Investigation of Practical Attacks on the Argon2i Memory-hard Hash Function

6.857 Computer and Network Security
Massachusetts Institute of Technology

Mitchell Gu
Ryan Berg
Casie Chen
May 18, 2017

Contents

1 Introduction

Modern password hashing functions are designed to protect low-entropy passwords such as "password" or "123456Seven" that can be brute-forced quickly. These hashing techniques have evolved over time as attacks on them have been developed. When plain hashing of passwords became compromised through precomputing hashes of common passwords and saving them in rainbow tables, the standard became salted hashing. As brute forcing salted hashing became more affordable with increased computational power, new and slower password hashing functions were developed. These include PBKDF2 (developed by RSA Laboratories) and bcrypt, which allow parametrization of the hash's runtime by adjusting how many iterations the hash performs. However, these slower hashing functions often have low memory requirements, which allow them to be run on fast, specialized hardware like ASICs and FPGAs. Even hashing functions with high memory requirements often allow for time-memory tradeoffs, and are thus vulnerable to attacks using custom hardware implementations.

Memory-hard hash functions (MHFs) are a new class of hashes developed to make the time-memory tradeoff prohibitively difficult, requiring sufficiently large amounts of memory. This proves problematic for custom hardware attacks, which suffer high memory latencies and require an increase in chip area to store the required amount of memory. The Argon2 hash function, winner of the Password Hashing Competition, is one MHF design that is currently being recommended as a more secure replacement to existing password hashes. Joël Alwen and Jeremiah Blocki have published an attack that claims to decrease the effectiveness of Argon2 in preventing time-memory tradeoffs, thus making it much easier to compute many hashes using custom hardware such as ASICs. In this report, we investigate the significance of their attack and assess some possible improvements.

2 Project Statement

Our motivation for this project was to develop a better understanding of the mechanics behind MHFs, and what sort of weaknesses they have. Our base goal was to understand the hash and the attack well enough to develop a qualitative judgement on just how concerning it is for the future of MHFs in crypto. Beyond that, we also wanted to reimplement/model the attack in software, determine what sort of optimizations were possible or not addressed, communicate with Alwen and Blocki about our proposed optimizations, and potentially implement the attack on an FPGA. We tabled the last goal as we came to a better understanding of the attack, and realized that an FPGA implementation would not be sufficiently advantageous or insightful for benchmarking.

3 Literature Review

Like any candidate cryptographic standard, Argon2 has been the subject of significant scrutiny and some back-and-forth updates as researchers investigate potential attacks against it. So far, this discourse has involved the authors of Argon2 (Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich), a group of researchers at Stanford and Microsoft (Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter), and the authors of the attack we are interested in (Joël Alwen and Jeremiah Blocki, hereafter referred to as A&B). The chronology is as follows:

- Feb. 2015: version 1.1 of Argon2 is published.
- Jul. 2015: version 1.2 is published. Changes include a nonlinear block indexing function and a changed compression function.
- Jul. 2015: Argon2 wins the Password Hashing Competition.
- Aug. 2015: version 1.2.1 is released with another change to the indexing function and some other minor updates.
- Jan. 2016: Boneh, Corrigan-Gibbs, and Schechter release a paper on Balloon hashing that includes an attack on Argon2 that computes the hash in $1/3$ of the memory and equal time as the honest algorithm [1].
- Feb. 2016: A&B publish a paper describing a general attack for all data-independent MHFs - including Argon2i - involving attacking the depth-robustness of the Argon2i computation graph [2].
- Mar. 2016: version 1.3 is released (referred to as Argon2i-B by A&B) that requires new blocks to be XOR'd with the block being overwritten, patching the attack from BCGS [3]. The authors argue that the bounds on attack quality given A&B's attack provide no benefit for practical hash parameters.
- Aug. 2016: A&B publish a response to v1.3 by making several optimizations on their attack for Argon2i graphs and provide simulation data demonstrating favorable attack qualities for many practical hash parameters [4].

The Argon2 authors haven't provided a formal public response to the second A&B paper other than a GitHub comment indicating that the default number of passes would not be increased to 10. In a discussion on the Crypto Forum Research Group mailing list, Dmitry Khovratovich notes a minor misrepresentation of Argon2i's recommendations on choosing parameters, but no further conclusions. Thus at the time of writing of this project, the consensus on how practically important A&B's attack is remains unclear, which is something we hope our project can shed light on.

4 Background

4.1 The Argon2 Hash

A characteristic of many hash functions is a time-area tradeoff: to be able to compute the hash faster, more memory (area) is often required. For a MHF to be effective, an adversary attempting to compute the hash with less than the intended amount of memory will be forced to increase the execution time enough to increase the time-area product. In other words, if an attacker designs a more inexpensive ASIC chip that runs a dishonest Argon2 implementation with less memory, the increase in execution time will be enough to increase the cost for a certain hashrate. In the same vein, an attack on Argon2 would allow the hash to be computed with less than intended memory and a small enough increase in execution time to decrease the time-area product.

The design of Argon2 has two flavors - Argon2d and Argon2i. The former is dependent on the input data to determine which blocks of memory to incorporate at each step, while the latter does this in a data-independent way and therefore is more secure to side-channel attacks. Like the Alwen and Blocki paper, we will focus on the Argon2i flavor as the former is already vulnerable to side-channel attacks by nature of its design (as acknowledged by the authors of Argon2i).

Argon2 starts by taking all its input parameters and running them through its primary hash function H: Blake2b. Blake2b was a candidate hash function for SHA-3 and is based on Daniel Bernstein's ChaCha20 stream cipher. Chacha20 operates in a somewhat similar manner to AES in that the internal state is interpreted as a matrix and a series of operations called quarter rounds permute information in the state by cycling through rows and columns. This initial hash serves the purpose of integrating all the inputs and parameters of the hash and extracting their entropy into a single 64-byte output: H_0.

Then, σ 1024-byte blocks of memory are allocated, where σ is an adjustable parameter of the hash. The first two blocks of memory are initialized by applying the Blake2b hash again to H_0, concatenated by 8 bytes indicating the position of the block. To produce a 1024-byte output, a variant of Blake2b is used that produces variable length outputs. Each subsequent block B_i is computed by choosing the previous block B_{i-1} and a different previously computed block B_j, for $j < i - 1$), and then passing them through the compression function H. The second block B_j is chosen via a probability distribution dependent on H. For an Argon2 computation with τ iterations, the hash will perform τ iterations over all σ memory blocks, filling $\tau * \sigma$ memory blocks total and requiring $\tau * \sigma - 2$ computations of the compression function H.

Clearly, the effectiveness of the Argon2 hash is highly dependent on choosing a good compression function H and a good method of mapping the current index i to another index i' in random-oracle fashion that cannot be predicted in advance. If the compression function G is not ideal, then one could detect patterns in the chaining of its outputs in a differential analysis attack. If the mapping of indices

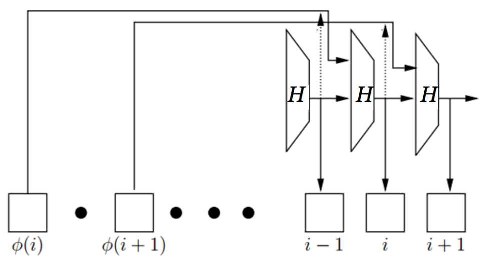


Figure 1: Argon2 mode of operation

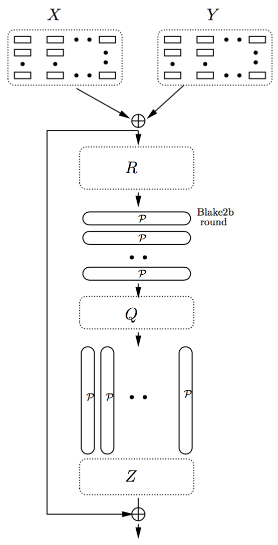


Figure 2: Argon2 compression function H

does not behave like a random oracle, an adversary could compute the hash with less memory in acceptable amortized time by anticipating which memory blocks may be used and which could be evicted.

4.1.1 The Compression Function H

The compression function H accepts two 1024 byte memory blocks X and Y and outputs one 1024 byte memory block that is a strong function of both inputs. It first computes $R = X \oplus Y$, then views R as an 8x8 square matrix of 16-byte registers and permutes the matrix using essentially the same round function as used in Blake2b (and ChaCha20). The result of these round functions Z is then XOR'ed with R again to yield the output.

4.1.2 Indexing

In Argon2i, the indexing of the memory blocks is done data-independently by running H^2 (two rounds of H) on a block of zeroes and a concatenation of several Argon2 parameters. The output is then used to index into a set R of all blocks that have already been populated. The mapping of the H^2 output to the index into R is intentionally very non-linear and favors older memory blocks over new ones.

Finally, after all $\tau * m$ blocks have been iterated over, the output of the hash is taken from the Blake2b hash of the final block. From the design of the Argon2i hash, it's easy to see the high-level intent of the authors: Fill the entire memory space by growing a chain of 1024-byte blocks that are a function of the previous block, but also an older block that is difficult to predict. The approach is simple to understand and easily adaptable to more parallelism, but at first glance would be very difficult to circumvent.

4.1.3 Revisions Made to Argon2i-A

Two changes of key significance to the strength of Argon2i were made before the specialized A&B attack was published. First, the probability distribution that selects the second dependancy block B_j was changed from a uniform distribution to one that that heavily favors more recent past blocks, so as to ensure that all blocks are used roughly the same number of times to compute future blocks. This consideration ensures that there are not certain blocks in the chain that present a higher value to a would-be attacker to store in memory, while the uniform distribution made it much more advantageous to store the earliest blocks in the sequence (as they accrued the most dependants over the course of the hash). The second change affects how passes through the memory work. In Argon2i-A, the last block in a pass p was chained into the first block of pass $p + 1$, but once the initializing blocks of the pass were complete, there was no reason to keep the rest of the previous σ blocks in memory from the last pass. This allowed a potentially trivial amortization to be made, as the honest hash would not need σ blocks to be stored constantly, but only at the end of a pass. In Argon2i-B, this was changed, so that instead of replacing the contents of the block in memory, the output is XOR'ed to produce the new block, thus requiring that the σ memory blocks be constantly utilized.

4.1.4 Simplifying Assumptions

For the rest of the paper we assume that parallelism is 1 for simplicity, but all the ideas we develop can be exended to higher parallelism.

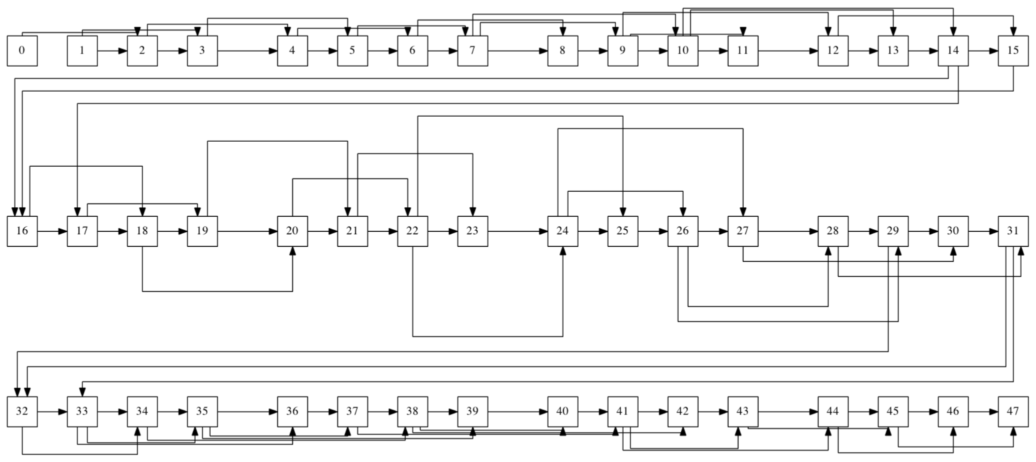


Figure 3: Example computation graph for $\theta = 16$, $\tau = 3$

4.2 The Alwen-Blocki Attack

4.2.1 Modeling the Hash as a DAG

The key framing for the A&B attack is that any member of the Argon2i hash family can be represented as a particular directed acyclic graph where each node corresponds to an block of memory to compute. Since τ passes are done over σ blocks of memory, there are $N = \sigma * \tau$ nodes in the graph labeled with [0..N-1]. Every node (except the first two) has indegree 3, with the three parent nodes represent blocks that must be resident in memory to be able to compute the new block of memory. Two of the parents are the inputs of the compression function H: one being the previous node and the other being the randomly chosen node within σ nodes. The third parent represents the change in Argon2i-B where each block is XOR'd with the node it is overwriting, making the overwritten node a parent of the new node. Figure 3 shows an example graph for $\sigma = 16$ and $\tau = 3$. The XOR edges have been omitted for visual simplicity.

The process of calculating the hash can then be analogized as a graph-pebbling problem, with a pebble on a node signifying that node's presence in memory during a particular step. The first two nodes begin pebbled (initialized), and to pebble a new node, the node's parents must be pebbled. Once the sink node (the last node) is pebbled, the hash is complete. In the honest algorithm, the graph is pebbled with a sliding window of σ pebbles representing the σ memory blocks the honest algorithm runs pebbles over. This uses $O(\sigma)$ memory and takes $O(N) = O(\sigma * \tau)$ timesteps for a time-area complexity of $O(\sigma * 2\tau)$.

4.2.2 Attacking the Graph's Depth-Robustness

Alwen and Blocki made the observation that the randomized graphs generated by Argon2i-B have limited depth-robustness. Limited depth-robustness signifies

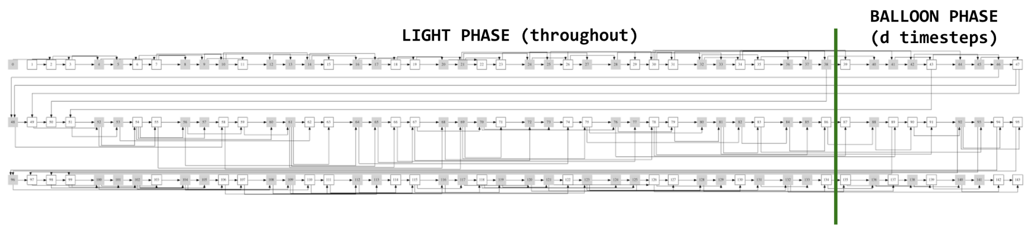


Figure 4: Diagram of balloon and light phases for a larger graph

that one can find a subset of nodes S that is relatively small, but dramatically reduces the depth of the graph. Alwen and Blocki present an algorithm that is able to find a set S for any Argon2i graph, such that the resulting depth of the graph with S removed is at most d, which is a number significantly lower than the original depth N. The significance of this property is that if nodes in S are always kept pebbled, any unpebbled node in the graph can be recovered within d steps. There are many ways to select such a set S, which will be addressed in the section on optimizations.

4.2.3 Light and Balloon Phases of Computation

The attack is broken up into sets of two phases, the light phase and the balloon phase. Each light phase is broken up into g steps which each involve pebbling one sequential node in a layer. At the end of each light phase, the next light phase immediately begins.

The balloon phase occurs concurrently with each light phase, beginning $g - d$ steps from the end of the current light phase. It is the job of the balloon phase to ensure that all parent blocks of the next light phase are pebbled before it starts. It does this by greedily pebbling the depth-reduced graph until it has pebbled all of the remaining parent nodes. This takes at most d operations due to the depth of the reduced graph being bounded by d, ensuring that all possible parents are ready before the next light phase begins. A diagram of how these two phases alternate is shown in Figure 4.

4.2.4 Parallelization and Amortization

The memory usage of the light phase is essentially constant (not quite, as we expand upon in 5.2.2), at $L = O(|S| + \delta * g)$, where $\delta = 3$ is the indegree of the graph. The memory usage of the balloon phase is typically higher, but bounded at $B = O(\sigma)$. Because the balloon phase only takes up d steps for each g step layer, that memory can be freed and pooled with other parallel staggered instances of Argon2i, such that the total memory cost for any instance is $L + \frac{g}{d}B$. A graph of the upper bounds on memory usage for the honest algorithm, single algorithm, pooled attack, and amortized pooled attack is shown in Figure 5. One

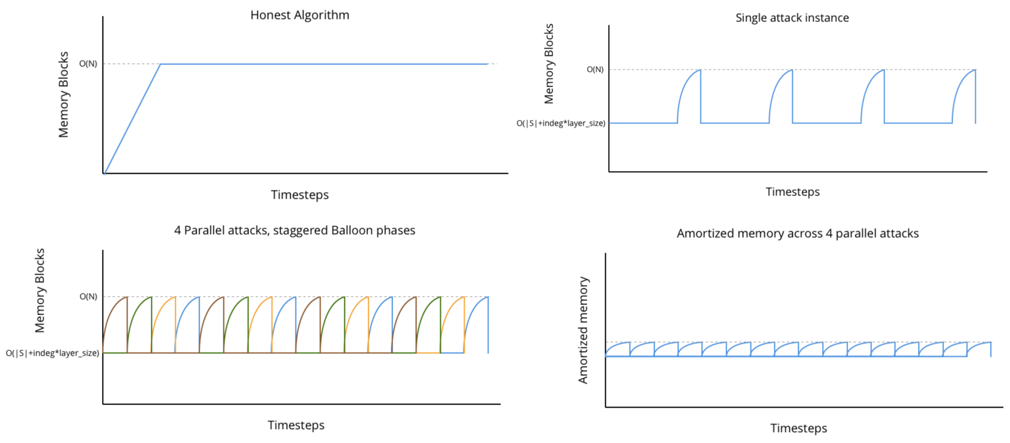


Figure 5: Memory upper bounds for the various algorithms.

can see that once the attack is parallelized and the balloon phases are staggered optimally, the attack's amortized memory is less than the honest algorithm's, yet the computation time is identical. This indicates a favorable time-memory tradeoff and an attack on Argon2i.

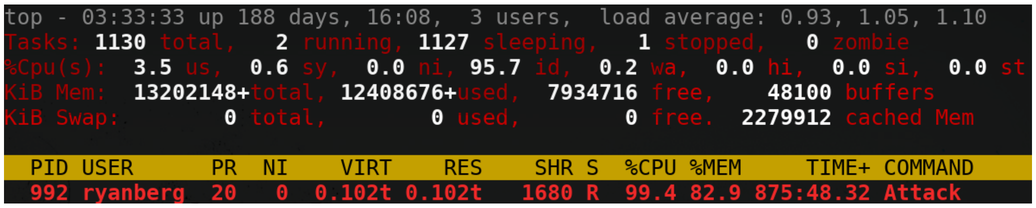
4.2.5 Simulation Results

Alwen and Blocki acknowledged the need to develop actual simulations of attack quality, because asymptotic bounds and theoretical analysis can only go so far. In their most recent paper, they developed C code to simulate their attack for common parametrizations of the Argon2i-B hash. After the optimizations on their attack presented in the paper, they found that for most practical ranges of σ, τ, and parallelism, their attack yields a favorable attack quality.

5 Our Contributions

5.1 Modeling the Attack

We began by using the attack-modeling code that Alwen and Blocki used to write their paper. There were some bugs regarding the undocumented constraints on random number generation on the computer they used, which resulted in segmentation faults on all the machines that we tried the code on. We tracked the segfaults down and applied an appropriate fix, but there was a considerable memory leak that caused the program to crash before completion on one of our computers which had 16GB of RAM. We had access to a server that had 128GB of RAM, and so executed the code there (Fig. 6), but the leak still prematurely terminated the program. Due to the finicky nature of debugging memory leaks in C, and the fact that each attempt was costing upwards of 12 hours of computation,



```
top - 03:33:33 up 188 days, 16:08,  3 users,  load average: 0.93, 1.05, 1.10
Tasks: 1130 total,   2 running, 1127 sleeping,   1 stopped,   0 zombie
%Cpu(s):  3.5 us,  0.6 sy,  0.0 ni, 95.7 id,  0.2 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  13202148+total, 12408676+used,  7934716 free,    48100 buffers
KiB Swap:        0 total,         0 used,        0 free.  2279912 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
  992 ryanberg  20   0  0.102t 0.102t   1680 R  99.4 82.9 875:48.32 Attack
```

Figure 6: screenshot of top, showing RAM usage of .102TB and runtime ≈ 14.5hrs

we decided to instead develop our own modeling/parameterization software in Python.

Our Python software provided classes to generate both Argon2i-A and Argon2i-B computation graphs and plot them in `graphviz`, a graph visualization package. This enabled us to visualize A&B's attack much more intuitively and suggested areas of the attack that could be further optimized.

5.2 Genetic Algorithm Minimization of Depth-Reducing Set S

In their attack, Alwen and Blocki provide an algorithm that quickly computes a set S that is guaranteed to reduce the computation graph G to a given depth d. Their algorithm can be described as follows:

- For a target depth d, choose `#layers` and `gap` such that $\#\text{layers}\cdot(\text{gap}+1) = d$. A reasonable first guess is $\#\text{layers} = \text{gap} = N^{1/4}$.
- View the graph as a grid with `#layers` layers and each layer split into segments of size (`gap`+1)
- Ensure that no path can pass through more than `gap` nodes in for each layer.
  - First, add the first node in each segment in each layer to S. This ensures that there are no series chains of more than `gap` nodes since each node has an edge to the next node.
  - For the randomized edges that go from one node in the layer to another node in the layer, check if the edge *makes progress* with regards to segment position. This means that the edge goes from a node at position i in a its to a node at position $j > i$ in its segment. If the edge doesn't make progress, add the child node to S. Visually, one can imagine this as stacking all segments of a layer vertically and ensuring that all edges move either to the right or downwards.

The strategy of the algorithm is to split the graph into some number of layers and evenly split the max depth amongst these layers such that each layer has max depth of d divided by the number of layers. In Figure **??**, the example graph from earlier is shown, but with shaded nodes representing nodes in A&B's set S. The

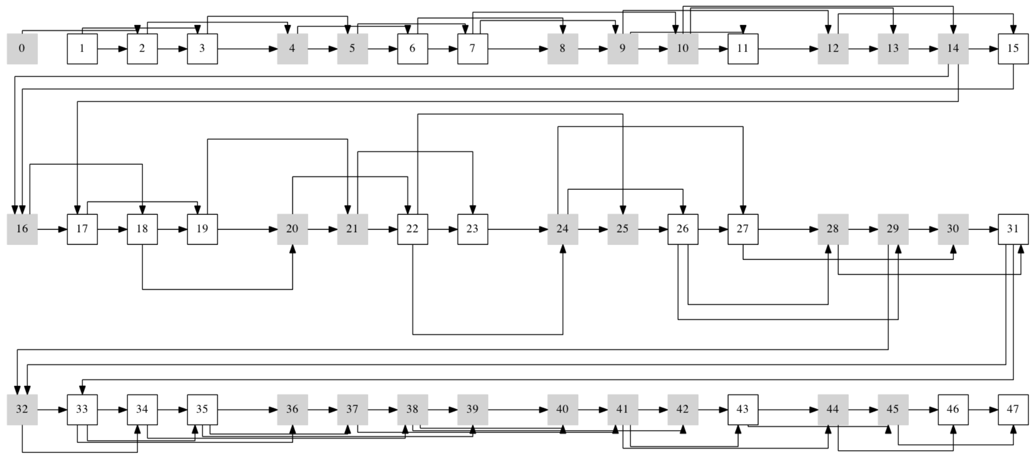


Figure 7: A&B's generated set S. Here, `#layers`=3 and `gap` = 3

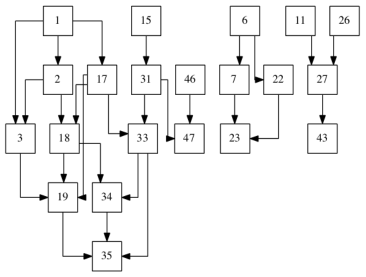


Figure 8: The computation graph in Figure 7 with S subtracted

following figure shows the computation graph with these nodes removed. The theoretical maximum depth for this graph is 9, and the actual depth is 5.

However, the generated set S is not a minimal set: upon visual inspection, it was clear that the algorithm was adding more nodes than necessary and a smaller set S could be constructed. Intuitively, the minimal set S likely does not have its longest path evenly distributed between the layers, as would be enforced by A&B algorithm. The size of S is critical to the attack quality because it correlates to a memory cost that cannot be amortized across multiple attack instances. Thus improving the algorithm for generating S was a natural direction for further optimization.

Finding the absolute minimal set S is likely an NP-hard problem. Therefore, we opted to use a genetic algorithm to evolve an initial population of valid sets into populations of sets smaller in size. The intitial population was seeded by outputs of A&B's approximation algorithm with an input offset parameter shifted to different offsets. To generate a new generation, the procedure was as follows:

- Select the 10 fittest individuals in the population as parents of the next population (10 smallest set sizes)

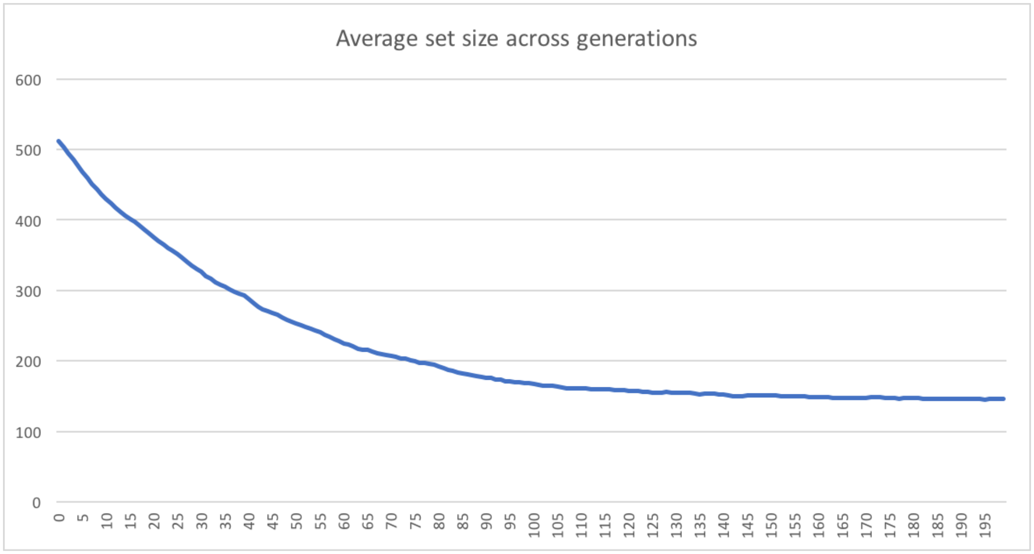


Figure 9: Average set size over 200 generations

- Randomly select pairs of these 10 parents and breed a child from them until a new population of 100 individuals is generated.
- Given two parents, ”cross over” their genomes (sets) by selecting 20 random crossover points from 1 to N and alternating which parent the child set inherits from for each interval between crossover points.
- Randomly mutate the child’s genome with mutation rate 0.005 by flipping each of N nodes’ inclusion in the set with probability 0.005.
- Ensure that the child survives. Here, we check that the depth of the computation graph G minus the child’s set of nodes has depth less than or equal to the target depth.

This primitive genetic algorithm worked remarkably well for converging to a local (possibly global) minimum set size. For a graph with 1000 nodes over 4 passes, the algorithm was able to optimize from set of size 525 provided by A&B’s algorithm to a set of size 146. This process is shown in Figure 9. It was interesting to observe that in the absence of either crossing over or random mutation, the algorithm failed to converge. Only with a mixture of these two evolutionary parameters was the algorithm able to descend into a deeper minimum.

To show the effect of the genetic algorithm visually, Figure 10 shows our previous computation graph example after genetic optimization. One can see that all the rules of A&B’s algorithm are discarded after many generations of crossing over and mutation. The next figure shows the computation graph with the optimized set removed. Clearly, the algorithm has taken full advantage of the maximum depth 9.

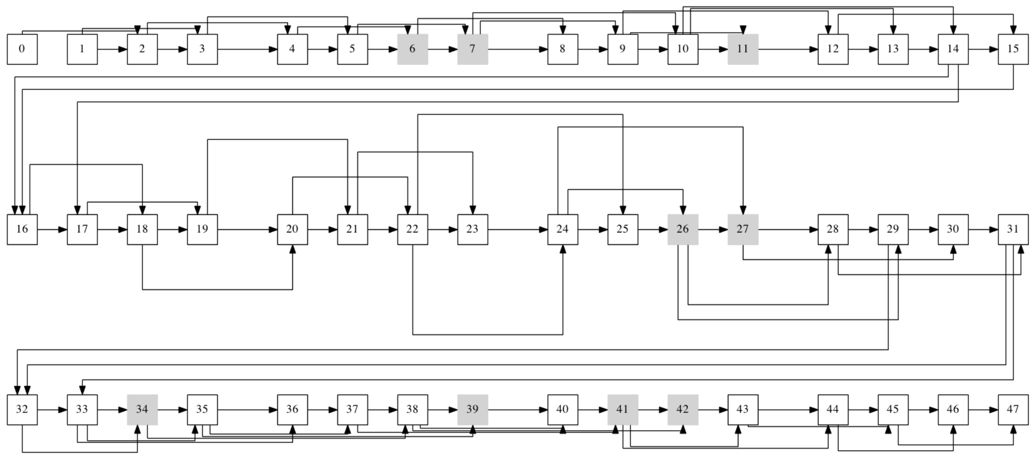


Figure 10: The computation graph with the optimized set shaded

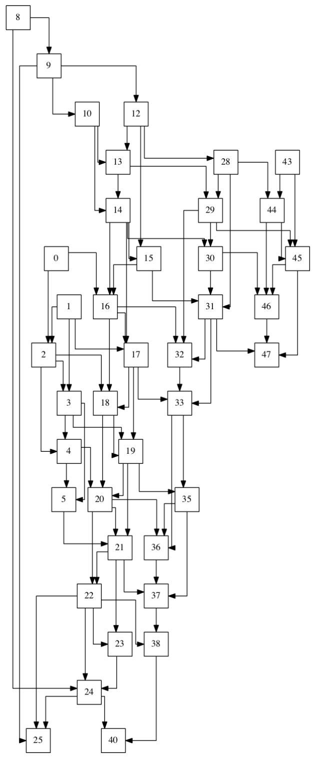


Figure 11: The computation graph with the optimized set subtracted

One practical concern of this approach is that it takes a long time to run. However, since Argon2i graphs are data-independent and only depend on the parameters of the hash (such as σ and τ), an adversary can precompute optimized sets for every hash parameter set of interest. These precomputed sets can then be loaded into a lookup table or ROM chip for an accelerated FPGA or ASIC implementation of the attack.

5.3 Additional Layer of Parallelization to Amortize Memory Costs

While in the canonical A&B attack instances are amortized by spacing out $\frac{g}{d}$ balloon phases such that they can share a balloon phase memory pool, there is an additional parallelization that offers constant-factor gains. Because a set S is progressively pebbled as the hash goes through layers, hashes that are on earlier layers will be consuming less memory on average for both phases. You can pair up hashes in earlier layers with hashes in later layers so as to share a memory pool and reduce total necessary memory.

5.4 Predetermining Actual Reduced Graph Depth

Because the layout of the graph is a function of the parameters of the hash function, and not a result of the input, attackers who have a specific hash in Argon2i that they want to attack can predetermine the actual depth D of the reduced-depth graph, instead of being concerned about the depth upper bound d. As a result, they can parallelize $\frac{g}{D}$ instances, rather than $\frac{g}{d}$.

6 Further Research Questions

Future areas of research include but are not limited to:

- Testing our genetic algorithm on significantly larger graph parameters, in order to more accurately determine efficacy.
- Implementing the optimized, parameter-specific attack in circuitry and benchmarking it with comparison to computers.
- Developing architectures for optimized but generalized circuits. Ideal parameters are specific to each Argon2 hash function (graph-specific), and are not generalized to the Argon2 family as a whole, so developing circuits that compromise/adapt to specific hash functions would reduce the total cost for a dedicated attacker.
- Applying the method of attack to other notable MHFs.
- Developing MHFs that do not have constant indegree, and/or expanding attack to compromise those graphs.

7 Conclusion

The attack proposed by Alwen and Blocki presents a considerable theoretical threat to Argon2. What is currently lacking is a matter of constant-factor optimization, which we have expanded upon in our models and proposals. However, the hardware-optimized attack is not generalizable to different members of the Argon2 family, and it is unlikely for most targets to utilize the same member of the family. This requires attackers to either buy quite sub-par generalized hardware, or a different set of ASICs for each target, which presents a significant pragmatic limitation.

The A&B attack is a concerning threat to any MHF with computation graphs of fixed indegree, practically speaking it is still an improvement over the disparity between ASICs and computers in performace on non-memory-hard hashing for passwords. Since most systems take a while to upgrade to modern standards, the sooner MHFs are adopted, the better.

8 Acknowledgements

We would like to thank Ronald Rivest, Yael Kalai, Sunoo Park, and the rest of the 6.857 course staff for their support and guidance throughout the semester and the course of this project. We would also like to thank Tristan Honscheid '18, for providing us with access to high-RAM computing.

Special thanks to Jeremiah Blocki for providing feedback on our proposed optimizations.

9 References

[1] D. Boneh, H. Corrigan-Gibbs, and S. Schechter, "Balloon hashing: A memory-hard function providing provable protection against sequential attacks." Cryptology ePrint Archive, Report 2016/027, 2016. `http://eprint.iacr.org/2016/027`.

[2] J. Alwen and J. Blocki, "Efficiently computing data-independent memory-hard functions." Cryptology ePrint Archive, Report 2016/115, 2016. `http://eprint.iacr.org/2016/115`.

[3] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: the memory-hard function for password hashing and other applications," 2017. Version 1.3. `https://github.com/P-H-C/phc-winner-argon2/raw/master/argon2-specs.pdf`.

[4] J. Alwen and J. Blocki, "Towards practical attacks on argon2i and balloon hashing." Cryptology ePrint Archive, Report 2016/759, 2016. `http://eprint.iacr.org/2016/759`.

10 Appendix

Our Python code, both for modelling/graphing the Alwen-Blocki attack and reducing the set S via genetic algorithm, can be found at `https://gist.github.com/ryantberg/71dd6bb5149189a4d0b793aa943e1a2c`