a CPU implementation.
- Argon2 is expected to show a certain degree of resistance to GPU parallelization attempts, hence with increased parallelism the performance of Argon2 should decrease
- Argon2s performance gap should be less, between different systems when compared to SHA256
- Argon2 should allow for a high level of customization, enabling the fine-tuning of the Hashcash puzzles difficulty.

The trends listed above should be reflected in the data gathered, to support the claim, that implementing Argon2 would improve the overall fairness of the Hashcash Proof-Of-Work algorithm.

3.3 Experimental Setup

Due to a lack of access to ASICs and other more specialized expensive hardware, a capable NVIDIA (RTX 4070) GPU, was used to replicate a highly parallelized environment. This approach allows for the performance evaluation of Argon2d and SHA256 in conditions that approximate the capabilities of specialized hardware. Additionally, gathered results and insights into the performance of Argon2d within a Proof-Of-Work system can be extrapolated ,and parameter adjustments can be made to meet higher hardware demands and requirements accordingly. It is important to note that the experiments utilizing the GPU were run through the WSL2 (Windows Subsystem for Linux) virtual environment in order to interface with NVIDIAs CUDA graphics card application programming interface. This may have caused a higher bandwidth and resulted in lower hashing rates being achieved, this will further be discussed in section 4 as a possible limitation. The following tables outline the specifications of the computing hardware and environment in which the experiments were conducted: 2, 1, 3.

Table 1: ***GPU specifications describing the NVIDIA RTX4070, that was used for the performance assessment of Argon2D and SHA256. The data has been gathered through the NVIDIA device query command.***

Characteristic	Value
Total constant memory	65536 bytes
Shared memory per block	49152 bytes
Shared memory per multiprocessor	102400 bytes
Registers per block	65536
Threads per multiprocessor	1536
Threads per block	1024
Global memory	12281 MB (12877955072 bytes)
Multiprocessors	46
CUDA Cores per MP	128
Total CUDA Cores	5888
GPU Max Clock rate	2505 MHz
Memory Clock rate	10501 MHz
Memory Bus Width	192-bit
L2 Cache Size	37748736 bytes

Table 2: ***Processor specifications describing the Intel i7-7700K, that was used for some of the performance assessments. The data derives from the Intel processor data specification sheet [2].***

Characteristic	Value
Physical Core Count	4
Total Threads	8
Max Turbo Frequency	4.50 GHz
Processor Base Frequency	4.20 GHz
Cache	8 MB Intel® Smart Cache
Bus Speed	8 GT/s

3.4 Measurement Metrics

The following metrics have been used to evaluate and analyze the performance and efficiency between instances of Argon2d, SHA256, and the Hashcash implementation itself:

- Hash Rate: The hash rate is most often measured in Kilo Hashes (KH/s) or Mega Hashes (MH/s) and denotes how long it takes for an algorithm, Argon2 or SHA256, to produce a certain number of hashes per second. The value calculated

Table 3: ***System specifications used for the performance assessment of Argon2D and SHA256. This includes details on the operating system and memory configuration.***

Characteristic	Value
Operating System	Windows 11 Pro v.23H2
System Architecture	$x86_64$
Compiler	NVCC 11.5.119
System RAM	16 GB DDR4
WSL Version	2

for the hash rate is based on the mean hash time, referred to here as the HT.

- Batch Size: The batch size refers to the total amount of items that need to be hashed. In a parallelized setting this number is distributed among the available cores and threads. Hence the batch size can be seen as an indicator of the degree of parallelism taking place.
- Hash Count: The hash count, or just count, specifically refers to the nonce count of the Hashcash algorithm. The nonce count is the total amount of attempts it took before computing a valid proof of work, the final hash which is the actual proof of work is included in the count as well.
- Mean Hash Time (HT): The mean hash time (HT) is used to compute and estimate the Hash Rate described above. The HT is the measure of how long it took an algorithm to compute a single hash on average ten times in a row.

The main metric used is the Hash Rate as it is a crucial metric for evaluating the overall performance of a hash function, especially in proof-of-work applications. It furthermore provides insight into the efficiency of the hashing process and allows for the creation of performance estimates, in this case regarding the Hashcash proof-of-work algorithm.

3.5 Experimental Procedure

This section outlines the experimental procedure adopted to evaluate the performance of the Argon2 cryptographic hash function in a proof of work setting. The experiments are performed on a computer system with the specifications listed in tables 2,1, 3, as explained in section 3.3. For the Argon2 performance assessment, the C++ CUDA benchmarking tool of the algorithm by [16] was used. For the assessment of the performance of SHA256 in a GPU setting, a tool was developed in C++ that uses the SHA256 Cuda library developed by [8]. Additionally, the Python implementation of the original Hashcash algorithm by [15] was modified to allow for better performance measurements, and two versions were created to allow for the parallelized use of the Argon2d algorithm, here the Python Argon2 library by [21] was used. Furthermore, several bash scripts were created to ease the use of these benchmarking tools. The same hash lengths and input lengths were used for all algorithms during the experiments, specifically an output hash length of 64. The following experimental procedures took place:

- 1. Establishing a baseline: Both algorithms SHA256 and Argon2d were run on the GPU with a batch size parameter that increased with every step until a certain limit was met, or the program ran out of memory. More specifically Argon2D was run in a one-shot kernel configuration, meaning that it takes one thread to compute an Argon2D instance. Each batch calculation was then repeated 10 times per batch size in order to account for variability and ensure statistical significance. The Argon2d variation was initially run with a default time cost parameter of 1 and a memory requirement of 10240 kibibytes
- 2. Varying Argon2D: Argon2D was then fine-tuned by varying the time cost and memory parameters to test whether the algorithm can be adapted to create an upper bound for a desired hash rate on a certain architecture, in this case, the GPU specified in table 1.
- 3. Establishing a mean hash count: The parallelized Hashcash Argon2d algorithm was then set to a difficulty level of five and run ten times, to ensure a statistical significance. With a maximum of eight available threads as shown in table 2, as much as possible of the CPU was used to generate Argon2d hashes, and the workload was equally distributed between each thread. Again the same parameters for the Argon2d hash function were used as in step 1. This was done to ensure that a similar hash rate per core is achieved and that the same pattern of output hash generation is used by the Argon2d hash function. The hash count of each solution was then averaged out between these ten runs.
- 4. Varying Hashcash: The Argon2d algorithm was run within the parallelized implementation of the Hashcash proof of work framework with a varied degree of processor usage, within the range of one to eight, as the CPU architecture used allows for a maximum usage of eight threads, see 2. Every "run" took place in batches of ten, meaning it was repeated ten times, to account for variations, and the runtime was averaged out.

As mentioned in the first sub-point of the experimental procedures listed above, data was collected until the desired batch size was achieved or the system crashed. These system crashes are indicated by the sudden stop of data being collected as the benchmarking procedures described targeted the same batch size across algorithms. It is important to note that these crashes were the result of the system running out of memory, for example, the GPU upper memory limit lies at around 12.2 GB, as shown in table 1. The crashes were caused by the CUDA benchmarking implementation as it failed to properly queue workloads exceeding the memory limit. In theory, the GPU is capable of computing even larger workloads, however, the hashing rate achieved at the time of a crash will remain constant from that point onwards and no further advantage of the parallelization will be gained. Thus crashes were considered to be a significant part of the data collection, providing insight into when a certain point of computational feasibility has been reached.

3.6 Results

The evaluation of the chosen cryptographic hash functions Argon2d and SHA256, reveal significant insights into their performance in a proof-of-work environment. The subsequent analysis focuses on baseline performance, the impact of parameter variations, and the comparative efficiency of Argon2D and SHA256 as outlined in

further detail in section 3.5. The results highlight key differences and performance trends that are crucial for assessing how viable Argon2 can be in a proof-of-work setting.

Figure 2 shows the baselines created, in kilo hashes (KH/s), for both the Argon2d and the SHA256 cryptographic hash functions. Here Argon2d was run with a time cost parameter of 1 and a memory cost of 10,204 kibibyte KiB.

When first viewing figure 2a one can be tempted to assume that Argon2D outperforms SHA256 up until a batch size of around 1200 is reached. However, this view is deceiving as this is not the case and is the result of the SHA256 workload, the batch size, being far too manageable for the GPU from table 1. The performance gap that was predicted to exist in section 3.2 between running SHA256 on a GPU and Argon2D becomes visible when changing units from Kilohashes to Megahashes (MH/s). Figure 2b depicts SHA256s hash rate in MH/s at a maximum workload. The diagram clearly shows how the parallelized version of SHA256 reaches a hash rate of 40 MH/s before it begins to stagnate, as the upper bound of the GPU and the underlying implementation have been reached. An opposing trend is evident in figure 2a, where the hash rate of Argon2 increases rapidly until it falls off once the point of computational feasibility has been reached. In figure 2a this can especially be seen at a batch size of around 1250, where it becomes inefficient to compute more instances of Argon2D in parallel, as the maximum global memory of the GPU in table 1 is being approached. Hence figure 2a demonstrates Argon2 GPU resistant properties and showcases that through adjusting Argon2Ds memory cost parameter the maximum number of instances computed in parallel can be approximately defined, directly influencing the hash rate itself.

Furthermore Figure 3a supports the finding that the memory cost parameter can be used to define an upper parallelization boundary, effectively limiting the available processing capacity of the computational hardware used to calculate the Argon2 function, in this case, the GPU. As it can be inferred from figure 3a a smaller memory cost results in a higher hash rate and an increased amount of instances that can be run in parallel. Argon2s parallelization parameter can likewise be applied to further fine-tune this boundary.

Conversely, relying solely on the memory cost parameter to adjust the upper computational boundary can be challenging, particularly when trying to limit the upper hash rate, while retaining the set computational boundary. This is evident from the significant gap between the plotted graphs within figure 3a. This gap suggests that it is not possible to adjust the taken-up computing capacity, measured in batch size, without significantly changing the hash rate. To address this issue, the time cost parameter of Argon2, which determines the frequency of memory matrix iterations as detailed in section 2.1, can be adjusted to increase or decrease the upper hash rate boundary. This is further shown in the performed benchmark test diagrammed in figure 3b. Diagram 3b uses two Argon2D functions that were configured with the same memory cost parameters and different time cost parameters. While the occupied batch size remains the same between both plots, the Argon2D instance using a time cost value of 5 has a significantly lower hash rate.

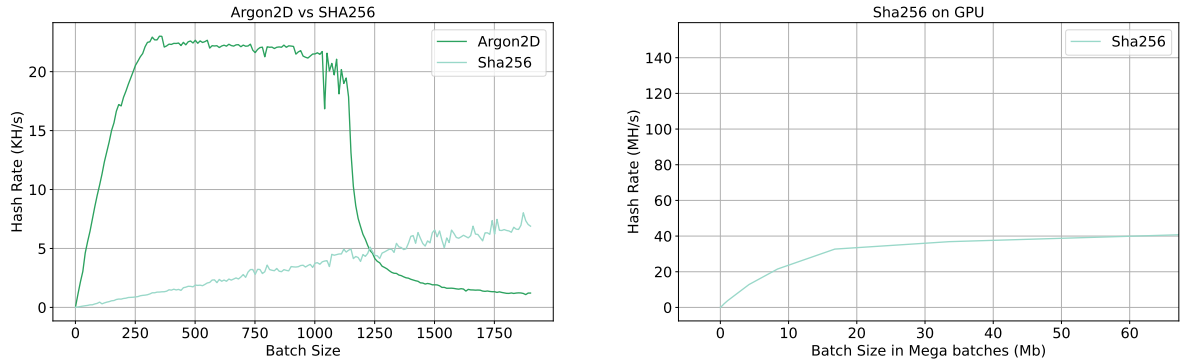
Overall figures 2 and 3 showcase how different parameters of the Argon2 hash function can be fine-tuned to establish an upper processing boundary and a maximum achievable hash rate for a given system, as the one described and used throughout the experimentation in tables, 1, 2 and 3. These two boundaries can then be configured to achieve a desired hash rate or, within the context of a proof-of-work algorithm like Hashcash, can be adjusted to establish a certain time frame within which the solution to the proof-of-work challenge should be found. In short, figure 2 and 3 show that Argon2s parameters can be used to define a time frame in which the Hashcash algorithm will successfully have found a solution, for a given system architecture.

Table 4: ***Hash Count for SHA256 and Argon2D running on the CPU described in table 2 with specified memory cost settings and a set Hashcash difficulty of 5. In this case, a parallelized variation of the Hashcash algorithm was used in order to speed up the computational process.***

Algorithm	Average Hash Count
SHA256	847
Argon2D (Memory cost 10240)	5085
Argon2D (Memory cost 1024)	4205

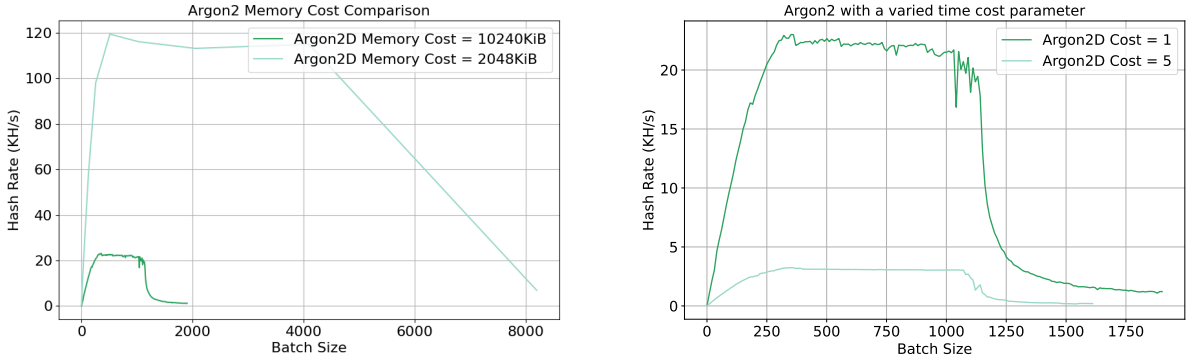
The time frame can be tailored around certain architectures thereby introducing or removing potential advantages. For instance, consider a hardware setup using the CPU outlined in table 2 and 16 Gigabytes of DDR5 random access memory. By configuring the Argon2D parameters to align with this particular hardware configuration, individuals with different setups would face significant penalties. Consequently, more costly computing systems might experience only marginal efficiency improvements compared to the benchmark architecture. For example in order to encourage users to use all eight threads of the CPU described above, the memory cost setting of the Argon2D algorithm could be set to 1,953,125 KiB which would use up all 16GB of RAM when run in parallel. Any system attempting to improve the hash rate even further would thus need to spend an additional 2GB of RAM per instance of Argon2D running, at which point the time cost or parallelization parameters may be adjusted to mitigate any advantage gain. Another option would be to simply increase the difficulty of the hashcash puzzle. This example shows how Argon2D can be adapted to create a more equal environment, by adjusting its memory and time cost settings to favor certain hardware configurations.

This idea of being able to use Argon2D to define a time frame for a proof-of-work to be found is further supported by table 4. Table 4 contains the average hash attempts it took to compute the solution to the Hashcash proof of work algorithm with a set difficulty of 5. Although SHA256 appears to require a lower number of hash attempts before a solution is found, the actual number of required hashes varies wildly and in many cases was shown to be very similar to the number of attempts required by Argon2D. These variations can be attributed to recorded outliers in the data, as the SHA256 algorithm completed the proof-of-work solution anywhere between 65 and 1574 attempts. Thus even though Argon2D on average requires more attempts to find a solution, as seen in table 4, the number of required attempts varies largely every time and is within a similar range to the attempts required when using SHA256.



(a) (b)

Figure 2: ***(a) Showcases the Argon2D function running on a GPU (as described in table 1) in parallel (batch size) plotted against the produced hash rate and compares that with a SHA256 implementation. (b) Depicts the SHA256 function running on a GPU in parallel***



(a) (b)

Figure 3: ***(a) Depicts the Argon2D function with varied memory cost parameters, specifically two Argon2D functions with costs set to 2048 and 10204 KiB. (b) Illustrates two Argon2D functions with the same memory cost parameter of 10204KiB but with different time cost parameters being set, a cost of 1 and 5.***

The increased number of hashing attempts required in Argon2s case is likely the result of the design of its underlying cryptographic function Blake2b [5]. Furthermore, we can see that the average number of hashes that need to be computed for Argon2D to find a proof of work, does not change in a significant manner and it can thus be assumed that this required computational range will remain similar for two iterations of Argon2, that use different memory cost parameters. The average hash count shown in table 4 can further be used to calculate an estimate of how long it would take for a certain instance of Argon2D or SHA256 to solve the proof-of-work. Using the highest recorded Hashrate of SHA256, within the collected data, of 40 MH/s as seen in figure 2b it would take the GPU implementation anywhere between 2.1175×10^{-5} s and 1.27125×10^{-4} s to find a solution to the Hashcash puzzle. This simple example shows how the average hash count can be used to establish a time frame which can then be adjusted using Argon2s time cost, parallelism, and memory cost parameters as mentioned above.

Table 5: ***Shows the performance of the Argon2D algorithm with a set memory cost of 1024, implemented in the Hashcash proof-of-work algorithm with varying difficulty. The mean time refers to the time it took to compute the proof-of-work.***

	Difficulty=5	Difficulty=8	Difficulty=9
Max count	17303	12688	1452020
Min Count	202	109	29809
Mean Count	4698	5187.5	365538
Mean Time	0.479s	0.518s	48.607s

As shown in table 5, the maximum and minimum number of hash attempts required for Argon2D to find a solution to the Hashcash proof-of-work algorithm varies significantly across different

difficulty levels. For instance, at a difficulty level of 9, the maximum number of attempts recorded was 1,452,020 whereas the minimum was 29,809. Similarly, the mean hash counts show how an increase in difficulty results in a large increase in the computational effort required. Additionally, table 5 is another example of how Argon2Ds implementation within the Hashcash proof-of-work algorithm can be used to designate a certain time frame for the proof-of-work to be computed, as mentioned previously. For example, if a solution should be computed on average every 10 minutes, the minimum hash count of a difficulty level could be used as a lower boundary for the required hash rate. One way of calculating the desired hash rate can be illustrated with the following equation:

$$\text{Hashrate/s required} = \frac{\text{Hash Count}}{\text{Minutes} \times 60} \tag{2}$$

If the minimum count is selected as the lower boundary for the desired computing time, the calculation can be adjusted to utilize this value instead of the arbitrary count presented in equation 2. In order to calculate a hash rate for a difficulty level of 5 for a given system to find a proof of work within 10 minutes, assuming that a solution is found after an average of 4,689 hashes, equation 2 can be applied to solve for the required hash rate. Applying this equation to the scenario described yields a required rate of around 7.83 hashes per second, which can effectively be rounded up to about 8 H/s, for a system to find a solution on average every 10 minutes. Now that a time frame and its required hash rate have been established, Argon2 can be fine-tuned to provide said hash rate on a given system architecture. Consider the following hypothetical scenario, where the objective is to incentivize users to perform Argon2 computations up to eight times in parallel. Assuming that the GPU presented in table 1 represents the standard of state-of-the-art hardware available, we can follow the previous example and, for instance, set a memory cost to 1,953,125 KiB. This configuration would hypothetically allow for eight instances of Argon2D to be executed simultaneously on a system with 16GB of available random access memory. However, the GPU described in table 1, with an available memory of approximately 12GB, would be limited to running around six instances of Argon2D concurrently, thus being less efficient than the CPU described in table 2.

Once this memory cost has been defined, the runtime speed of this variant of Argon2D needs to be determined, and respective adjustments to the time cost parameter can be made. If the runtime of a single instance of Argon2D remains too slow to achieve the desired hash rate of 7.83 H/s, the parallelization parameter of Argon2 can be adjusted. This parameter dictates the number of threads that may be used to compute a single Argon2 instance. However, increasing the parallelization parameter would reduce the total number of Argon2 instances running in parallel, effectively trading off the number of threads and total memory required, for a faster runtime.

This example shows how the knowledge gained from figures 2 and 3, along with the data collected from analyzing the behavior of the Hashcash algorithm, as seen in tables 4 and 5, can be used to establish a model of an Argon2 instance that performs similarly across various systems. This resulting Argon2 model can then be implemented within the Hashcash algorithm to achieve a targeted time frame, in which the proof-of-work should be found, independent of the underlying system architecture. It is important to note that the example stated above assumes that the GPU outlined in table 1 represents the most advanced GPU available on the consumer market, which in reality is not the case. Furthermore, in a real-world scenario, it would be beneficial to compute a range of desired hash rates, using the minimum and maximum required attempts, instead of selecting a single rate as the target.

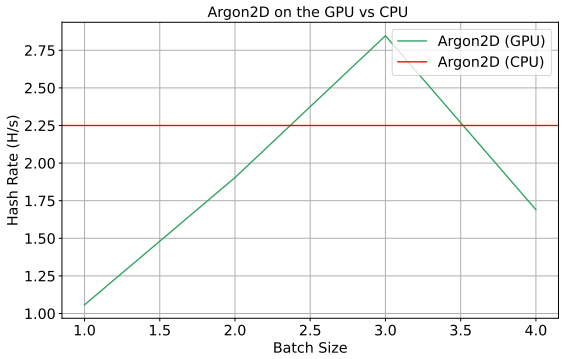


Figure 4: ***Shows the hash rate in H/s for the GPU and the maximum hash rate, as a constant, of the CPU for a given Argon2D instance. The targeted hash rate was roughly around 2 H/s***

The thought experiment outlined above was then simulated on the GPU and CPU, with the target of reaching a hash rate of around 2 - 2.5 H/s. Here the targeted time frame for finding a Hashcash solution was determined to be around 35 minutes, using the average estimate of 4698 hashing attempts, for a Hashcash algorithm with difficulty set to 5, as shown in table 5. Figure 4 compares the performance of an Argon2D instance with the memory cost parameter set to 3,906,252 KiB and the parallelism parameter set to 4, between the GPU and the maximum achieved hash rate of the CPU. As can be interpreted from the figure the desired hash rate of around 2-2.5 H/s has been reached successfully. Furthermore, it can clearly be seen that the GPU does not achieve a significantly higher hash rate when compared to the rate of the CPU.

Overall, the results clearly indicate that an Argon2 instance can be fine-tuned to perform consistently across different systems. Furthermore, it was demonstrated how Argon2 can be adapted in combination with the Hashcash algorithm to achieve a desired hash rate. The data collected clearly shows that, although minor performance differences between different hardware configurations may exist, they do not exhibit the vast discrepancies observed with SHA256. Therefore, Argon2 can in fact be implemented in the Hashcash algorithm, actively contributing to a fairer computational environment.

4 DISCUSSION

The analysis of the Argon2 hashing function within a proof-of-work environment, in this case, the Hashcash algorithm, has provided several key insights into the performance characteristics and implications for achieving computational fairness within the Hashcash

algorithm. This discussion section will delve into the implications of the results and possible limitations, such as security, hardware, and fine-tuning concerns, and will outline areas that could be of interest for future research.

As was shown in section 3.6 the Argon2 hashing function can be adjusted to reach a targeted hash rate that is similar across different systems. It was demonstrated that, unlike SHA256, Argon2 does exhibit GPU-resistant properties through which it ensures that specialized setups will not gain any significant advantage. Furthermore, it was examined how Argon2 can be regulated in order to favor certain hardware architectures, by adjusting the memory and parallelization cost parameters. Besides, it was demonstrated how one can attempt to define a time frame within which the proof of work solution of the Hashcash algorithm can be computed, and it was shown that similar times will be reached independent of the hardware being used. Additionally, as confirmed by the data in figure 4, it can be seen that the Hashcash algorithm exhibits the same GPU-resistant properties as its underlying hash function Argon2. Taking all of the points mentioned above into consideration leads to the conclusion being drawn that implementing Argon2 into the Hashcash algorithm as its underlying cryptographic hashing function, does indeed improve the overall fairness of the proof of work algorithm.

This insight is valuable to anyone utilizing proof of work algorithms, especially Hashcash, as it was shown that Argon2 can be used to create more equal conditions for systems to prove they performed a certain computational cost. This becomes especially relevant within the field of cryptocurrencies, where certain proof-of-work algorithms, such as Hashcash, are vulnerable to a variety of attacks once the integrity of the decentralized system is at stake. Such vulnerabilities, like the 51% attack, place the network at risk, once a majority is able to control more than 50% of the network [13]. Using an Argon2 implementation of Hashcash within such a setting would mitigate the advantage an attacker gains by utilizing specialized hardware such as ASICs or mass GPU cracking attempts.

However, it is not without mention of the several limitations the research conducted within this thesis faces. One of the primary considerations in using Argon2 for proof-of-work algorithms such as Hashcash is its security and resistance to certain attacks. Throughout the experimentation part of this thesis, the Argon2D variant was used, which is known to be viable for side-channel attacks, as mentioned by [5]. If an attacker were to successfully perform a side-channel attack, a potential time advantage could be gained over other systems. As the impact and viability of such side-channel attacks on the performance of the proof-of-work algorithm were not investigated throughout this thesis, it would be of further interest to examine this topic in more detail as part of future research.

While the study demonstrated that Argon2 can indeed be fine-tuned to achieve a similar performance across different systems, there will always be certain limitations that arise due to differences in the hardware being used. Faster computer systems will inevitably have an edge over older and slower systems. Although this performance difference is less pronounced when utilizing Argon2 instead of SHA256, it will nonetheless be present. A limitation of this study was the inability to acquire specialized, more expensive hardware, such as an ASIC setup, to conduct the experimentation. Analysing the performance of Argon2 in the context of a proof of work algorithm, through the use of specialized ASIC hardware instead of a regular GPU may yield different performance results. Further examination of how customized hardware components impact the performance of running the Argon2 algorithm, compared to a regular computer, could provide additional insight into the feasibility of using Argon2 within a proof-of-work context. Another point to consider is how weaker systems are impacted by using Argon2 within the Hashcash algorithm. As discussed in the results section 3.6 the focus of this thesis was primarily placed on making the computation of Argon2 challenging for systems more capable than the defined hardware configuration used. Hence systems weaker than the setup described in table 1, were not considered throughout the analysis.

Moreover, changes in the underlying system structure deviating from the system described in table 3 may lead to performance differences. All benchmarking tests performed were run within the Windows Subsystem for Linux (WSL2). This also means that the graphics card was accessed through the WSL2 environment, which may have impacted performance. Furthermore, as the Python implementation of the Hashcash algorithm was used, for the CPU tests, it is to be expected that due to the interpreted nature of the language, a slower performance was achieved. Other more efficient GPU and CPU implementations may result in even higher hash rates and faster run times. Due to these hardware limitations, it is difficult to project the achieved result onto actual real-world applications. Although the experiments performed proved the concept of being able to create a fairer environment by implementing Argon2 within the Hashcash algorithm and outlined how the selection of parameters could be considered, they are in no way representative of how the Hashcash algorithm performs in a high-performance environment. For this further testing with better equipment and specialized hardware would be needed.

Due to the mentioned limitations above it would be interesting to conduct further research into several of these topics. For instance, fully analyzing the security vulnerabilities of the Argon2 algorithm and its underlying hash function Blake2b. This could provide significant insights into the effects of possible attacks on a Hashcash implementation utilizing the Argon2 function. Additionally one could experiment with implementing Argon2 into other proof-of-work algorithms. Implementing a memory-hard function into a proof of work concept that allows for a finer tuning of the difficulty setting than Hashcash, could be another interesting topic of future research.

Given these findings and limitations, it becomes clear that further exploration and testing are needed to fully understand and optimize the use of Argon2 in a proof-of-work system. Nevertheless, the current analysis underscores the potential of Argon2 to improve computational fairness and security aspects within proof-of-work applications such as their use in decentralized networks.

5 RELATED WORK

Exploring the concept of integrating memory-hard functions into proof-of-work algorithms has been an area of active research, especially with the rise of cryptocurrencies and blockchain technologies.

Studies such as, [12] have examined the performance of several proof-of-work algorithms like CryptoNight, Ethash, and Cuckoo Cycle on specific CPU architectures. However, it is noteworthy to mention that the memory hard function Argon2 was not examined in their research. In contrast, this thesis places a significant focus on the Argon2 hash function, analyzing its potential as the carrying function of the Hashcash proof of work algorithm.

Moreover, studies such as [5] have proposed the Argon2 hash function itself. The paper suggests how Argon2 has been designed specifically to be used in the proof of work schemes. The paper illustrates how Argon2 can demonstrate GPU and ASIC-resistant properties. This thesis builds on the foundational concepts explained by [5], not only considering the theoretical aspects of Argon2 as outlined but by implementing it into the Hashcash proof of work algorithm.

Other areas of research have placed emphasis on creating ASIC-resistant proof of work algorithms as well. So has the research of [20], where new methods for developing ASIC-resistant hashing functions are being examined. One of these examples, the bandwidth hard function (BHF), contrasts the memory hard hashing schemes analyzed throughout this thesis. Further examination of how BHF hashing functions could be implemented into a proof of work system could yield valuable research insights, into creating a fairer overall computing environment.

It is also important to point out that Argon2 itself has been implemented into several proof of work schemes, including notable examples such as MTP-Argon2 and Itsuku [7], which highlights the growing interest in leveraging the memory hard properties of algorithms such as Argon, to enhance the security and fairness of various proof of work algorithms and cryptocurrency applications. Likewise, several attempts at implementing memory-hard functions within cryptocurrencies have already been made, such as the usage of Scrypt within the cryptocurrency of Litecoin [1].

It can confidently be said that the research on integrating memory hard, ASIC-resistant, functions like Argon2 into proof-of-work algorithms is ongoing. Future studies could explore and perform a more comprehensive security analysis, the impact of different hardware configurations, and the scalability of such algorithms in the context of large decentralized networks. This thesis has contributed to this ever-evolving field by providing a detailed examination of implementing Argon2 into the Hashcash proof of work algorithm, as well as suggesting directions for future research to build on as already pointed out throughout section 4.

6 CONCLUSION

The analysis of the Argon2 hashing function within the context of a proof-of-work environment, specifically within the Hashcash algorithm, has yielded several significant insights into its performance characteristics and potential to enhance computational fairness across different systems in a proof-of-work environment. The results have shown that Argon2 can be fine-tuned, to perform similarly across different hardware configurations. This in turn provides a more level playing field when compared to traditional hashing functions such as SHA256.

Throughout the study, it was demonstrated that Argon2 exhibits properties that make it resistant to being heavily parallelized, thereby reducing the time advantage gained by specialized hardware setups. This resistance is critical to ensuring that the proof of work algorithm performs similarly, regardless of the hardware employed by participants. Furthermore, it was shown that adjusting the parameters of Argon2 allows for further tailoring of the algorithm to favor certain hardware architectures.

The data analyzed showed that while minor performance differences due to hardware variations exist, they are less pronounced when using Argon2 compared to SHA256. This is a strong indication that Argon2 is a viable option for creating more fair and secure proof of work systems.

However, several limitations were identified, as talked about in section 4. Limitations such as the potential vulnerability of Argon2D to side-channel attacks and the inherent performance advantage of faster computing systems. These limitations highlight the need for further research into the security aspects of Argon2 and its underlying hashing function Blake2b, as well as further analysis utilizing highly specialized computing hardware, such as ASIC setups.

In conclusion, implementing Argon2 within the Hashcash algorithm offers a promising approach to enhancing the fairness and security of proof of work systems.

All code used throughout this thesis can be found at https://github.com/Schorle21/BachelorThesis_Code/tree/main.

REFERENCES

[1] [n.d.]. https://litecoin.info/docs

[2] [n.d.]. https://www.intel.com/content/www/us/en/products/sku/97129/intel-core-i77700k-processor-8m-cache-up-to-4-50-ghz/specifications.html

[3] Martin Abadi, Mike Burrows, Mark Manasse, and Ted Wobber. 2005. Moderately hard, memory-bound functions. *ACM Trans. Internet Technol.* 5, 2 (may 2005), 299–327. https://doi.org/10.1145/1064340.1064341

[4] Adam Back et al. 2002. Hashcash-a denial of service counter-measure. (2002).

[5] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. 2015. Fast and Tradeoff-Resilient Memory-Hard Functions for Cryptocurrencies and Password Hashing. Cryptology ePrint Archive, Paper 2015/430. https://eprint.iacr.org/2015/430 https://eprint.iacr.org/2015/430.

[6] Alex BIRYUKOV and Dmitry KHOVRATOVICH. December 2015. Tradeoff Cryptanalysis of Memory-Hard Functions. In *21st International Conference on the Theory and Application of Cryptology and Information Security* (Auckland, New Zealand). Springer. https://doi.org/10.1007/978-3-662-48800-3_26

[7] Fabien Coelho, Arnaud Larroche, and Baptiste Colin. 2017. *Itsuku: a memory-hardened proof-of-work scheme.* Ph.D. Dissertation. MINES ParisTech-PSL Research University.

[8] Brad Conte. 2019. Cuda-hashing-algorithms. https://github.com/mochimodev/cuda-hashing-algos/

[9] Cynthia Dwork, Andrew Goldberg, and Moni Naor. 2003. On Memory-Bound Functions for Fighting Spam. In *Advances in Cryptology - CRYPTO 2003*, Dan Boneh (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 426–444.

[10] Cynthia Dwork and Moni Naor. 1993. Pricing via Processing or Combatting Junk Mail. In *Advances in Cryptology — CRYPTO' 92*, Ernest F. Brickell (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 139–147.

[11] M. Dürmuth and T. Kranz. 2015. On password guessing with gpus and fpgas. *Technology and Practice of Passwords* (2015), 19–38. https://doi.org/10.1007/978-3-319-24192-0_2

[12] Zonghao Feng and Qiong Luo. 2020. Evaluating memory-hard proof-of-work algorithms on three processors. *Proceedings of the VLDB Endowment* 13, 6 (2020), 898–911.

[13] Thomas P. Keenan. 2017. Alice in Blockchains: Surprising Security Pitfalls in PoW and PoS Blockchain Systems. *2017 15th Annual Conference on Privacy, Security and Trust (PST)* (2017), 400–4002. https://api.semanticscholar.org/CorpusID:52917879

[14] Duc Khai Lam, Vu Trung Duong Le, and Thi Hong Tran. 2022. Efficient Architectures for Full Hardware Scrypt-Based Block Hashing System. *Electronics* 11, 7 (2022). https://doi.org/10.3390/electronics11071068

[15] David Mertz. [n.d.]. Hashcash Python library. http://www.hashcash.org/libs/python/

[16] Ondrej Mosnáček. 2017. Argon2-GPU Benchmark tool. https://github.com/WebDollar/argon2-gpu/blob/master/LICENSE

[17] NVIDIA. 2024. Cuda C++ Programming Guide. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[18] Colin Percival. 2009. Stronger key derivation via sequential memory-hard functions.

[19] H. L. Pham, T. H. Tran, T. D. Phan, V. T. D. Le, L. D. Khai, and Y. Nakashima. 2020. Double sha-256 hardware architecture with compact message expander for bitcoin mining. *IEEE Access* 8 (2020), 139634–139646. https://doi.org/10.1109/access.2020.3012581

[20] Ling Ren and Srinivas Devadas. 2017. Bandwidth Hard Functions for ASIC Resistance. In *Theory of Cryptography*, Yael Kalai and Leonid Reyzin (Eds.). Springer International Publishing, Cham, 466–492.

[21] Hynek Schlawack. 2015. CFFI: Argon2 for python. https://argon2-cffi.readthedocs.io/en/stable/index.html