Internship report

Security of memory-hard hash functions

Amaury Bouchra Pilet

École Normale Supérieure « Ulm »
Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA)

Ventôse - Thermidor 225

Contents

I Summary

By Amaury Bouchra Pilet, under the supervision of Pierre-Alain Fouque, affiliated to the EMSEC team at IRISA.

I.1 The general context

Password-based authentication is widely used in modern information systems. To prevent an attacker with read-only access to the server from being able to get users passwords, most services do not store passwords themselves but a hash. A hash of a password is obtained using some function which is difficult to invert.
Effective attacks on hash functions usually imply lots of evaluations of the considered hash function, except if this function has some known mathematical weakness, and thus, should not be used. When performing such attacks, the attacker will try to minimize the cost of evaluation the hash function. For classical hash function, the main difficulty in their evaluation is the calculation itself, thus, attackers often reduces the cost using hardware implementation of hash functions.
To prevent this kind of optimization, *memory-hard* hash functions where developed. The idea of this kind of function is that the cost of evaluating the function is not based on the calculations themselves but on the amount of data that need to be kept in memory during the calculation. This concept has been introduced in 2009 by Colin Percival with his scrypt key derivation function [Per09]. Since it is not possible to reduce the cost memory by developing dedicated hardware, attackers can not reduce the cost of evaluating this kind of function with dedicated hardware.

I.2 The research problem

While it is difficult, but not impossible, to optimize the cost of evaluating a memory-hard hash function using dedicated hardware, it is possible to optimize the way it is evaluated to reduce the amount of memory required to evaluate it. Particularly, it is possible to forget some values and recalculate them later, which is called *time-space trade-off*. To evaluate the quality of a time-space trade-off, we use a cost called *energy cost* which allows use to sum the cost of the calculation itself and the cost of keeping information in memory.
In [AB16a], Joël Alwen and Jeremiah Blocki proposed a general way of attacking memory-hard hash function and later developed it in [AB16b] in the particular case of two functions: Argon2i and Balloon Hash, but these attacks have never been actually implemented. We are in particular interested in Argon2i [BDK16], winner of the PHC[1], to be proposed for normalization by IETF [BDKJ17] and included or soon to be included in various commonly used software[2].

[1] password-hashing.net
[2] see Update on Argon2, slide 6

I.3 Your contribution

In this paper, we will recall the principles of Alwen & Blocki's attack against Argon2i. We will show how to, optimally, extend this attack to the multi-threaded version of Argon2i. We will present our pre-calculation-based effective version of the attack and several associated optimizations. Then, we will exhibit our effective implementation of the never implemented [AB16b] attack and present empirical results on its effectiveness. Based on these results, we will make statements about this attack's effectiveness, Argon2i's security (ans secure parameter range) and explain why we think that the statement in [ABH17] about multi-threaded Argon2i's presumed insecurity maybe excessive. We will also explain how to strengthen Argon2i's effective resistance to our attack, and, in general, any memory-hard hash function's resistance to pre-calculation-based attacks.

I.4 Arguments supporting its validity

Unlike previous results on Argon2i cracking, supported by theoretical proofs [AB16a] or simulation -based cost estimation [AB16b], our results are supported by effective cost calculation. We assume that energy cost is a good way to estimate the effective cost of an attack, like Alwen & Blocki, and also had to assume a value for the effective ratio between the cost memory and the cost of computation, the same value used by Alwen & Blocki and estimated by Argon2i' authors. Given these assumptions, we calculated the energy cost of an evaluation of Argon2i using our attack and the energy cost of the same evaluation using the standard Argon2i algorithm. We ensure that our attack allows to correctly compute an Argon2i hash by comparing its output with the one given by our implementation of Argon2i using its standard algorithm.

I.5 Summary and future work

We improved Alwen & Blocki's attack against Argon2i, a recent memory-hard hash function proposed to normalization and beginning to receive official support in commonly used network services software. We developed a working implementation of this attack and evaluated its effectiveness.
Future work may include optimizing the calculation time of the pre-calculation algorithm, which we did not tried to do, improving the attacks quality when attacking parallel Argon2i, or providing a theoretical explanation of the better than expected resistance of parallel Argon2i to our attack.
In this work, we also explain how to drastically increase resistance of memory-hard hash function to pre-calculation-based attacks; developing new (versions of) hash functions implementing our idea is probably the most important work that should be done, based on our conclusions.

II Context

II.1 Argon2i

The following is a general description of the Argon2i hash function, for a more detailed description, read the reference Argon2 v1.3 paper [BDK16] or the associated RFC draft [BDKJ17].

II.1.a Internal Hashing function

Argon2i makes use of an internal hash function, this function is BLAKE2b [ANWOW13], an improved version of the SHA-3 finalist[3] BLAKE, proposed for normalization by IETF [SA15]. BLAKE2b is used in Argon2i in two ways. It is used as-is to get a fixed size (1 KiB) initial hash, $H0$, get initial *blocks* (see later) and obtain the output hash from final blocks. Argon2i also uses a modified version of BLAKE2b's *compression function*, G, for its non-initial blocks computations, including *addresses blocks* (see later); this function is essentially a $2 \times 1KiB \rightarrow 1KiB$ hash function.

II.1.b Main principle

Argon2i is based on the computation of several *blocks*. A block is 1Kio of data, initial blocks are computed from input using the internal hash function, other blocks are computed from previously computed blocks. The final output is computed from final blocks using the internal hash function.

II.1.c Blocks computation

The two first blocks of a lane (see later) are computed from the initial hash $H0$, all other blocks are computed using the G function from the immediately preceding block and a random already computed block.

II.1.d Parallelism

A single Argon2i hash can be computed using several processes in parallel. To allow this, blocks are given a *lane* number, with the same number of blocks in each lane and a number of lanes equal to the wanted degree of parallelism. The memory space is also divided in 4 *slices*, computed sequentially. After every block of a slice have been computed, all processes are synchronized and the next slice's computation begins. To ensure that, when computing a block, we never need a value that is still to be computed by an other process, block's random parents are chosen the following way: if we are in the first slice, the block is selected randomly in already computed blocks of the current lane, else, the slice of the random block is selected randomly then, if both blocks are on the same slice, the random block is selected in already computed block, else, it is selected in block belonging to already computed slices.

II.1.e Multiple passes

Argon2i allows to make multiple computation passes. In this case, initial block are re-computed from already computed blocks. In original Agon2i, re-computed blocks just overwrote previously computed ones but, to improve Argon2i's resistance to certain attacks, notably Alwen & Blocki's one, Argon2i v1.3 XOR new blocks with old ones. Multiple passes effectively increases Argon2i's resistance to Alwen & Blocki's attack but, by reducing the memory space influence in its computation cost, it also decreases its resistance to classical, custom hardware-based attacks.

[3] SHA-3 finalists list (NIST)

II.1.f Random parents generation

To choose randoms parents, Argon2i uses a method based on its G function. In Argon2i, the seed for this calculation only depends on parameters and not on data (including salt). This design is supposed to prevent side-channel attacks but also make attacks like ours possible. Argon2 also exists in a version in which the random parents depends on data. Since the calculation flow is dependent from the data, this version of Argon2, while being resistant to our attack, is sensitive to memory access-based side-channel attacks. There is also an intermediate version, Argon2id which mixes the two preceding versions trying to get the best of both, but necessarily also gets weaknesses from both.

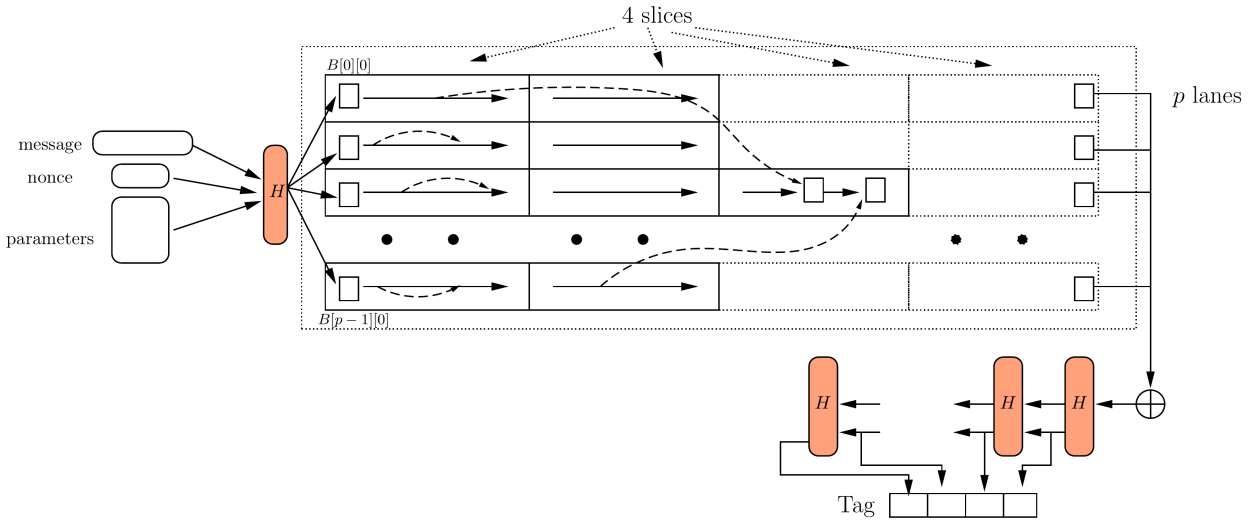

Figure 1: Argon2's calculation flow (single pass). Credits: [BDK16]

II.2 Alwen & Blocki's original attack

In the following we will explain the woring principles of Alwen & Blocki's original attack, on which our attack is based. For a more detail technical description of this attack see [AB16b], for theoretical foundations of this attack see [AB16a].

II.2.a Energy Complexity

To estimate the effective cost of evaluating a memory-hard hash function, especially when performing an attack which can reduce the amount of memory used but increase the number of calculi to be done, we use the notion of *Energy Complexity*. Energy Complexity allows us to sum memory and calculation complexities, it is defined this way: $Ec(F) = Mc(F) + R \cdot Cc(F)$ where Cc is the calculation complexity, here, the number of calls to Argon2i's G function, Mc is the memory complexity, here, the some over all calls G of the number of blocks in memory during G's calculation and R the ratio of energetic cost between computation and memorization. In our case, had $R = 3000$, which is the values used by Alwen & Blocki in there work and given by Argon2i's authors in [BK15].

II.2.b Computation graph

Alwen & Blocki's attack sees Argon2i's, and similar functions', computation flow as a Directed Acyclic Graph (*DAG*) G. Nodes are blocks and a node's parents are the blocks needed for its computation (2 or 3 for Argon2i).

II.2.c Depth reducing set & sandwich graphs

The *depth* of a DAG is the length of the longest (directed) path in the graph. For an Argon2i graph, this is the number of blocks divided by the parallelism degree.
To achieve the attack, we need to reduce this depth; for this purpose, we introduce the notion of *sandwich graph*. A sandwich graph is a graph witch can be divided into l *layers* in which all paths have length $\leqslant d_l$ for some d_l and l such that $ld_l \leqslant d$ where d is the wanted depth for our graph.
Argon2i graph are not sandwich graphs but can be reduced to sandwich graph by removing a (small) set of nodes S. d_l and l known (we will explain later how these values are chosen), and all n nodes begin indexed by integers (naturally for mono-lane Argon2i) we can divide the graph into l successive and equally sized layers, themselves divided into successive *segments* of size d_l. This being done, we define S as the set of all node which are either the end of a segment or have a parent in an other segment of the same layer which is closer to the end of its segment than the child node is.
With S defined this way, $G - S$ (G whiteout all node of S) is a sandwich graph and has depth $\leqslant d$.

II.2.d Pebbling game

Alwen & Blocki's attack evaluation of Argon2i consists in solving a pebbling game. The principle of this game is that you can place a pebble on a node if all its parents are already pebbled. A the beginning of the game, you place a pebble on the initial node(s), a the end, you pebble the final node(s). Pebble may also be removed during the game. When we place a pebble on a node, we compute the associated block, when we remove a pebble, we free the allocated memory.
This abstract approach to get time/space trade-off evaluations was originally introduced in [PH70] and [Coo74]. It has been more recently used in cryptography in [DNW05] and later adapted to parallel evaluation in [AS15]. The attack's way of playing the game is based on two alternating kinds of phases: *balloon* phases, during which lots of nodes are pebbled, and *light* phases, during which only a minimal number of nodes is kept pebbled. The following algorithm describes the process precisely:

Algorithm 1: GenPeb (G, S, g, d)

Arguments : $G = (V, E)$, $S \subseteq V$, $g \in [\mathsf{depth}(G - S), |V|]$, $d \geq \mathsf{depth}(G - S)$
Local Variables: $n = |V|$

for $i = 1$ *to* n **do**
  Pebble node i.
  $l \leftarrow \lfloor i/g \rfloor * g + d + 1$
  if $i \mod g \in [d]$ **then** // Balloon Phase
    $d' \leftarrow d - (i \mod g) + 1$
    $N \leftarrow \mathsf{need}(l, l + g, d')$
    Pebble every $v \in N$ which has all parents pebbled.
    Remove pebble from any $v \notin K$ where $K \leftarrow S \cup \mathsf{keep}(i, i + g) \cup \{n\}$.
  else // Light Phase
    $K \leftarrow S \cup \mathsf{parents}(i, i + g) \cup \{n\}$
    Remove pebbles from all $v \notin K$.
  end
end

Algorithm 2: Function: $\mathsf{need}(x, y, d')$

Arguments: x, $y \geq x$, $d' \geq 0$
Constants : Pebbling round i, g, gap.
$j \leftarrow (i \mod g)$ // Current Layer is $L_{\lfloor j/gap \rfloor}$
Return $L_{\lfloor j/gap \rfloor} \cap \left\{ i \cdot gap + j \;\middle|\; i \leq \frac{n}{gap} \right\}$

Algorithm 3: Function: $\mathsf{keep}(x, y)$

Arguments: x, $y \geq x$
Constants : Pebbling round i, g, gap, $\#layers$, n, σ.
$j \leftarrow (i \mod g)$
$\ell \leftarrow \lfloor (j/gap \rfloor$ // Current Layer
Return $L_{\geq \ell - \lceil \frac{\sigma \# layers}{n} \rceil}$

Figure 2: Alwen & Blocki's algorithm. Credits: [AB16b]

III Our attack

Our attack is essentially Alwen & Blocki's one with addition to work with optimal performance on the multi-lane version of Argon2i, several post-optimizations to improve performance and a system to allow parallel computation when attacking the multi-lane version of Argon2i.
We decided to make our attack work in two phases: first, a pre-calculation, where all non-trivial non-data-dependent calculations are made, which gives use an ordered list of instructions, either computes of free a block, then, we proceed to the evaluation of the hash functions, following the instructions we got from pre-calculation. We observed that pre-calculation takes much more time than evaluation of the function. If we did not used pre-calculation, our attack would not be effective at all.
We present two versions of our attack: the first is optimized for monothreaded evaluation and may be adapted to multithreaded evaluation with a post-treatment algorithm we provide; the second is optimized for multithreaded evaluation, like Alwen & Blocki's original attack, and includes threads synchronization barriers management.
In Appendix A, a generalized version of our attack's pebbling game solving algorithm that can output both, non-parallel and parallel instructions. Warning, 3 pages long!

III.1 Adaptation to multi-lane Argon2i

III.1.a Graph linearization

When performing the attack, nodes must be indexed by integers; we call the process of indexing them *linearization (of the graph)*.
This process is done naturally in mono-lane Argon2i; for multi-lane Argon2i we proceed the following way. First, we chose the number of layers and compute the limits of layer for one lane. Then, the nodes are indexed the following way: first we order all nodes by layer, then, in each layer, we order nodes by lane, finally, we order nodes in each (layer,lane) pair the natural way.
This, combined with the good choice of values for layers and segments explained in the next section, allows us to get a smaller set S, like explained next.

III.1.b Values for layers and segments

We chose for the number of layer a multiple of $4 \times t$, 4 being the number of Argon2i sync points and t the number of passes, and for the number of segments (in a layer) a multiple of the number of lanes. This way, sync barriers are always layers limits and segments are always on one lane only. This tends to reduce the size of the set S because, in multi-lane Argon2i, the random parent of a block is always in a previous layer, so this way, we reduce the number of node which have parents in the same layer but positioned later in their segments, nodes that we have to add to S.

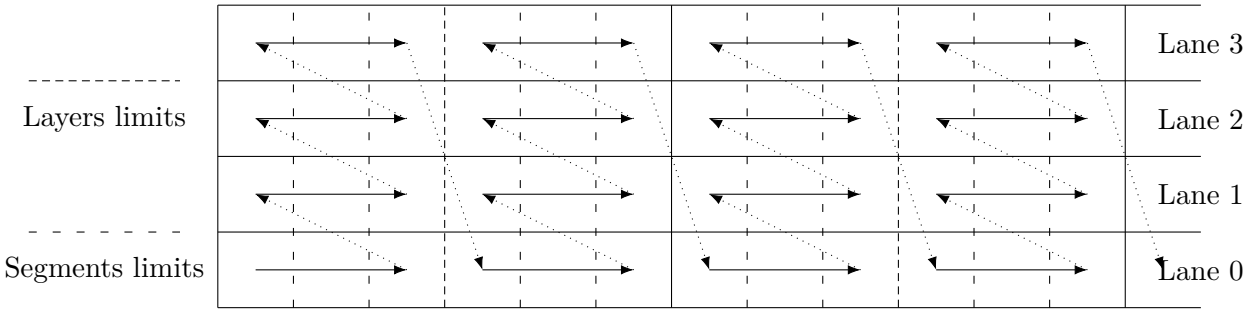

Figure 3: Illustration of our linearization and layers & segments system

III.2 Post-optimizations

Dividing our attack in two parts allows use to perform some additional optimizations at the end of the pre-calculation. These were designed for the non-parallel version of our attack, then we adapted them to the parallel attack. For the parallel version, we did not made our optimization work across sync barriers, this being the easiest way to keep the instructions consistent. All these optimization are designed to always decrease energy cost, some can however lower parallelisability of the non-parallel attack and thus, are not applied before parallelizing. The order in which post-optimization are applied is relevant and the one we use is the results of experimentation.
Following the descriptions of post-optimizations, we will provide associated algorithms for non-parallel attack, see Appendix B for the parallel versions.

III.2.a Ensure Free

Since the pebbling algorithm do not produces free instructions for all block, we ensure that no block, except final one, are still in memory after the end of the computation, we add necessary free instructions after the last calculate instruction for which the block is necessary.
For parallel attack, freeing all blocks is ensured at the end of the initial calculation, freeing everything as soon as possible is left to the Free Earlier optimization.

III.2.b Clean Useless

There is no guaranty that the pebbling algorithm will not produce (calculate n,free n) pairs without any calculate instruction where n is needed between. We remove such pairs in this process. We check nodes in decreasing order because removing a (calculate n,free n) pair for a node may make a similar pair for one of its parents useless, allowing cascading removals.

III.2.c Free Earlier

When a free instruction is produced by the pebbling algorithm, it is not, in general, placed after the last use of the block. In this process, we ensure that all blocks are freed as soon as possible.
This optimization, as well as the Ensure Free optimization, relies on the same principle as the attack against the first version of Argon2i mentioned in [BCGS16]. It was originally made for the first version of Argon2i and received a counter-measure in v1.3 but combined with Alwen & Blocki's attack, it can still give slight improvement.

III.2.d Calculate Later

The symmetric of the previous post-optimization for calculate instructions. Here, we scan nodes in decreasing order for the same reason as in Clean Useless. Because of counter-performances, this optimization is not used before multithreading the non-parallel attack.

III.2.e Reduce Memory

The Reduce Memory optimization is the direct application of the time-space trade-off principle. During this process, we ensure that no block stays in memory without being use for a time τ enough long to make recomputing the block cheaper than keeping it in memory: $\tau > R$. Which order on nodes should be used for this optimization is not clear, we choose decreasing order rather than increasing after some tests but other orders could be more effective.
Because of counter-performances, this optimization is not used before multithreading the non-parallel attack. Since it's difficult to give a valid value for used memory in the parallel case, this optimization has not been adapted to the parallel case.

Algorithm Ensure Free

```
1: function ensureFree(pr, N)
2:     for all n ∈ [[0; N[[ do
3:         if isFinal(n) then
4:             ins ← pr
5:             toFree ← ⊥
6:             while ins do
7:                 if node(ins)= n then
8:                     if isCalc(ins) then
9:                         toFree ← ⊤
10:                        insrtPoint ← ins
11:                    else
12:                        toFree ← ⊥
13:                    end if
14:                else if toFree∧isParent(ins, n) then
15:                    insrtPoint ← ins
16:                end if
17:                ins ←next(ins)
18:            end while
19:            if toFree then
20:                addFree(insrtPoint, n)
21:            end if
22:        end if
23:    end for
24: end function
```

Algorithm Clean Useless

```
1: function cleanUseless(pr, N)
2:     for n ← N − 1 to 0 do
3:         ins ← pr
4:         rm ← ⊥
5:         while ins do
6:             if node(ins)= n then
7:                 if isCalc(ins) then
8:                     clc ← ins
9:                     rm ← ⊤
10:                else
11:                    if rm then
12:                        rmInst(clc)
13:                        rmInst(ins)
14:                    end if
15:                end if
16:            else if rm∧isParent(ins, n) then
17:                rm ← ⊥
18:            end if
19:            ins ←next(ins)
20:        end while
21:    end for
22: end function
```

Algorithm Free Earlier

```
1: function freeEarlier(pr, N)
2:     for all n ∈ [[0; N[[ do
3:         ins ← pr
4:         while ins do
5:             if isCalc(ins) then
6:                 if isParent(ins, n) then
7:                     lst ← ins
8:                 end if
9:             else if node(ins)= n ∧ ¬isParent(previous(ins), n) then
10:                addFree(lst, n)
11:                rmInst(ins)
12:            end if
13:            ins ←next(ins)
14:        end while
15:    end for
16: end function
```

Algorithm Calculate Later

```
1: function calcLater(pr, N)
2:     for n ← N − 1 to 0 do
3:         ins ←next(pr)
4:         w ← ⊥
5:         while ins do
6:             if isCalc(ins)∧node(ins)= n then
7:                 clc ← ins
8:                 w ← ⊤
9:             else if w ∧ (isParent(ins, n)∨(isFree(ins)∧isParent(clc,node(ins)))) then
10:                w ← ⊥
11:                if node(previous(ins))≠ n then
12:                    cpInst(clc,previous(ins))
13:                    rmInst(clc)
14:                end if
15:            end if
16:            ins ←next(ins)
17:        end while
18:    end for
19: end function
```

Algorithm Reduce Memory

```
1: function reduceMem(pr, N, R)
2:     for n ← N − 1 to 0 do
3:         ins ← pr
4:         w ← ⊥
5:         ws ← 0
6:         mem ← 0
7:         while ins do
8:             if isParent(ins, n) then
9:                 if w then
10:                    if parentsPebbled(peb, n) then
11:                        if R + mem < ws − 1 then
12:                            addFree(beg, n)
13:                            addCalc(previous(ins), n)
14:                        end if
15:                    end if
16:                    beg ← ins
17:                    ws ← 0
18:                else
19:                    beg ← ins
20:                    w ← ⊤
21:                    ws ← 0
22:                end if
23:            else if node(ins)= n∧isFree(ins) then
24:                w ← ⊥
25:            end if
26:            updatePebbling(peb, ins)
27:            if isCalc(ins) then
28:                mem + +
29:                ws + +
30:            else
31:                mem − −
32:            end if
33:            ins ←next(ins)
34:        end while
35:    end for
36: end function
```

III.3 Parallel computation

For the non-parallel version of the algorithm, an additional post-treatment is required to allow mutlithreaded computation. This post treatment is required to make this version of the attack competitive in the case of multi-lane Argon2i, whom standard evaluation is multithreaded. We tried to use a parallelization algorithm that relies on the natural parallelisability of the attack, but tests showed that a more general greedy algorithm was more efficient. We also observed that we got better results when parallelizing before Calculate Later and Reduce Memory post-optimizations were performed.
Our parallelization algorithm is based on the following principle. It takes as parameters the minimum number of threads we wants to maintain active permanently, the maximum number of parallel threads usable and the tolerated difference between the number of instructions executed by two different threads for considering they are both active for the same time. Given these parameters, it assigns calculate instructions to threads greedily, choosing the less loaded thread on which the instruction can be executed regarding the values already calculated in this thread. If not thread has all required values or if the assignation would break the load equilibrium condition then all threads are synchronized and we reinitialize the process. Free instructions are not assigned but spreed over all threads and, when evaluating, the last thread hitting the free instruction for a block will free it. To prevent data-races we don't allow recalculation of blocks freed if all threads have not been synchronized since freeing.

Algorithm Parallelize

```
1: function PARALLELIZE(ins, N, gap, minP, maxP)
2:     lst ← maxP
3:     td ← ⊤
4:     for all i ∈ ⟦0; maxP⟦ do
5:         order[i] ← i
6:     end for
7:     while ins do
8:         if ISCALC(ins) then
9:             if ¬(unSyncedFree[NODE(ins)]∨(td∧¬ISBALLOON(ins)∧ISFIRSTINLAYER(NODE(ins))
   )) then
10:                 i ← 0
11:                 while i < lst do
12:                     j ← order[i]
13:                     if PARENTSPEBBLED(peb[j],NODE(ins)) then
14:                         CPINST(int, ans[j])
15:                         UPDATEPEBBLING(peb[j], ins)
16:                         len[j] + +
17:                         SORTBYLENGTH(order, len)
18:                         maxLen ← len[order[maxP − minP]] + gap
19:                         lst ←LASTSHORTERINORDER(order, len, maxLen)
20:                         td ← ⊤
21:                         goto next
22:                     end if
23:                     i + +
24:                 end while
25:             end if
26:             SYNCPEBBLES(peb)
27:             lst ← maxP
28:             for all i ∈ ⟦0; maxP⟦ do
29:                 len[i] ← 0
30:                 order[i] ← i
31:                 ADDBARRIER(ans[i])
32:             end for
```

Algorithm Parallelize (part 2)

```
33:             RESET(unSyncedFree)
34:             td ← ⊥
35:         else
36:             for all i ∈ ⟦0; maxP⟦ do
37:                 CPINST(ins, ans[i])
38:                 UPDATEPEBBLING(peb[i], ins)
39:             end for
40:             unSyncedFree[NODE(ins)]← ⊤
41:         end if
42:         : next
43:         if td then
44:             ins ←NEXT(ins)
45:         end if
46:     end while
47:     for all i ∈ ⟦0; maxP⟦ do
48:         REWIND(ans[i])
49:     end for
50:     return ans
51: end function
```

III.4 Parameters choice

During our experiments, we observed that Alwen & Blocki's way of choosing parameters was not optimal. They first initialized g to the theoretical optimum, then computed optimal layers & segments for that g, using grid search, then computed an optimal g the same way. We observed that there is a strong inter-dependence between g and layers & segments values and Alwen & Blocki's optimization algorithm easily leads to non-optimal choices of parameters. We first tried to do the same algorithm twice, using the g value computed in first pass for the second pass rather than the theoretical optimum, but this was not sufficient.

We decided to compute optimal values for all three parameters at the same time. This is significantly slower, for the same granularity level, cubic rather that quadratic in the number of data points, but it proved to return better values. In most case, even with a lower number of data points, we obtained equivalent results, but, in several cases, we observed that the original method actually got "stuck" on sub-optimal values and we got better results with our method.

To get an estimated valued of the cost of our attack, we modified our instructions generator to return an estimated value of the attack's cost. While being very precise, this estimator can not take into account the effect of post-optimizations. In addition, for the parallel version of our attack, it assumes that we get perfect parallelism which was definitely not the case, thus it significantly under-estimates the real cost of the attack.

For our attack, we use 3-dimensional grid search on three parameters: the number of layers, the number of segments and g. We use 10 data-points for each variable, for a total of 1000 data-points. Following, our attack whole pre-calculation process for both, non-parallel and parallel version.

Algorithm Attack pre-calculation

function attackPrep(p, m, t, R)
  $n \leftarrow m \cdot t$
  $G \leftarrow$ genGraph(p, m, t)
  $(l, s, g) \leftarrow$ searchForParameters($G, n, t, p, R, \bot$)
  $\bar{l} \leftarrow$ getLayers($l, \frac{n}{p}$)
  $H \leftarrow$ linGraph($G, \bar{l}, p, m, n$)
  $\bar{s} \leftarrow$ getSegments($l, s, \bar{l}$)
  setLayers($\bar{l}, n, H$)
  setSegments($\bar{s}, n, H$)
  $S \leftarrow$ selectS(H, n)
  $S \leftarrow$ expandS($S, n, p, l, \bar{l}$)
  $ans.insts \leftarrow$ getInsts($H, S, n, t, l, s, \bar{l}, \bar{s}, g$)
  $Z \leftarrow$ expandS($\varnothing, n, p, l, \bar{l}$)
  ensureFree($ans.insts, n, Z$)
  cleanUseless($ans.insts, n$)
  freeEarlier($ans.insts, n$)
  $ans.instsp \leftarrow$ parallelize($ans.insts, H, n, \frac{n}{l \cdot p}, p, p + s$)
  calcLater($ans.insts, n$)
  reduceMem($ans.insts, n, H, R$)
  return ans
end function
function attackPrepParallel(p, m, t, R)
  $n \leftarrow m \cdot t$
  $G \leftarrow$ genGraph(p, m, t)
  $(l, s, g) \leftarrow$ searchForParameters($G, n, t, p, R, \top$)
  $\bar{l} \leftarrow$ getLayers($l, \frac{n}{p}$)
  $H \leftarrow$ linGraph($G, \bar{l}, p, m, n$)
  $\bar{s} \leftarrow$ getSegments($l, s, \bar{l}$)
  setLayers($\bar{l}, n, H$)
  setSegments($\bar{s}, n, H$)
  $S \leftarrow$ selectS(H, n)
  $S \leftarrow$ expandS($S, n, p, l, \bar{l}$)
  $Z \leftarrow$ expandS($\varnothing, n, p, l, \bar{l}$)
  $ans.insts \leftarrow$ getInstsParallel($H, S, n, t, l, s, \bar{l}, \bar{s}, g, Z$)
  cleanUseless($ans.insts, n$)
  freeEarlier($ans.insts, n$)
  calcLater($ans.insts, n$)
  return ans
end function

IV Experiments

IV.1 Implementation

We implemented our attack in C. While C is not the most easy to use programming language and some algorithm's may have been easier to implement in another language, C is remarkably fast, probably the reason why BLAKE2 and Argon2's reference implantation are also using this language. We followed the C11 norm. For parallelism, we used POSIX threads. C11 provides threads but this is an optional part of the norm which has not received much interest from C standard libraries developers. At the time we are working, is not part of the GNU C Library. We did not used OpenMP because we wanted to work at the lower possible level (without working with assembly) to get the performance possible.
While we did not focused on improving our pre-calculation algorithm's speed, our evaluation algorithms, for standard Argon2i, mono-threaded attack and multi-threaded attack, are manually optimized. The whole program received GCC optimization level 3.
We checked that our implementation does not cause memory corruption using Valgrind for values for which the induced slowdown allowed it.
You can find our code here.

IV.2 Results

This being an applied science work, our results are experimental. We provide exact values of attack qualities of different versions of our attack depending on all parameters of Argon2i as well as pre-calculation times.

IV.2.a Methodology

Attack quality was measured from instructions outputted by the pre-calculation, these instructions were also used to effectively compute an Argon2i hash and the resulting hash was compared with our reference Argon2i implementation's output to ensure that these instructions allowed effective computation of the Argon2i hash function.
For values inducing more than 2^{17} node, we used an estimator to speed-up the computation. The estimator ignores the post-optimizations and outputs directly the estimated value of the attack's cost used by our parameter choice algorithm. We also reduced the number of data-points used by the parameters choice algorithm: 5 data-points per variable, for a total of 125, a 8× speedup.
It is important to note that the Energy Complexity we use is memory space×time cost but a synthesis of the memory space×time and the used processing power. Note also that the reference cost we used is not the one of an implementation using the target memory space all time but of an implementation using the trivial optimization of only allocating memory when the value to be stored in is computed. This is the same metric used in [AB16b] but other works may used different metrics for attack quality, notably metrics giving higher values for the same attack, please be careful when comparing results.

IV.2.b Testing system

- Hardware configuration:
  - Processor: Advanced Micro Device Ryzen 7 1800X, 8 cores 3.6GHz, 96KB L1 cache per core + 512KB L2 cache per core + 16MB L3 cache, SMT activated 16 threads 4Ghz Turbo, TDP 95W
  - Memory: G.Skill Ripjaws V Black, 2×16GB DDR4 Dual Channel non-ECC, 2800MHz-PC22400 14-14-14-35
  - Motherboard: ASUSTek ROG CROSSHAIR VI HERO
- Software configuration:
  - Operating System: Gentoo GNU/Linux ~amd64
  - Kernel: 4.12 Linux with Gentoo patchset, custom config
  - Compiler: GNU Compiler Collection 7.1.0, -O3, GNU C Library 2.24

IV.2.c Plots

In the following $\|$ means parallel and $\nparallel$ non-parallel. Raw means that the attack did not received post-optimizations, Optimized mean it received them. Multithreaded means non-parallel attacks adapted to multithreaded evaluation using post-treatment, after optimizations. When the attribute is omitted, it is the optimized version, when the type of attack is omitted, it is the non-parallel one.

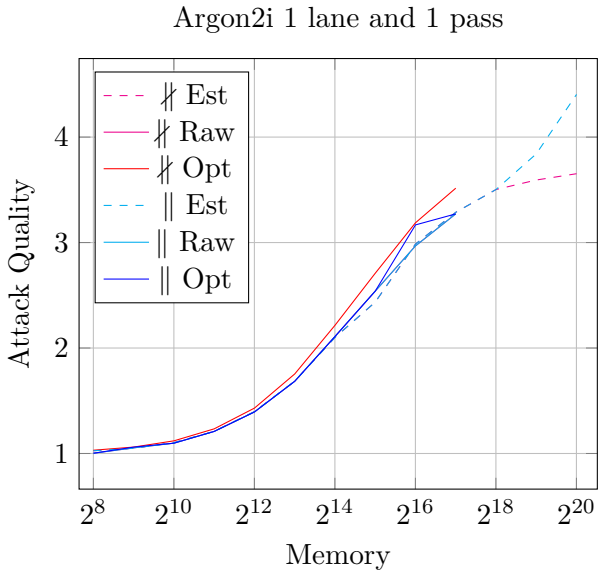

Figure 4: Attack quality depending on memory space for mono-lane mono-pass Argon2i

We observe here that our attack is already effective for 256KiB of memory. For 128MiB, we get more than a 3× gain.
All versions of our attack are close in quality, the non-parallel optimized being the best. We see that our estimators are very close to the value they approximates. The non-parallel raw plot is no visible, hidden by the parallel raw plot which is quasi the same.
The estimators drop under the actual values at 2^{15}, our program's detail output allows us to see that the parameters choice of the estimator was different from the effective attack's one, certainly due to the estimator's lower number of data-points.
For great values, the estimated quality of our non-parallel attack seems to hit a limit but see that the parallel version of our attack do not hit that limit.

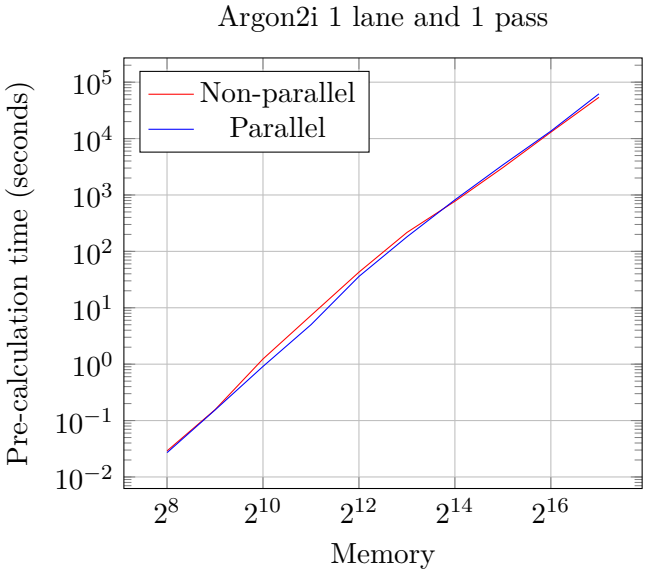

Figure 5: Pre-calculation time depending on memory space for mono-lane mono-pass Argon2i

We see that, while the pre-calculation can be quite long, ~1 day for 128MiB, it's complexity remains polynomial. We have approximately a $O(n^{2.2})$, where n is the number of nodes of the calculation flow graph.

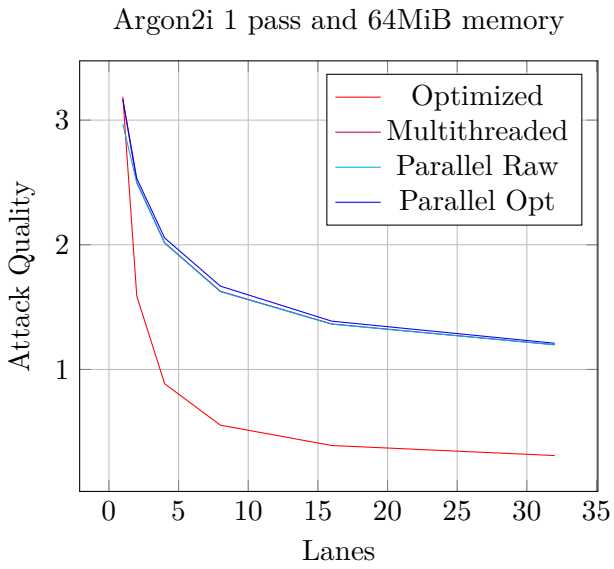

Figure 6: Attack quality depending on number of lanes for mono-pass 64MiB memory Argon2i

We see here that, for a fixed amount of memory, increasing the number of lanes significantly decreases the attack quality.
We also see that, unlike for mono-lane Argon2i, the optimized parallel version of our attack is significantly above the non-optimized one and also beats the multithreaded version making it the best attack in this case. The monothreaded non-parallel version is ineffective when attacking multi-lane Argon2i.

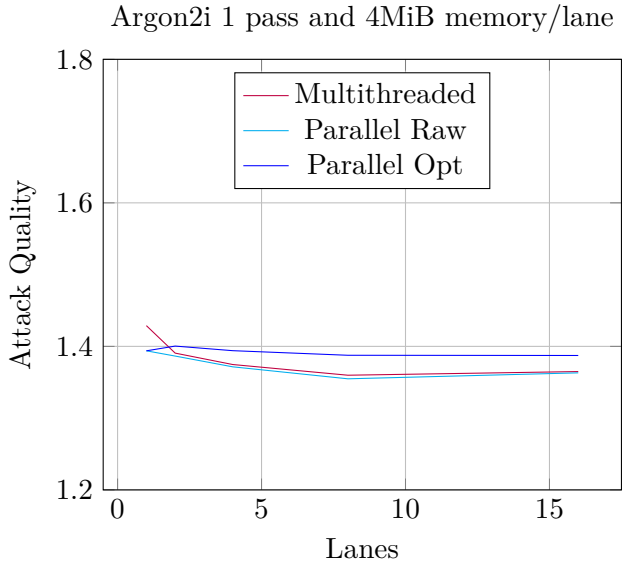

Figure 7: Attack quality depending on number of lanes for mono-pass 4MiB memory/lane Argon2i

These results show that, when maintaining the same lane length, increasing the number of lanes has a very limited effect on attack quality, it still seems to slightly decrease it.

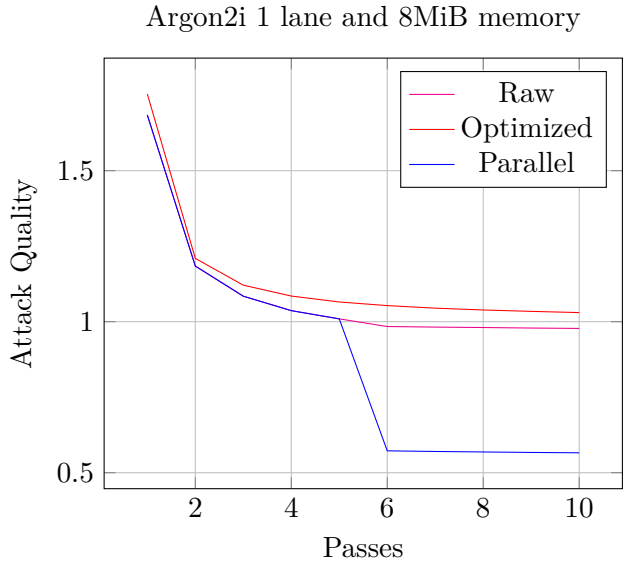

Figure 8: Attack quality depending on number of passes for mono-lane 8MiB memory Argon2i

Increasing the number of passes appears to greatly reduce attack quality. At 6 passes, our non-optimized attack is already not an attack anymore, we also note that our parallel attack sees its quality drop significantly when passing under 1.

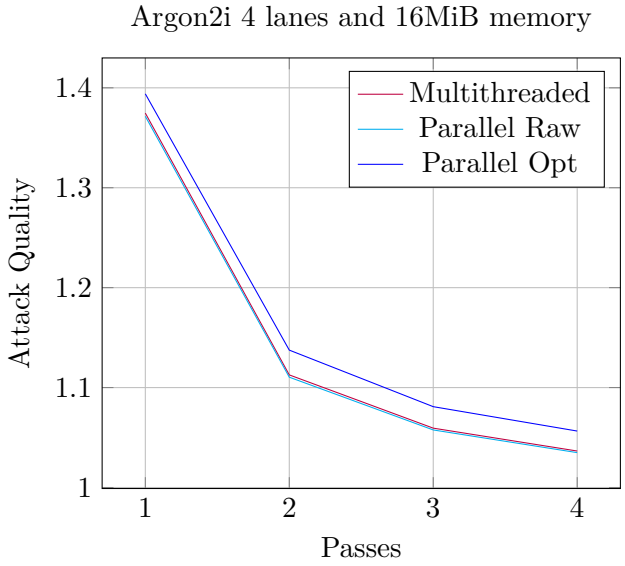

Figure 9: Attack quality depending on number of passes for quad-lane 16MiB memory Argon2i

This plot to check that combining multi-pass with multi-lane do not triggers unexpected effects.

IV.3 Discussion

IV.3.a Attack's effectivity in practical cases

Argon2i's authors suggest 1GiB of memory space but for us, this is actually an upper limit on current systems. Server admins want authentication to be transparent and monopolizing 1GiB of memory for ~1 second is not transparent, not to mention the processor time. The Python implementation of Argon2i argon2_cffi, used for example by Django, uses 512KiB by default (with 2 lanes and 2 passes). We think that most users in the near future will probably focus on parameter choices ranging between these two values, as a compromise between speed and security. For these values, our attack's pre-calculation time is not too high for effective use and we can get a > 3 attack quality with memory space >64MiB.
Wa can conclude that, while our attack does not make Argon2i insecure for practical values of parameters, it can still significantly reduce the cost of an attack.

IV.3.b Results on great values

For great values, the estimated quality of our non-parallel attack seems to hit a limit, this is probably due to the fact that balloon phases cost is very high for a monothreaded attack and thus, for a relatively low number of nodes, it will not perform several balloon phases ($g \simeq n$), limiting the attack quality. We see that the parallel version of our attack does not hit that limit, detail output of our program shows that, for all tests, we had $g \simeq n$ except for the parallel estimator after 2^{19}, for which we had $g \simeq \frac{n}{4}$.
While we could not compute results for higher values due to long calculation times, we can conjecture that, for the parallel attack, gains will continue to increase.

IV.3.c Multi-lane Argon2i

In [AB16b], Alwen & Blocki affirmed that multi-lane Argon2i should be more sensitive to attacks that mono-lane but did not provide effective results supporting their hypothesis. In [ABH17], authors declared multi-lane Argon2i useless, citing [AB16b] which provides a way to achieve parallelism based on simultaneous computation of several mono-lane instances, this method being allegedly less sensitive to attacks than multi-lane Argon2i.
Our results show that, even for our attack, which is optimized for multi-lane Argon2i, at equal lane length, increasing the number of lanes did not increases the attack quality, it may even decrease it.
Based on these results, we can affirm that, for our attack optimized for multi-lane Argon2i, multi-lane Argon2i is not less resistant than the method proposed in [AB16b] would be. Multi-lane Argon2i should even be more secure in practice, considering that the pre-calculation time will be greater that for [AB16b]'s method, especially if, in this last method, all parallel instances have the same graph.

V Conclusion

V.1 Argon2i's security

We conclude that, regarding our attack, Argon2i is secure for current practical parameter ranges but attackers can still get significant improvements in costs using our attack. In addition, Argon2i can become insecure for future parameters ranges.

V.1.a Memory space

We recommend using enough memory space to ensure that the memory-hardness of Argon2i will be effective. We suggest using >1MiB. Using greater amounts of memory increases security, even if attacks are more effective, the cost of attacking itself is still greater. But keep in mind that neither system admins nor users want a slow authentication process. Using too much memory space can make authentication problematic in some use cases, taking resources from effective services or inducing long waiting time when an authentication server is under heavy load. It can also induce sensitivity to some kinds of (D)DoS attacks.

V.1.b Number of passes

We observed that making several passes significantly increase Argon2i v1.3's resistance to our attack. We strongly recommend Argon2i v1.3 to be used systematically with $t > 1$. We recommend to use more passes when using greater memory space. Note that it also increases pre-calculation time as much as multiplying the memory space by the number of passes.

V.1.c Parallel lanes

Unlike affirmed by other researchers, it seems that multi-lane Argon2i is perfectly secure and increasing the number of lane effectively increases it's resistance to our attack. We recommend using 2 ou 4 lanes if the system can achieve this level of parallelism. Keep in mind that the cost of computing multi-lane Argon2i increases if all lanes are not effectively computed in parallel, and thus, effective attack quality also increases.

V.2 Recommendations

V.2.a Use better calculation flow graphs

Argon2i's calculation flow graph is not resistant to time-space trade-off based attacks. While this sensitivity is not sever for current practical parameters range, regarding both, the time needed to compute the function in the standard way and the time needed for attack pre-calculation, practical parameter range will evolve with computers' power and eventually, Argon2i may become insecure, particularly if attacks get improved. See [ABH17] works on stronger calculation flow graphs.

V.2.b Seeded hashing

Our attack's efficiency is based on the fact that it's possible to make the pre-calculation only once for a whole system, and even for all systems with the same set of parameters. It's likely that most attacks on data-independent memory-hard hash functions based on optimizations in the evaluation process will be is the same case. To significantly reduce these attacks efficiency without making the functions data-dependent, we recommend to use add a new value on which the evaluation flow will depend.
We call this value the *seed*. Like a salt, the seed is stored with the hash and should not be different for all passwords. The seed will be used by the hash function when computing random parents for a node. In the case of Argon2i, this can be implemented using the 968 0-bytes at the end of the G^2 function's input, we recommend that (part of) these bytes should be replaced with seed bytes.
Since the seed value is distinct of the password, the function will still be data-independent and insensitive to memory access-based side-channel attacks.
In practice, we think that a 4 bytes seed should be enough for most use case, using 8 bytes being a relatively cheap way to ensure the seed is enough long. Hash functions should be able to use variable length seeds.
Seeded hashing can easily be adapted to current systems which already support salt by using part of the salt as seed, for example, a 8 bytes salt divided into 4 bytes of real salt a 4 bytes of seed.

V.2.c Use good parameter values

There is nothing new, but our work shows how using good values for certain parameters, especially the number of passes but also the number of lanes, can make attacks much less effective.

V.3 Future works

V.3.a Speeding-up pre-calculation

In our work, we did not focused on reducing the time needed for pre-calculation. A constant gain factor may be achieved by modifying the parameters optimization loop, but we have doubt on this since we had to increase the number of trials to get better results and the estimator, which uses 8 times less trials, while usually returning the same values as the final optimizer, sometimes gave sub-optimal values. The optimization loop can also be adapted to parallel computation to gain time on parallel systems. Finally, algorithms used for pre-calculation could potentially be improved. Experimentally, we observed that our pre-calculation's complexity was approximately $O(n^{2.2})$, where n is the number of nodes of the calculation flow graph. With these improvements, it should be possible to use more data-points for the grid search, to get better parameter values, especially for great values of n were the parameters ranges are larger.

V.3.b Developing new attacks

We developed several versions of our attack, none is the best in all cases and some observed phenomena did not received explanations. We think that a deeper analysis of our attack may be helpful for developing more effective future attacks.

V.3.c Making better hash functions

We hope that new (versions of) hash functions will be developed, following our recommendations. There is still room for improvements in memory-hard hash functions.

References

[AB16a] Joël Alwen and Jeremiah Blocki. Efficiently computing data-independent mermory-hard functions. 2016.

[AB16b] Joël Alwen and Jeremiah Blocki. Towards practical attacks on argon2i and balloon hashing. 2016.

[ABH17] Joël Alwen, Jeremiah Blocki, and Ben Harsha. Practical graphs for optimal side-channel resistant memory-hard functions. 2017.

[ANWOW13] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. 2013.

[AS15] Joël Alwen and Vladimir Serbinenko. High parallel complexity graphs and memory-hard functions. 2015.

[BCGS16] Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter. A memory-hard function providing provable protection against sequential attacks. 2016.

[BDK16] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: the memory-hard function for password hashing and other applications. 2016.

[BDKJ17] Alex Biryukov, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson. The memory-hard argon2 password hash and proof-of-work function. Technical report, IETF, 2017.

[BK15] Alex Biryukov and Dmitry Khovratovich. Tradeoff cryptanalysis of memory-hard functions. 2015.

[Coo74] Stephen A. Cook. An observation on time-storage trade off. *Journal of Computer and System Sciences*, 1974.

[DNW05] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In *CRYPTO 2005: 25th Annual International Cryptology Conference*, 2005.

[Per09] Colin Percival. Stronger key derivation via sequential memory-hard functions. 2009.

[PH70] Michael S. Paterson and Carl E. Hewitt. Comparative schematology. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*. ACM, 1970.

[SA15] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. The blake2 cryptographic hash and message authentication code (mac). Technical report, IETF, 2015.

A Get Instructions Algorithm

Algorithm Get Instructions

```
1: function getInsts(G, N, p, t, S, l, s, l̄, s̄, g, para)
2:     P ← p + s
3:     balloon ← ⊤
4:     for n ← 0 to N − 1 do
5:         i ← n[g]
6:         if i = 0 then
7:             balloon ← ⊤
8:             la ← 0
9:             j ← 0
10:        end if
11:        if para then
12:            if ¬isPebbled(peb[lane(n)], n) then
13:                if isPebbled(peb, n)∨unSyncedFree[n] ∨ ¬parentsPebbled(peb[lane(n)], n)
   then
14:                    for all i ∈ ⟦0; P⟦ do
15:                        addBarrier(ans[i])
16:                    end for
17:                    syncPebbles(peb)
18:                    reset(unSyncedFree)
19:                end if
20:                if ¬isPebbled(peb[lane(n)], n) then
21:                    addCalc(ans[lane(n)], n)
22:                    updatePebbling(peb[lane(n)], ans[lane(n)])
23:                end if
24:            end if
25:        else if ¬isPebbled(peb, n) then
26:            addCalc(ans, n)
27:            updatePebbling(peb, ans)
28:        end if
29:        if balloon then
30:            next ← ⊤
31:            : doNext
32:            for all se ∈ ⟦0; s⟦ do
33:                if s̄[la][se + 1] − s̄[la][se] > j then
34:                    nd ← l̄[la] + s̄[la][se] + j
35:                    if para then
36:                        if ¬isPebbled(peb[p+segment(nd)], nd)∧parentsPebbled(peb, nd) then
```

Algorithm Get Instructions (part 2)

```
37:                         if isPebbled(peb, nd)∨unSyncedFree[nd] ∨ ¬parentsPebbled(
    peb[p+segment(nd)], nd) then
38:                             for all i ∈ ⟦0; P⟦ do
39:                                 addBarrier(ans[i])
40:                             end for
41:                             syncPebbles(peb)
42:                             reset(unSyncedFree)
43:                         end if
44:                         if ¬isPebbled(peb[p+segment(dn)], nd) then
45:                             addCalc(ans[p+segment(nd)], nd)
46:                             updatePebbling(peb[p+segment(nd)], ans[p+segment(nd)])
47:                         end if
48:                     end if
49:                     next ← ⊥
50:                 else
51:                     if parentsPebbled(peb, nd)∧¬isPebbled(peb, nd) then
52:                         addCalc(ans, nd)
53:                         updatePebbling(peb, ans)
54:                     end if
55:                 end if
56:             end if
57:         end for
58:         if next then
59:             if ++la >layer(n) then
60:                 balloon ← ⊥
61:                 la − −
62:             else
63:                 j ← 0
64:                 goto doNext
65:             end if
66:         else
67:             j++
68:         end if
69:         if la − l/t > 0 then
70:             for all nd ⩽ l̄[la − l/t] do
71:                 if ¬S[nd]∧isPebbled(peb, nd) then
72:                     if para then
73:                         for all i ∈ ⟦0; P⟦ do
74:                             addFree(ans[i], nd)
75:                             updatePebbling(peb[i], ans[i])
76:                         end for
77:                         unSyncedFree[nd] = ⊤
78:                     else
79:                         addFree(ans, nd)
80:                     end if
81:                 end if
82:             end for
83:         end if
```

Algorithm Get Instructions (part 3)

```
84:         else
85:             K ← S
86:             for all k ∈ ⟦n + 1; min(n + g, N − 1)⟧ do
87:                 addParentsToSet(n, K)
88:             end for
89:             for all k < N do
90:                 if isPebbled(peb, k)∧¬K[k] then
91:                     if para then
92:                         for all i ∈ ⟦0; P⟦ do
93:                             addFree(ans[i], k)
94:                             updatePebbling(peb[i], ans[i])
95:                         end for
96:                         unSyncedFree[k] = ⊤
97:                     else
98:                         addFree(ans, k)
99:                     end if
100:                end if
101:            end for
102:        end if
103:    end for
104:    if para then
105:        for all k < N do
106:            if isPebbled(peb, k)∧¬isFinal(k) then
107:                for all i ∈ ⟦0; P⟦ do
108:                    addFree(ans[i], k)
109:                end for
110:            end if
111:        end for
112:        for all i ∈ ⟦0; maxP⟦ do
113:            rewind(ans[i])
114:        end for
115:    else
116:        rewind(ans)
117:    end if
118:    return ans
119: end function
```

B Post-optimization Algorithms for parallel attack

B.1 Clean Useless Parallel

Algorithm Clean Useless Parallel

```
1: function cleanUselessParallel(pr, N, p)
2:     for n ← N − 1 to 0 do
3:         for all i ∈ ⟦0; p⟦ do
4:             ins ← pr[i]
5:             rm ← ⊥
6:             while ins do
7:                 if isBarrier(ins) then
8:                     rm ← ⊥
9:                 else if node(ins)= n then
10:                    if isCalc(ins) then
11:                        clc ← ins
12:                        rm ← ⊤
13:                    else
14:                        if rm then
15:                            rmInst(clc)
16:                            rmInst(ins)
17:                        end if
18:                    end if
19:                else if rm∧isParent(ins, n) then
20:                    rm ← ⊥
21:                end if
22:                ins ←next(ins)
23:            end while
24:        end for
25:    end for
26: end function
```

B.2 Free Earlier Parallel

Algorithm Free Earlier Parallel

```
1: function freeEarlierParallel(pr, N, p)
2:     for all n ∈ ⟦0; N⟦ do
3:         for all i ∈ ⟦0; p⟦ do
4:             ins ← pr[i]
5:             while ins do
6:                 if isBarrier(ins) then
7:                     lst ← ins
8:                 else if isCalc(ins) then
9:                     if isParent(ins, n) then
10:                        lst ← ins
11:                    end if
12:                else if node(ins)= n ∧ ¬isParent(previous(ins), n) then
13:                    addFree(lst, n)
14:                    rmInst(ins)
15:                end if
16:                ins ←next(ins)
17:            end while
18:        end for
19:    end for
20: end function
```

B.3 Calculate Later Parallel

Algorithm Calculate Later Parallel

```
1: function calcLaterParallel(pr, N, p)
2:     for n ← N − 1 to 0 do
3:         for all i ∈ ⟦0; p⟦ do
4:             ins ←next(pr[i])
5:             while ins do
6:                 if isBarrier(ins) then
7:                     if w then
8:                         w ← ⊥
9:                         if node(previous(ins))≠ n then
10:                            cpInst(clc,previous(ins))
11:                            rmInst(clc)
12:                        end if
13:                    end if
14:                else if isCalc(ins)∧node(ins)= n then
15:                    clc ← ins
16:                    w ← ⊤
17:                else if w ∧ (isParent(ins, n)∨(isFree(ins)∧isParent(clc,node(ins)))) then
18:                    w ← ⊥
19:                    if node(previous(ins))≠ n then
20:                        cpInst(clc,previous(ins))
21:                        rmInst(clc)
22:                    end if
23:                end if
24:                ins ←next(ins)
25:            end while
26:        end for
27:    end for
28: end function
```