Information Sciences 507 (2020) 161–171

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

A flexible method to defend against computationally resourceful miners in blockchain proof of work

Wei Ren[a,b,c,*], Jingjing Hu[a], Tianqing Zhu[a,d], Yi Ren[e], Kim-Kwang Raymond Choo[f]

[a] School of Computer Science, China University of Geosciences, Wuhan, China
[b] Hubei Key Laboratory of Intelligent Geo-Information Processing, China University of Geosciences, Wuhan, China
[c] Guizhou Provincial Key Laboratory of Public Big Data, GuiZhou University, Guizhou, China
[d] School of Software, University of Technology Sydney, Ultimo, NSW 2007, Australia
[e] School of Computing Science, University of East Anglia, Norwich NR4 7TJ, UK
[f] Department of Information Systems and Cyber Security, University of Texas at San Antonio, San Antonio, TX 78249-0631, USA

ARTICLE INFO

Article history:
Received 22 February 2019
Revised 8 August 2019
Accepted 11 August 2019
Available online 13 August 2019

Keywords:
Pow
Blockchain
Hash
ASICs

ABSTRACT

Blockchain is well known as a decentralized and distributed public digital ledger, and is currently used by most cryptocurrencies to record transactions. One of the fundamental differences between blockchain and traditional distributed systems is that blockchain's decentralization relies on consensus protocols, such as proof of work (PoW). However, computation systems, such as application specific integrated circuit (ASIC) machines, have recently emerged that are specifically designed for PoW computation and may compromise a decentralized system within a short amount of time. These computationally resourceful miners challenge the very nature of blockchain, with potentially serious consequences. Therefore, in this paper, we propose a general and flexible PoW method that enforces memory usage. Specifically, the proposed method blocks computationally resourceful miners and retains the previous design logic without requiring one to replace the original hash function. We also propose the notion of a memory intensive function (MIF) with a memory usage parameter k (kMIF). Our scheme comprises three algorithms that construct a kMIF *Hash* by invoking any available hash function which is not kMIF to protect against ASICs, and then thwarts the pre-computation of hash results over a nonce. We then design experiments to evaluate memory changes in these three algorithms, and the findings demonstrate that enforcing memory usage in a blockchain can be an effective defense against computationally resourceful miners.

1. Introduction

Cryptocurrencies, particularly Bitcoin, are a disruptive innovation for e-payments and value exchanges in banking and finance-related fields [30,34]. In Bitcoin and Bitcoin-like public blockchain systems, one of the most critical components is the consensus mechanism. The latter regulates the miners who try to link blocks onto the chain in exchange for a reward (e.g., a Bitcoin). Currently, proof of work (PoW) is one of the most widely used consensus mechanisms. PoW conducts

* Corresponding author at: School of Computer Science, China University of Geosciences, Wuhan, China
E-mail address: weirencs@cug.edu.cn (W. Ren).

https://doi.org/10.1016/j.ins.2019.08.031
0020-0255/

hashing computing to find a nonce that satisfies a given requirement. The first individual to find the nonce is the successful miner and can link the block onto the blockchain for their reward (i.e., a coin-based transaction).

As such, it has become widespread practice to use dedicated Bitcoin mining hardware [1], such as an application-specific integrated circuit (ASIC) machine, to accelerate finding the nonce; thus, increasing mining success. ASIC machines are specifically designed and hardware-optimized to perform target hashing functions (e.g., SHA256) at significantly increased speed and with very low operating costs. This also increases the risks of "double spending attack" and "51% attack".

It is, therefore, unsurprising that ASIC machine-facilitated attacks, and how to prevent such attacks, have become a subject of intense study (e.g., [20,25,36]). One remedy is to replace the hash function with multiple hashing computations [20] on the premise that an ASIC machine can only be optimized for a few target hash functions, not many or all of them. Percival and Josefsson [25] took a different approach with a new hash function that consumes a large amount of memory, since the memory in ASIC machines is typically very limited. While promising, these solutions present an interesting dilemma. Approaches based on non-target hash functions, such as those described in [20,25,36] require the replacing of current hash functions with either a new hash function or several hash functions, and the hard-fork this creates is onerous. Furthermore, these newer alternative hash functions have yet to stand the test of time and may not be as secure as existing hash functions, such as SHA256. Conversely, proposing viable solutions to new attacks is already challenging, and restricting developers to only the minimum design changes is perhaps too constraining. Hence, instead of replacing the hash function to defend against ASICs, we propose a flexible method that stores intermediate hash values in the memory and enforces memory consumption for computing the result (or in searching of the final nonce). The proposed solution preserves legacy hash functions, like SHA256, as well as previous design logics, such as flexibility for tuning mining difficulties. The only change is a slight modification to the computation method for hash values.

In summary, our proposal includes:

1. A general PoW method designed to defend against ASICs without the need to replace the underlying concrete hash function.
2. A flexible PoW method that guarantees the least amendments to the design logics in enhanced blockchain systems.

From empirical recordings of the data length changes in the three algorithms before and after $k\mathrm{MIFH}(\cdot)$ in the two schemes, we find that $k\mathrm{MIFH}(\cdot)$ consumes concrete memory in size of configurable parameter k.

The rest of the paper is organized as follows. Sections 2 and 3 present relevant background and problem formulation, respectively. In Sections 4 and 5, we present our proposed scheme and the evaluation findings. Related literature is discussed in Section 6. Finally, Section 7 concludes this paper.

2. Background

2.1. ASIC

ASICs are chips designed and customized for a specific use. In cryptocurrency mining, this "specific use" is calculating a particular hash algorithm, so an ASIC miner is a professional mining device built from these chips. Since each cryptocurrency has its own cryptographic hash algorithm, an ASIC miner is usually dedicated to mining the particular hash algorithm of a particular cryptocurrency. For instance, Bitcoin ASIC miners are specifically designed to calculate the SHA256 hash algorithm. Given the same equipment cost in a CPU or GPUs, ASICs are far more efficient in terms of performance and energy consumption [31].

2.2. PoW in bitcoin

Bitcoin is, arguably, the poster boy of blockchains. As previously discussed, PoW is the main consensus mechanism in Bitcoin applications and generally in current cryptocurrencies. PoW requires all miners to find solutions by computing hard puzzles based on cryptographic hashes. Finding the right answer means the next block is appended to the blockchain [22,24].

The puzzle can be described as follows: Given data D, find a number N such that the hash result of N appended to D is a number less than T. For example, $hash()$ is a hypothetical hash function. Its outputs are listed as below:

$D = 'test', T = 16$
$hash(D||1) = hash('test1') = 0xb8 = 184 > 16$
$hash(D||2) = hash('test2') = 0x5a = 90 > 16$
$hash(D||3) = hash('test3') = 0x0c = 12 < 16$
$N = 3$

Solved successfully.

To verify the validity of the block, other nodes check whether the hash result of a new block is less than a preset target. The POW procedure is illustrated in Fig. 1.

Given the attributes of a hash function, the only way to solve a hash puzzle is a brute force search, i.e., trying all possible values of n. That means the first miner to solve the puzzle is usually the one with the most computing power. It is a reinforcing cycle, the miner obtains cryptocurrency as a reward and is encouraged to continue mining.

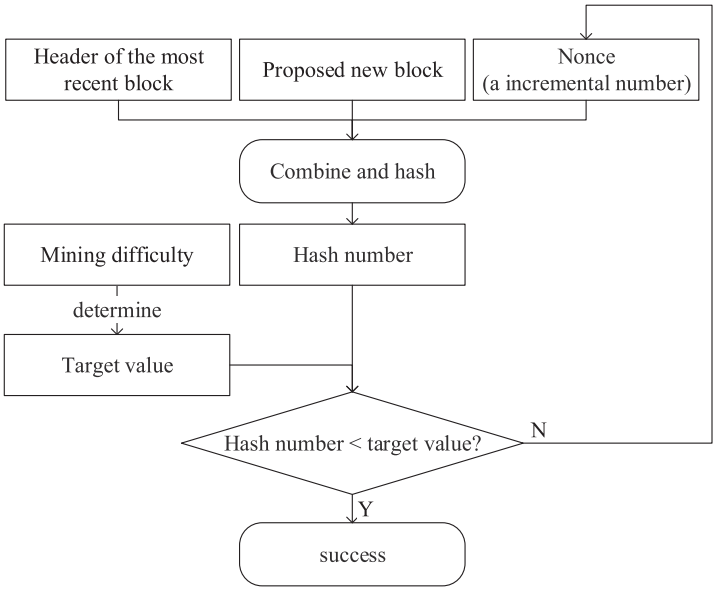

Fig. 1. PoW in blockchain systems.

Table 1
List of notations.

Notation	Comments
$\|x\|$	the length of x
$Hash(\cdot)$	one way hash function
t	the length of hash function output, that is, $t = \|Hash(\cdot)\|$
s	the segmentation length of the hash output. Note that, s divides t, i.e., $s\|t$
$S[i]$	the value of the segmentation
Nonce	the value to be found in mining
C	the first value in the current block available for computing the *Nonce*
R	the hash value required in the mining competition
$f_p(\cdot)$	a function determined by a system parameter

3. Problem formulation

Given the current block information C, a PoW miner will attempt to find a *Nonce* such that $Hash(C||Nonce) < R$ through random trials, where R is the required hash value in the mining competition, and $Hash(\cdot)$ is a one-way function. $Hash(\cdot)$ has the following properties: given x, it is easy to compute $Hash(x)$; but given $Hash(x)$ it is hard to compute x. The notations used in this paper are listed in Table 1.

Definition. (the kMIF$(\cdot)$ memory intensive function and memory usage parameter k) kMIF$(\cdot)$ is a function that must consume at least k memory in its computing operation, where the memory scale can be bits, bytes, KB, etc.

The desired solution should be independent of the hash function. That is, $Hash(\cdot)$ should be unchanged. Our construction of kMIF treats $Hash(\cdot)$) as a building block (see kMIFH$(\cdot)$). That is, the function is required to use at least k memory.

The above original inequality remains unchanged. However, kMIF is built using the original components. The design goals are as follows: 1) defend against ASICs; 2) only rely on currently available design components, namely,$Hash(\cdot)$, X, *Nonce*, and R; and 3) retain former design logics with the least amendments.

4. Proposed schemes

4.1. Basic scheme

Proposition 4.1. *To build kMIFH by invoking a hash function that is not kMIF, the intermediate results of the hash function computation must be memorized, at least until k memory is used.*

Proof. If an individual hash function is not kMIF, it consumes less than k memory. To build a kMIF by invoking a hash function that is not kMIF, memory must be aggregated to at least k memory by accumulating the consumed memory of the

underlying hash functions. If the hash function results are persistent, this consumed memory will both reliably contain the hash function results and allow the accumulation of k memory, which can be achieved by letting the final result of kMIF depend on the persistent results. Once the calculation is complete, the accumulated memory can be released. □

Proposition 4.2. *Suppose the underlying hash function is* $\|Hash(\cdot)\| = t$. *In building a* kMIFH *through* $Hash(\cdot)$, *the results of* k/t *hash functions must be persisted to consume* k *memory.*

Proof. Each hash function result consumes t memory. If the memory is persist rather than being released, the hash function results of k/t hash functions are able to accumulate to k memory. □

For example, suppose we build kMIFH using SHA256, and the length of SHA256 is 256 bits. Then, the results of the four hash functions will consume 1024 bits.

Thus, our focus is on how to construct a kMIFH using a hash function with $\|Hash(\cdot)\| = t$, in which all intermediate hash function results are persistent, not released.

Proposition 4.3. *A, B are both required for the final result of* kMIFH. *If A is used before B in the computation, and B can be computed by A, then B can be not persistent as it can be computed on the fly upon request.*

Proof. Straightforward. □

For example, $Hash(A)$ and A are both used for computing the final result of kMIFH, and $Hash(A)$ must be used before A. Thus, the memory of A is required for computing the final result. For a concrete example, $\mathrm{MIFH}(\cdot) = Hash(Hash(A)\|A)$.

This observation can be generalized as follows in Proposition 4.4.

Proposition 4.4. *A* kMIFH *can be constructed using* $Hash(\cdot)$ *as follows:*

$$k\mathrm{MIFH}(X) = Hash(X),$$

$$X \Leftarrow Hash^n(A)\|Hash^{n-1}(A)\| \ldots \|Hash(A),$$

where $n = k/t, t = |Hash(\cdot)|$, $A \in \{0, 1\}^*$.

Proof. If n hash values are not stored (namely, $Hash^i(A), i = 1, 2, \ldots, n$), the computation for kMIFH is $1 + n + (n-1) + \ldots + 1 = 1 + n(n+1)/2$ times the hash computation. If n hash values are stored, then the computation of kMIFH is $n+1$ times the hash computation, in which the cost of $Hash^i(A), i = 1, 2, \ldots, n$ is n times of - (suppose $Hash^1(A) = Hash(A)$) and the cost for the final outer hash function is one. □

Therefore, the tradeoff for miners becomes $O(n^2)$ computation plus 1 memory or $O(n)$ computation plus n memory. As miners are competing for rewards, they need to compute faster with the same budget, and the price of computing goes up as the memory required increases. Obviously, n^2 mining machines plus 1 memory will be more expensive than n mining machines plus n memory.

This only leaves parallel computation as an avenue of attack, which we defend against with the following computation structure:

$$\mathrm{MIFH}(X) = OWF(X),$$

$$X \Leftarrow OWF(Rightside)\|RightSide = OWF(A).$$

Remark

(1) *OWF* is a one-way function. $Hash(\cdot)$ in Proposition 4.4 can also be generalized to *OWF*.
(2) The construction of intermediate X defines the structure of inputting argument for MIFH (namely, X), noting that the structure is *iterative*.
(3) $Hash(\cdot)$ is an instantiation of *OWF*. Thus, for simplicity, we simply refer to $Hash(\cdot)$.
(4) For example, consider $\mathrm{MIFH}(X) = Hash(X), X \Leftarrow Hash(Hash(A))\|Hash(A))$;
Or, $X \Leftarrow Hash(Hash(Hash(A))\|Hash(A))\|Hash(Hash(A))\|Hash(A)$.

This computation of X is created by
$X \Leftarrow Hash(A)$;
$For(i = 1; i < n; i++)$ $\{Y \Leftarrow Hash(X)$; $X \Leftarrow Y\|X$; $\}$
where n is a system parameter.
That is, $X = Hash(Hash(A))\|Hash(A)$ when $n = 2$, and
$X = Hash(Hash(Hash(A))\|Hash(A))\|Hash(Hash(A))\|Hash(A)$ when $n = 3$.

Proposition 4.5. $\mathrm{MIFH}(X) = Hash(X)$,
$X \Leftarrow Hash(Rightside)\|Rightside = Hash(A)$ *iteratively can defend against parallel computation.*

Proof. *Hash*(·) is one-way. Thus, the *Rightside* must be computed first (namely, it is available) Once computed, the computation for *Hash*(*Rightside*) can begin. That is, the computation of the final result can only be computed in serial, not in parallel. □

Further, we observe that in computing of *Hash*(*Hash*(*A*)), if the first segment in the head of *Hash*(*A*) is available, then the computation of first segment in the head of *Hash*(*Hash*(*A*)) can begin. This is called a "pipeline" computation. Generally speaking, defending against parallel and pipeline computing means, $f(f(\cdot))$ should be changed to $f(\overline{f(\cdot)})$, where $\overline{f(\cdot)}$ is the inverse bit sequence of $f(\cdot)$.

Proposition 4.6. *(Defending against parallel and pipeline computing.) A k*MIFH *can be constructed using Hash*(·) *as follows:*
$k\text{MIFH}(X) = Hash(X)$, *X is created by*
$X \Leftarrow A$;
$For(i = 1; i < n; i++)$
$\{Y \Leftarrow \overline{Hash(X)};\ X \Leftarrow Y\|X;\ \}$,
where $n = k/t$, $t = |Hash(\cdot)|$, $A \in \{0, 1\}^*$, *and* $\overline{Hash(X)}$ *is the inverse bit sequence of Hash*(*X*).

Proof. If n hash values are not stored, then the computation for kMIFH is $1 + n + (n-1) + \ldots + 1 = 1 + n(n+1)/2$ times the hash computation. If n hash values are stored, then the computation the kMIFH is $n+1$ times of hash computation, in which n is the number of inner hash computations and 1 is the final outer hash function.

Using $\overline{Hash(X)}$ instead of $Hash(X)$ guarantees that the computation for $Y\|X$ can only start after $\overline{Hash(X)}$ has been fully computed, i.e., when the last segment of $\overline{Hash(X)}$ is available. □

Thus, our proposal for a basic scheme is as follows:
Given C and the mining target of a *Nonce* that guarantees a requirement in terms of R, the process for the PoW is:

1. Randomly select a $Nonce \in \{0, 1\}^{|Nonce|}$.
2. Prepare X.
   $X \Leftarrow Nonce$;
   $For(i = 1; i < n; i++)$
   $\{Y \Leftarrow \overline{Hash(X)};\ X \Leftarrow Y\|X;\ \}$,
   where $n = k/t$, $t = |Hash(\cdot)|$, $A \in \{0, 1\}^*$, $\overline{Hash(X)}$ is the inverse bit sequence of $Hash(X)$.
3. Compute final result. $k\text{MIFH}(X) = Hash(C\|X)$.
4. If $k\text{MIFH}(X) < R$, then the *Nonce* is found. Otherwise, try another *Nonce* (i.e., go to Step (1)).

This basic PoW procedure is outlined in Algorithms 1 and 2, the difference between the two being the method used to compute the hash function.

Algorithm 1: Basic Scheme(1).

Data:
C ← the available value in the current block
k ← memory needs to be consumed
t ← the length of the hash function
R ← the requirement of the hash value
Result: *Nonce*; *hash* when success

```
for Nonce ← 1 to max − nonce do
    X = str(Nonce)
    n = k/t
    output begin length
    for i ← 1 to n do
        for j ← 1 to n do
            h = Hash(x)
        end
        X = X||h
    end
    output end length
    Z = Hash(C||X)
    if Z > R then
        output
    else
        continue
    end
end
```

Algorithm 2: Basic Scheme(2).

Data:
$C \leftarrow$ the available value in the current block
$k \leftarrow$ memory needs to be consumed
$t \leftarrow$ the length of the hash function
$R \leftarrow$ the requirement of the hash value
Result: *Nonce*; *hash* when success

```
for Nonce ← 1 to max − nonce do
    X = str(Nonce)
    n = k/t
    output begin length
    for i ← 1 to n do
        h = Hash(x)
        Y = reverse(h)
        X = Y||X
    end
    output end length
    Z = Hash(C||X)
    if Z > R then
        output
    else
        continue
    end
end
```

The difference between two algorithms in Basic Scheme is different methods for hash function computation.

4.2. Advanced scheme

To thwart pre-computing of the hash results over a *Nonce*, we have also prepared an advanced scheme in which a random segment is selected and included with *Nonce*. The procedure is as follows:

Consider C and a mining target of a *Nonce* that guarantees a requirement in terms of R.

1. Randomly select a $Nonce \in \{0, 1\}^{|Nonce|}$.
2. Select a random segment in $Hash(Nonce)$ and include it with the *Nonce*.
   2.1. Compute $Z \Leftarrow Hash(Nonce)$. $t \Leftarrow |Z|$.
   2.2. Cut Z into segments with length s. $s|t$. Each segment is denoted as $S[i]$. Obviously, $S[i] \in [0, 2^s - 1]$, $i = 0, 1, \ldots, t/s - 1$.
   2.3. $i \Leftarrow [\frac{t}{2*s}]$.Return the largest integer not larger than x.
   2.4. $i \Leftarrow S[i] + S[t/s - 1] \bmod t/s$.
3. Prepare X.
   3.1. $X \Leftarrow C \| Nonce \| S[i]$;
   3.2. $For(i = 1; i < n; i++)$
   $\{Y \Leftarrow \overline{Hash(X)};\ X \Leftarrow Y \| X;\ \}$,
   where $n = k/t$, $t = |Hash(\cdot)|$, $A \in \{0, 1\}^*$, $\overline{Hash(X)}$ is the inverse bit sequence of $Hash(X)$.
4. Compute the final result. $k\text{MIFH}(X) = Hash(X)$.
5. If $k\text{MIFH}(X) < R$, then the *Nonce* is found. Otherwise, try another *Nonce* (i.e., go to Step 1).

Remark

(1) In step (2.4), we assume $2^s - 1 \le t/s - 1$. That is, $2^s \le t/s \Rightarrow 2^s * s \le t$. For instance, suppose $t = 256 = 2^8$, and we have $2^5 * 5 \le 256$, but $5 \nmid 256$, $2^6 * 6 > 256$. Thus, for $t = 256$, we have $s = 4$. When $t = 512 = 2^9$, we have $2^7 * 7 > 512$, $2^6 * 6 < 512$ and $6 \nmid 512$. Thus, we also have $s = 4$ for $t = 512$. Usually, we have $t = 2^u$. $2^s * s \le 2^u \Rightarrow s * \log_2 s \le u$.

(2) $S[i] + S[t/s - 1] \bmod t/s$ is the summation of two segments: one is the segment in the middle location of $S[i], i = 0, 1, \ldots, t/s - 1$, and the other is the last segment in $S[i]$. These two segments are both in the range of $[0, 2^s - 1]$, and the index (i.e., i) is obtained by summing the final selected $S[i]$ after the modulo operator t/s.

In addition, we propose two methods of further thwarting pre-computation. The basic idea is to let the computation depend on some additional and unpredictable information but that still relates to the *Nonce*.

(1) $S[i]$ in Step 3.1 can be replaced with $f_p(S[i])$, where $f_p(\cdot)$ is a function determined by instantly generated information from the *Nonce*, e.g., $f_0(x) = x^2$ for $p = 0$, $f_1(x) = 2^x$ for $p = 1$. p is determined by the last p bits of *Hash*(*Nonce*) and $\|f_p(\cdot)\| = 2^p$, which means $f_p(\cdot)$ is of the kind 2^p.

For example, suppose there exist $2^p = 2^3 = 8$ sorts of functions. Denote them as $f_0(\cdot), f_1(\cdot), \ldots, f_7(\cdot)$. Suppose $f_0(x) = x^2$, $f_1(x) = 2^x$, and so on. If the last $p = 3$ bits of *Hash*(*Nonce*) is 001, then $f_1(x) = 2^x$ will be selected. That is, $f_1(S[i]) = 2^{S[i]}$ will be used in Step 3.1.

(2) In Step 3.1, i.e., $X \Leftarrow C\|Nonce\|S[i]$, $S[i]$ could also be selected from a segment in *Hash*(*Nonce*) *on the fly* as follows:

Search for the first occurrence of E in a hex representation of *Hash*(*Nonce*) from the rear, and let all bits in the front of E in *Hash*(*Nonce*) be $S[i]$. Note this enhancement means, Step 2 in the advanced scheme can be avoided.

For example, suppose *Hash*(*Nonce*) = ...*EA3D*. The first occurrence of E from the rear is fourth from the right, then let $S[i]$ equal all the bits in *Hash*(*Nonce*) in front of this E.

This advanced scheme is detailed in Algorithm 3.

Algorithm 3: Advanced scheme.

Data:
$C \leftarrow$ the available value in current block
$k \leftarrow$ memory needs to be consumed
$t \leftarrow$ the length of hash function
$R \leftarrow$ the requirement of hash value
Result: *Nonce*; *hash* when success

```
for Nonce ← 1 to max − nonce do
    X = str(Nonce)
    n = k/t
    Z = Hash(Nonce)
    t = |Z|
    s = 4
    cut Z into segments with length s
    each segment is denoted as S[i]
    i = [t/2*s]
    i = S[i] + S[t/s − 1]mod t/s
    X = C||Nonce||S[i]
    output begin length
    for i ← 1 to n do
        h = Hash(x)
        Y = reverse(h)
        X = Y||X
    end
    output end length
    Z = Hash(C||X)
    if Z > R then
        output
    else
        continue
    end
end
```

5. Evaluation

5.1. Time and space complexity

The time and space complexities of the three algorithms are listed in Table 2.

For Algorithm 1, assume that the number of executions of the basic statement is $f(n)$, and we have $f(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} 1 + \sum_{i=1}^{n} 1 = \sum_{i=1}^{n} n + n = n^2 + n$. Thus, the time complexity of the algorithm is, $T(n) = O(f(n) = O(n^2)$. For Algorithm 2, assume that the number of executions of the basic statement is $f(n)$, and we have $f(n) = \sum_{i=1}^{n} 3 = 3n$. Thus, the time complexity of the algorithm is, $T(n) = O(f(n) = O(n)$. For Algorithm 3, the basic statement associated with n is the same as Algorithm 2. Therefore, its time complexity is also $O(n)$.

With all three algorithms, the hash value is stored in each loop, and the length of the hash value is fixed. So the space complexity is $O(n)$.

Table 2
Time complexity and space complexity of the three algorithms.

	Basic scheme		Advanced scheme
	Algorithm 1	Algorithm 2	Algorithm 3
Time complexity	$O(n^2)$	$O(n)$	$O(n)$
Space complexity	$O(n)$	$O(n)$	$O(n)$

5.2. Experiment

In keeping with the basic scheme, we processed the *Nonce* and then determined whether the final result guaranteed the requirement. In processing the *Nonce*, the inverse number after the hash calculation was concatenated to the original number, which was used to generate a new number and the same action continued iteratively. We finally compute the hash value of combining all intermediate results to obtain the final result.

Proposition 5.1. *We can generalize the above observation as follows:*

$$M = m + i * t,$$

where $t = |Hash(\cdot)|$, *M represents current memory, m is initial memory, and i is the frequency of hash computation.*

Proof. Because each hash function consumes t units of memory and the memory is persistent, the hash function accumulates k memory after $i = k/t$ hash functions upon choosing a random number, i.e., nonce. The maximum memory consumption for searching this nonce is $m + k$. If this nonce does not guarantee the requirement (of block mining for blockchain), the memory will be released and a new number is randomly selected for the hash function and the memory accumulation process. □

For example, suppose the underlying hash function is SHA256 and the initial memory is 4 bits. The real-time memory consumed would accumulate to 1028 bits after four rounds of hashing. However, the actual memory used would differ very greatly in different operating environments, which is why we focussed on studying changes in the data lengths before and after n hashes (output in bytes) in our experiments.

Taking the memory changes during the complete process of finding the first legal nonce as an example, we can see that the change in data length under each random number is identical before finding a random number that guarantees the requirement. Therefore, the memory used with different numbers can be considered identical when k, t, and m are the same. Also, the maximum memory occupied in the complete process of finding a legal number is approximately equal to the memory under a certain random number. However, in the actual experiments, we found that m changed dynamically, and k needed to be large enough to ensure the change in m was negligible.

The settings for the experiments were as follows: the hash function was SHA256, $t = |f(\cdot)| = 256bit$, and k was changed to observe the differences in memory use when the *Nonce* is generated first.The same set of data was used as input for all three algorithms to analyze their similarities and differences. The results with a changing k are shown in Table 3, and Fig. 2 depicts the impact of k on the data length. From these results, we derived the following findings.

1. The beginning length for Algorithms 1 and 2 was $4B$ because the length of the preset *nonce* was $4B$. The beginning length of Algorithm 3 was larger than the other two algorithms because of the extra unpredictable information (recall $s[i]$) consumes $33B$.
2. The differences in data length for each of the three algorithms was the same with the same input k, which indicates that their capacity to consume memory is comparable.
3. There is a linear relationship between the input k and the difference of the begin and end length. Thus, we can restate the formulas as follows:

$$M = m + k,$$

where M is the present memory, m is the initial memory, and k is the memory that needs to be consumed.

In summary, the results illustrate that the memory consumed has a direct relationship to k, and k can be set on demand. Therefore, enforcing memory usage is feasible.

Table 3
$n = k/t$, *begin length* and *end length* are the data lengths before and after n hashes. *difference* is the gap between the two.

	Algorithm 1				Algorithm 2				Algorithm 3			
k(B)	1024	2048	3072	4096	1024	2048	3072	4096	1024	2048	3072	4096
n	32	64	96	128	32	64	96	128	32	64	96	128
begin length(B)	4	4	4	4	4	4	4	4	37	37	37	37
end length(B)	1028	2052	3076	4100	1028	2052	3076	4100	1061	2085	3109	4133
difference(B)	1024	2048	3072	4096	1024	2048	3072	4096	1024	2048	3072	4096

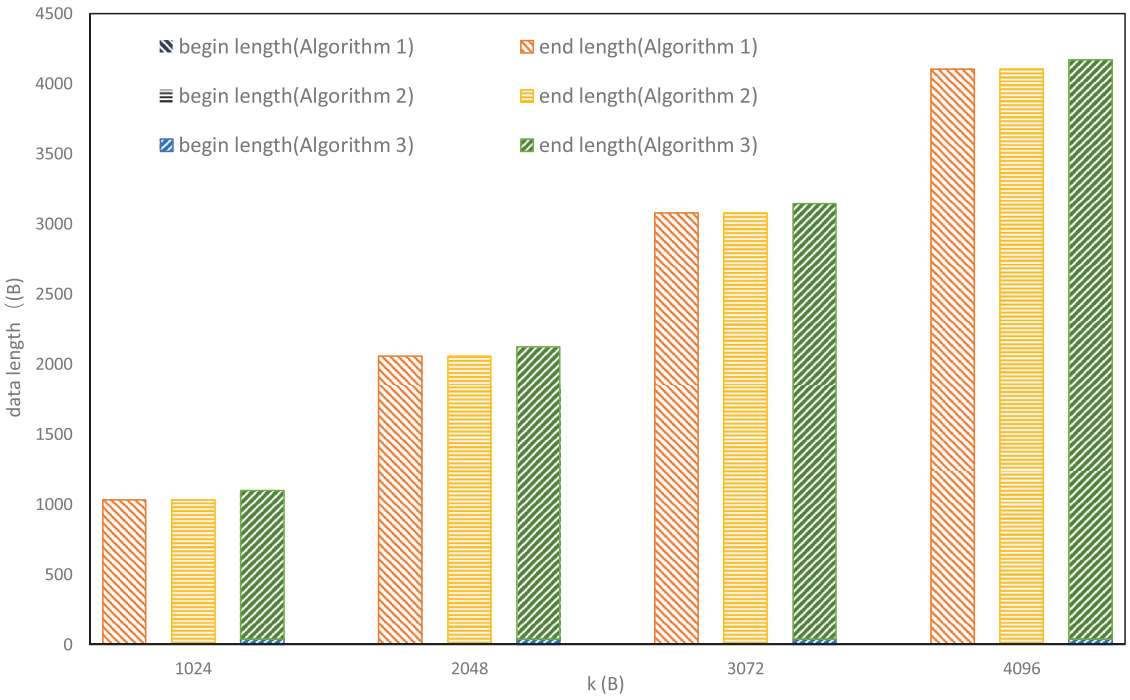

Fig. 2. Data length changes in the three algorithms.

5.3. Comprehensive analysis

Our method can be employed in realistic scenarios. The blockchain constructed with these schemes demand serialized calculations that must be computed with a set amount of memory. Miners cannot calculate hash values in parallel or in a pipeline. They can only conduct their calculations in a one-device-to-one-nonce manner, and they are obliged to consume an amount of memory in doing so. Yet, the normal operations of the blockchain remain unaffected by the number of miners participating in the PoW.

Currently, the principal applications of blockchains are financial transactions, smart contracts, and cryptocurrencies. However, as IoT develops, its security will need to be improved. Many studies have shown that blockchain could play an important role in IoT environments and become part of the IoT security solution [23]. The proposed method changes the computations of hash values in PoW to enforce memory usage. That is, there is no need to replace hash functions or previous design logics - there is no hard-fork. Consequently, our approach can not only be applied to familiar scenarios like cryptocurrencies, but also to IoT applications like smart grids and intelligent transportation systems.

6. Related work

More and more miners are employing ASICs instead of CPUs and GPUs to improve mining. M. B. Taylor [32] proposed that Bitcoin ASIC miners provide much higher performance compared to CPUs, GPUs, and FPGA (Field Programmable Gate Array)s. ASICs have raised the entry barrier to Bitcoin mining for the general public to the extent that it is almost impossible to make a profit without making an enormous investment into equipment. In this case, mining may become only possible for miners that hold the main computing power. The centralization of computing power will affect the decentralized nature of blockchain networks; giving rise to concerns and threats to Bitcoin's future [14].

There are currently two main approaches to defend against ASICs. The first is memory-hard functions, and considered to be the standard approach. C. Percival [26] first proposed this tactic and the scrypt construction to protect against attacks using ASICs. Many constructions with similar tactics have been proposed in subsequent research [8–13,21]. Notably, Zeusminer developed an ASIC miner specifically for scrypt in 2014 [5]. Another construction is to use multiple hash functions to defend against ASICs. Here, several hash functions are usually concatenated and the data is forced through different calculations sequentially. Schemes with several hash functions play on the vulnerability that ASICs can only be optimized for a few hash functions, not all required in a multi-function scheme. However, H. CHO's evaluation revealed that systems with multiple hash functions do not sufficiently defend against ASICs [19].

A detailed comparison of the two existing methods is given in Table 4.

Notably, if the hash function in the construction (e.g., SHA256) is changed to a memory-hard function, the resulting hard-fork may destabilize the cryptocurrency and undermine its value [33]. This also means that raising mining costs by

Table 4
Comparison of the two existing methods.

ASIC-resistant methods	Algorithms	Principles	Deficiencies
Memory-hard functions	Ethereum's Ethash [35], Scrypt [26], Lyra2[7], Balloon password-hashing algorithm [18], Argon2 [15], Equihash [16] and Crypt-Night [28].	Memory-hard functions require a large amount of data to be stored during the calculation, and the memory limitations of ASICs will affect its performance.	Emerging technologies, such as 3D stacking [29] have exerted an influence. If the memory requirement is fixed, miners can invest in expanding their ASICs memory for more profit.
Multiple hash functions	X11 [20], X14, X17 [6], X11EVO [2], X16S [27], X16R [17], Quark [3], and Time Travel [4].	Multiple hash functions employ a variety of functions to compute the valid value in parallel or serially to defend against ASIC miners designed for specific hash function.	It would not take too much time to develop an ASIC miner for a fixed hash family. The design cost and effort to exploit an ASIC miner for various hash functions is not extremely high.

fixing a certain property (such as memory or fixed hash series) is not a way to eliminate the ASIC miners completely. Thus, we can try to defend against ASICs by flexibly controlling the memory requirements without changing the underlying hash function.

7. Conclusion

In this paper, we proposed a PoW method that can mitigate ASIC attacks, without the need to replace the underlying hash function while minimizing change to the design logics. In other words, the proposed scheme allows us to build a *k*MIFH by invoking a hash function, which is not *k*MIF as a defense against attacks by ASICs. We also proposed an advanced scheme to thwart the pre-computing of hash results over a nonce. We recorded the resulting data length changes in the three algorithms of two schemes - one general, one flexible - both before and after *k*MIFH. The results show that the memory consumption increases to *k* as requested, where *k* is a configurable parameter on demand. In other words, computationally intensive miners in a blockchain can be thwarted by simply controlling the argument of *k* to change the memory consumption in the hash computation.

In future research, we plan to evaluate our scheme in a real-world blockchain in collaboration with a company or a cryptocurrency startup.

Declaration of Competing Interest

We declare that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Acknowledgments

This research was financially supported by the Major Scientific and Technological Special Project of Guizhou Province under Grant No. 20183001, the Open Funding of Guizhou Provincial Key Laboratory of Public Big Data under Grant Nos. 2018BDKFJJ009 and 2017BDKFJJ006, Open Funding of Hubei Provincial Key Laboratory of Intelligent Geo-Information Processing under Grant No. KLIGIP2016A05, and the National Natural Science Foundation of China under Grant no. 61502362.

References

[1] Bitcoin mining hardware guide, https://www.bitcoinmining.com/bitcoin-mining-hardware/.
[2] Proof-of-work: X11evo algorithm based on twisting X11 algos. http://revolvercoin.org/resources/.
[3] Quark vs. bitcoin. http://www.quarkcoins.com/bitcoin-vs-quarkcoin.html.
[4] Time travel (timetravel10 | bitcore) algorithm, coins, miners and hashrate. https://coinguides.org/time-travel-coins/.
[5] The first X11 mining ASIC ibelink DM384m ASIC DASH miner. https://cryptomining-blog.com/7117-the-first-x11-mining-asic-ibelink-dm384m-asic-dash-miner/.
[6] X17 Algorithm-List of all X17 Coins and miners for NVIDIA & AMD, https://coinguides.org/x17-algorithm-coins/.
[7] L.C. Almeida, E.R. Andrade, P.S.L.M. Barreto, M.A. Simplicio, Lyra: password-based key derivation with tunable memory and processing costs, J. Cryptogr. Eng. 4 (2) (2014) 75–89.
[8] J. Alwen, J. Blocki, Efficiently computing data-independent memory-hard functions, in: Proceedings of the CRYPTO, in: LNCS, 9815, Springer, Heidelberg, 2016, pp. 241–271.
[9] J. Alwen, J. Blocki, Towards practical attacks on Argon2i and balloon hashing, in: Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P), Paris, 2017, pp. 142–157.
[10] J. Alwen, J. Blocki, K. Pietrzak, Depth-robust graphs and their cumulative memory complexity, in: Proceedings of the EUROCRYPT, in: LNCS, 10212, Springer, Cham, 2017, pp. 3–32.
[11] J. Alwen, B. Chen, K. Pietrzak, L. Reyzin, S. Tessaro, Scrypt is maximally memory-hard, in: Proceedings of the EUROCRYPT 2017, in: LNCS, 10212, Springer, Cham, 2017, pp. 33–62.
[12] J. Alwen, P. Gazi, C. Kamath, K. Klein, G. Osang, On the memory-hardness of data-independent passwordhashing functions, 2016. Cryptology ePrint Archive. Report 2016/783, https://eprint.iacr.org/2016/783.pdf.
[13] J. Alwen, V. Serbinenko, High parallel complexity graphs and memory-hard functions, in: Proceedings of the Forty Seventh Annual ACM on Symposium on Theory of Computing, ACM, 2015, pp. 595–603.

[14] A. Beikverdi, J.S. Song, Trend of centralization in bitcoin's distributed network, in: Proceedings of the IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015, pp. 1–6. Takamatsu.
[15] A. Biryukov, D. Dinu, D. Khovratovich, Argon2: new generation of memory-hard functions for password hashing and other applications, in: Proceedings of the IEEE European Symposium on Security and Privacy (EuroS & P), 2016, pp. 292–302. Saarbrucken.
[16] A. Biryukov, D. Khovratovich, Equihash:asymmetric proof-of-work based on the generalized birthday problem, in: Proceedings of the Network and Distributed System Security Symposium, 16, 2016.
[17] T. Black, J. Weight, X16R-ASIC resistant by design, Technical report. https://ravencoin.org/wp-content/uploads/2018/03/X16R-Whitepaper.pdf.
[18] D. Boneh, H. Corrigan-Gibbs, S. Schechter, Balloon hashing: a memory-hard function providing provable protection against sequential attacks, in: Proceedings of the Advances in Cryptology-ASIACRYPT, 2016, pp. 220–248.
[19] H. Cho, ASIC-resistance of multi-hash proof-of-work mechanisms for blockchain consensus protocols, in: Proceedings of the IEEE Access, 6, 2018, pp. 66210–66222.
[20] E. Duffield, D. Diaz, Dash: a payments-focused cryptocurrency, 2018. https://github.com/dashpay/dash/wiki/Whitepaper.
[21] C. Forler, S. Lucks, J. Wenzel, Catena : a memory-consuming passwordscrambling framework, 2013. Cryptology ePrint Archive. Report 2013/525, https://eprint.iacr.org/2013/525.pdf
[22] V. Gramoli, From blockchain consensus back to byzantine consensus, Future Generation Computer System, 2017, doi:10.1016/j.future.2017.09.023.
[23] D. Minoli, B. Occhiogrosso, Blockchain mechanisms for IoT security, Internet of Things, 1–2, 2018, pp. 1–13.
[24] S. Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. https://bitcoin.org/bitcoin.pdf.
[25] C. Percival, S. Josefsson, The scrypt password-based key derivation function, IETFRFC7914.
[26] C. Percival, Stronger key derivation via sequential memory-hard functionsy, 2009. http://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf.
[27] L. Pighetti, X16s-sixteen shufed algorithms designed for small miners, Technical report. https://github.com/Pigeoncoin/brand/blob/master/X16S-whitepaper.pdf.
[28] N.V. Saberhagen, Cryptonote v.2.0, Technical report, 2013. https://cryptonote.org/whitepaper.pdf.
[29] M.M.S. Aly, et al., Energy efficient abundant-data computing: the N3XT 1,000x, Computer 48 (12) (2015) 24–33.
[30] M. Swan, Blockchain: Blueprint for a New Economy, USA:O'Reilly Media, Newton, MA, 2015.
[31] M.B. Taylor, Bitcoin and the age of bespoke silicon, in: Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), Montreal, QC, 2013, pp. 1–10.
[32] M.B. Taylor, The evolution of bitcoin hardware, Computer 50 (9) (2017) 58–66.
[33] B.D. Trump, E. Wells, J. Trump, Cryptocurrency: governance for what was meant to be ungovernable, Environ. Syst. Decis. 38 (3) (2018) 426–430.
[34] F. Tschorsch, B. Scheuermann, Bitcoin and beyond: a technical survey on decentralized digital currencies, IEEE Commun. Surv. Tutor. 18 (3) (2016) 2084–2123. Thirdquarter.
[35] G. Wood, Ethereum: a secure decentralised generalised transaction ledger, in: Proceedings of the Ethereum, 2014. http://yellowpaper.io/.
[36] A.R. Zamanov, V.A. Erokhin, P.S. Fedotov, ASIC-resistant hash functions, in: Proceedings of the IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), 2018, pp. 394–396.