A More Elaborate Computation Model for Analyzing Balloon Hashing

Cui Hongrui, Sha Jinrui

December 12, 2017

1 Introduction

MHF's intuition Memory hard hashing function has been widely applied as password hashing functions and as proof-of-effort functions. The reason why the quality of memory hard[1] is preferable is the asymmetric gap between the computing power of commodity server (which the majority of honest users will use), and integrated circuits of different architectures (like ASIC, FPGA, and GPU.) More specifically, consider that a malicious user trying to brute force every possible input combination in order to obtain the log-in password after obtaining the hashed password files from a compromised authentication server, or one who tries to use custom circuit to compute proof-of-effort functions. In either settings, the inputs are irrelevant from each other (we consider the most naive brute force attack), and when this is the case, features like parallelism and pipelining can help custom circuits beats computers of common architecture. The authors of [3] claims that it takes 100000 times more energy to compute a SHA-256 hash on a general-purpose x86 CPU than it does to compute SHA-256 on special-purpose hardware. The advantage lies that when the input are completely independent, the pipeline can hardly be disturbed (recall that modern CPUs do a lot of work to prevent this from happening, like when a bench command is encountered.) And thus such circuits can achieve as high as one hash per clock cycle.

The intuition behind memory hard functions is that although the computing power of custom circuits have taken absolute advantage, constructing enough memory on VLSI is a technical challenge not yet solved. Two main types of memory in widely use nowadays are DRAM and SRAM. DRAM is relatively cheap and has large volume, but are considerably slow compared to modern CPU, not to mention custom circuits. SRAM, on the other hand, is fast , but every single memory cell in SRAM takes more transistors than that in DRAM, and thus a SRAM circuit takes up more space and energy that DRAM circuit of the same volume. Design of custom circuit usually use SRAM on board in order to match the speed of other components, while in commodity servers, DRAM and cache made of SRAM are used in combination. And therefore computing any function on custom circuits and on commodity server are limited to two strategies:

[1] for concrete definitions, see definition 3

- Use custom circuit, which has faster computation speed and limited memory
- Use commodity server, which has slower computation speed and large memory (although slow)

Intuitively, if a function has the property that when working memory is smaller than a particular threshold, then the time to finish the computation becomes considerably longer. In particular, given concrete parameter settings, if such increase in time makes computational time on custom circuits greater than or equal the time honest users would take, then such memory hard property will render the advantage of custom circuits useless.

1.1 Theoretical analysis

The analysis of memory hard functions usually use a "graph reduction" method in order to simply the analysis process. That is, after defining a pebbling rule on a DAG (the data dependency graph of the MHF), any valid algorithm computing the function can be converted (with high probability) to pebbling on that graph. And by graph labeling[4], any legal graph pebbling can be converted to an algorithm computing the function. And thus in order to lower bound the time to compute the function, one only needs to lower bound the steps to pebble the underlying DAG. Some notable functions include Balloon hashing, scrypt, argon2, catena, and a function proposed by Alwen et.al. from [1] that is proved to enjoy memory hard property under a parallel setting.

Balloon hashing Balloon hashing[3] is a hashing function proposed by Dan Boneh et.al. With the application of password hashing algorithm in mind, the algorithm was designed to have three important properties:

- *Memory hardness* product of time (T) and working space (S) is proved to be greater than $n^2/32$
- *Password independent* the memory access pattern is independent of the password being hashed.
- *Performance* the Balloon algorithm is easy to implement and it matches or exceeds the performance of the fastest comparable password-hashing algorithms

The sparking point of the analysis of balloon hashing is that it avoids asymptotic notations whenever possible, and thus under the assumptions the exact lower bound of the function can be found.

Work of Alwen et.al. An impressing research line of Alwen et.al. focused on extending the analysis of memory hard function to a parallel setting and device attacks on known sequential memory hard functions. In particular, they designed general attack utilising parallelism.

1.2 Our contribution

In this work, we presented a new model for computing the memory hard functions. Under the assumption of this model (which is considered valid under current electronic technology), the lower bound of known memory hard functions can be lowered. More specifically, we can lower the ST bound of balloon hashing from $r \cdot n^2/32$ to $c \cdot r \cdot n$, and reduce the cumulative complexity $CC(G)$ of the graph constructed from [1] from $\Omega(\tilde{n^2})$ to roughly $O(n \cdot \log^3(n))$.

2 Model Specification

In this section we formally describe our model and prepare for the proof of our results in section 3. The intuition behind our model is that the common means in constructing a memory-hard function is by carefully constructing the underlying data dependency graph, to make re-computation (which is unavoidable since by assumption custom circuit has only limited memory) takes very long time, as the function runs towards the end. But those values re-computed has already been computed, and fetching a block of memory with arbitrary address from DRAM takes constant time. This gives rise of the following intuition: If we can combine DRAM and custom circuit together, when such re-computation becomes not advantageous compared to memory fetching, then can we improve the performance of custom circuit? After survey of current VLSI and DRAM technology, we believe our model is useful given real parameter settings.

Constant definition We will use the following constant throughout the analysis. We use T to denote the absolute time needed to for a computational entity to acquire the hash of a given input. In the Random Oracle setting, this is the time required to access a particular random oracle call. We use $c \cdot T$ to denote the time required to fetch a block of memory (equals to the size of RO output), and thus c is the ratio representing the fetch time over the hash time[2]. We will see later that this value correspond to the rounds that a legal pebbling must wait after making a pebble fetch move.

2.1 Pebbling in DRAM model

In this section we define rules of sequential pebbling game in our elaborated DRAM model.

Definition 1. *Let $G = (V, E)$ be a directed acyclic graph, a legal pebbling on G given S pebbles and target $T \subset V$ is a sequence of moves, measured in rounds. Each move must be legal according to the following rules:*

1. *Put a red pebble on source node*
2. *Put a red pebble on a node only if all of its parents have red node or blue node in the previous round*
3. *Put a blue pebble on any nodes previously pebbled*
4. *Remove a pebble from any node*

[2] c is approximately 1000

Additional constraints are at any time, there can be only at most S red pebbles, and blue pebbles are removed automatically one round after placement. Move 1, 2, and 4 takes one round and move 3 takes c rounds. A legal game is valid only if it begins with no node on the graph and end with all nodes in T has been pebbled at least once.

Note that in our definition of pebbling games, only one rule is added compared to the pebbling rules in [1, 3], that is the blue pebble placement rule. This is an simplified abstraction of DRAM fetching in the real world. By our assumption, a block can only be used immediately after being fetched, then it is lost. Of course in the real world, certain cache method is most likely applied to save some time when, say, one block is repeatedly fetched. But as we will later show, this somewhat naive abstraction will save considerably amount of time given real parameter setting.

2.2 DRAM computational model

In this section we define the theoretical computational machine on which algorithm with DRAM fetching is executed.

Definition 2. *Let $\mathcal{M}$ be a computational machine with available memory blocks of number σ and an instruction set as follows:*

- ***HASH**: compute the hash[3] of a given set of inputs*
- ***STORSRAM**: store a block into fast memory*
- ***LOADSRAM**: load a block from fast memory*
- ***STORDRAM**: store a block into slow memory*
- ***LOADDRAM**: load a block from slow memory*

Any algorithm under this model is a sequence of instructions mentioned above. The first instruction takes time T, the second and the third instructions takes negligible time. For the simplification of our analysis, we consider that the fourth instruction is executed whenever a new block is computed. The fifth instruction takes time $c \cdot T$.

Note that this model can be easily modified to model computation on circuits without slow memory (the model previous analysed), just disable the fourth and the fifth instruction and we are done.

2.3 Memory hardness in the sequential setting

In this section we define memory hardness in the sequential setting. The idea behind this definition is that such property of a function is desirable: the majority of the complexity of the optimal algorithm (in terms of time complexity) will be spent on memory fetching, which means that no mater how fast the computational module is, as long as the memory module is stable (not having rapid development), then the development in faster computational module will not

[3]standard cryptographic hash function like SHA-256 or blake2b

result in considerable boost in execution time. If such property is real, then we can construct, say, a public block-chain using this function as its proof-of-work consensus method, without having to worry about custom circuit will make computational power overly centralized (a problem severely affecting bitcoin.)

Definition 3. *Fix a function f, then it is considered to be memory-hard in a computational machine $\mathbb{M}$ in definition 2 if given the optimal algorithm that can correctly output the result, the ratio of the time taken to fetch blocks from DRAM (denote as T_f) over the time taken to compute underlying hash function (denote as T_h) is greater than a constant c_0.*

The idea behind this definition is similar to Amdahl's law [2]. In particular, even if the hashing hardware has evolved into a level where all the hashes summed up in one execution of the function instance can be viewed as negligible, then the performance boost is still bounded by a constant $1/c_0$. Although we currently are not aware how the optimal strategy are like given common graphs, the analysis in section 3 shows that for random sandwich graph, for the optimal strategy, the ratio of fetching time T_f over hashing time T_h must satisfy a certain lower bound.

Armed with the definition above, we are ready to prove the time reduction in some proved

3 An example: upper bound on sandwich graph

In this section we prove that given our realistic parameter settings, fetching from memory will definitely become an advantageous strategy compared to re-computation. We use sandwich graph from [3] as an example. We take our ideas from the proof in [3], especially, we are interested in the *big-spread* property and the *everywhere-avoiding* property.

3.1 Big spread lemma variant

Room for improvement The *big spread lemma* plays an important role in the security proof of balloon hashing algorithm [3], however, the theorem require that the subset being considered, $V\prime$ have the same size of the pebbles available by the adversary. This is also reflected in Lemma 30, which bound the probability of the graph being too "dense". Recall in Lemma 30, for $\delta = 3$, every sandwich graph is an (m, m, n/16)-consecutively-avoiding graph, for $n_0 > 16$ and all $n_0 \leq m < n/6$ except with probability $P_{consec}(n, d, n_0) \leq 2 * 8 \cdot d \cdot n \cdot 2^{-n_0/2}$; And for $\delta = 7$, such stack of sandwich graph is (n/64, n/32)-everywhere-avoiding graph when $n > 2^{11}$, except with probability $P_{every} \leq 128 \cdot d \cdot 2^{-n/50}$. Note that in the later part of the lemma, the size of V' is defined as $m = \frac{n}{2^{\omega+1}}$, and thus m grows with n.

This limitation that the subset of V begin fixed by the number of pebbles of the adversary is somehow inconvenient, in that to make the probability of bad event small, one need the number of pebbles available grow, which is the contrary of natural instinct, as the more pebble the adversary acquires, the easier the attack (re-computation) should be. Although the result in [3] is absolutely correct, as we can view the growth of m as the result of the growth of n, an if the subset V' being considered too small, the variables sampled independently

uniform at random from the last layer would have a bigger probability to being not so well-spread. Still, we want to separate the size of subset (m) and the number of pebbles ($|V'|$).

Lemma 1 (Big Spread Lemma Variation)**.** *For all positive integer $\delta \geq 3$, $\omega \geq 2$, n_0, and n, and for all positive integers m such that $2^\omega < m < 2^{2-\omega+\frac{2\omega}{\delta e}}$, a list of δm elements sampled independently and uniformly at random from $1, \ldots, n$ is an $(n_0, n/2^\omega)$-well-spread set with probability at least $1-2^{(1-\omega)\frac{\delta}{2}m} \cdot (2^{\omega m}+2^{\omega+1}2^{\omega n_0})$*

Proof. The Strategy to bound the bad event that such property does not exist is the same as that used in [3]. Let $R = (R_1, \ldots, R_{\delta m})$ be integers sampled independently and uniformly at random from $1, \ldots, n$, we want to prove that for all subset $S \subseteq R$ of size at most n_0, $spread_S(R) \geq n/2^\omega$. To do so, we first define a bad event B, and then show that bounding Pr[B] is enough to prove the lemma, then bound Pr[B] to complete the lemma.

The Bad Event B Write the integers in R in non-decreasing order as $X_1, \ldots, X_{\delta m}$, then define $X_0 = 0$, $X_{\delta m+1} = n$. Let bad event B be the event that there exists a set $S' \subseteq X_1, \ldots, X_{\delta m+1}$ of size at most (n_0+1), such that $\sum_{X_i \in S'} X_i - X_{i-1} \geq (1-2^{-\omega})n$.

Whenever there exists a set $S \subseteq R$ of size at most n_0 that cause $spread_S(R) < n/2^\omega$, then bad event B must occur. Assuming that such a bad set S exists, construct a set $S' = S \bigcup X_{\delta m+1}$ of size at most $n_0 + 1$, Then we compute

$$\sum_{X_i \in S'} (X_i - X_{i-1}) = n - X_{\delta m} + \sum_{X_i \in S} (X_i - X_{i-1}) = n - \sum_{X_i \notin S} (X_i - X_{i-1}) \quad (1)$$

The last inequality holds because $X_{\delta m} = \sum_{X_i \in R} (X_i - X_{i-1})$. Now that $Spread_S(R) < n/2^\omega$, so we have bad event B occurs. Thus, $\Pr[Spread_S(R)] \leq \Pr[\text{B}]$, and therefore bounding a upper bound of bad event B is sufficient to prove the lemma.

Strategy to bound Pr[B] Let D be a random variable denoting the number of distinct integers in the list of random integers R. For any fixed integer $D^* \in \{1, \ldots, n\}$, we can write:

$$\begin{aligned} Pr[B] &= Pr[B|D < d^*] \cdot Pr[D < d^*] + Pr[B|D \geq d^*] \cdot Pr[D \geq d^*] \quad (2) \\ &\leq Pr[D < d^*] + Pr[B, D \geq d^*] \quad (3) \end{aligned}$$

In the following proof, we take $d^* = \delta m/2$.

Bounding $Pr[D < d^*]$ The probability that this event occurs is at most the probability when we throw δm balls into n bins, all the balls fall into a set of $\delta m/2$ bins.

$$Pr[D < d^*] \leq \binom{n}{d^*} (\frac{d^*}{n})^{\delta m} \leq (\frac{n \cdot e}{d^*})^{d^*} (\frac{d^*}{n})^{\delta m} = (\frac{d^*}{n})^{\delta m - d^*} e^{d^*} \quad (4)$$

By the hypothesis of the lemma, $m < 2^{(1-\omega+\frac{2\omega}{\delta})\frac{2n}{\delta e}}$. Therefore, we have

$$Pr[D < d^*] \leq (\frac{\delta m e}{2n})^{\delta m/2} \leq 2^{[(1-\omega)\frac{\delta}{2}+\omega]m} \quad (5)$$

Bounding $Pr[B, D \geq d^*]$ First I would like to re-state the lemma 8 in the original work [3], because this lemma is useful in the proof of big spread lemma.

Lemma 2. *Let $(R_1, \ldots, R_d)$ be random variables respecting integers sampled uniformly, but without replacement, from $\{1, \ldots, n\}$. Write the Rs in ascending order as $(Y_1, \ldots, Y_d)$, and let $Y_0 = 0$, and $Y_{d+1} = n$. Next define $L = (L_1, \ldots, L_{d+1})$, where $L_i = Y_i - Y_{i-1}$. Then for all functions $f : \mathbb{Z}^{d+1} \to \{0, 1\}$, and for all permutations π on $d + 1$ elements,*

$$Pr[f(L_1, \ldots, Ld + 1) = 1] = Pr[f(L_{\pi(1)}, \ldots, L_{\pi(d+1)}) = 1]. \tag{6}$$

Particularly, this lemma shows that if we define a function that maps a list of segments (separated from $\{1, \ldots, n\}$ by the δm random integers) in to $\{0, 1\}$, such that if bad event B happens, then it is one. Using this lemma, we can define a permutation on those integers that if it appears in a bad set, then we shift it with the first $n_0 + 1$ integers in the list, and thus we can only focus on the first few elements on the list without changing the result. As lemma 2 require the list to be mutually distinct, we have to transform the random list in to a distinct one first.

Step 1: Bad event B occurs whenever there is a subset $S' \subseteq X = (X_1, \ldots, X_{\delta m+1})$ of size at most $(n_0 + 1)$, such that $\sum_{X_i \in S'} (X_i - X_{i-1}) \geq (1 - 2^{-\omega})n$. Whenever bad event B occurs, there is also a subset S'_{dist} of distinct integers that also satisfy the sum equation, and also has the size $(n_0 + 1)$. (This is because if duplicate exists, we can always delete duplicates, without changing the sum, and then add new integers to make the sum even bigger.) thus we can use lemma 2 to simplify calculation.

Step 2: we consider the case there is explicitly d distinct integers in the list R, and then use union bound to deduce the final probability. Write the d distinct integers in R is ascending order as $Y = (Y_1, \ldots, Y_d)$. We want to bound the probability that some subset Y'_{dist} of size n_0 is bad. (Note that the only change from the original lemma is that I changed m into n_0.) Then define L_i as in lemma 2. Let the set of indices $I \subseteq \{1, \ldots, d + 1\}$ of size $n_0 + 1$ be the set of index of bad subset. That is, $S'_{dist} = \{Y_i | i \in I\}$. Then define the permutation $\pi : \mathbb{Z}^{d+1} \to \mathbb{Z}^{d+1}$, such that if $i \in I$, then L_i appears in the first $n_0 + 1$ elements of $(L_{\pi(1)}, \ldots, L_{\pi(d+1)})$. Define function $f : \mathbb{Z}^{d+1} \to \{0, 1\}$ such that if its first $n_0 + 1$ arguments sums to at least $(1 - 2^{-\omega})n$, returns 1 and it returns 0 otherwise. Therefore, we have:

$$Pr[\sum_{i \in I} L_i \geq (1 - 2^{-\omega})n | D = d] = Pr[f(L_{\pi(1)}, \ldots, L_{\pi(d+1)}) = 1 | D = d] \tag{7}$$

$$= Pr[f(L_1, \ldots, Ld + 1) = 1 | D = d] \tag{8}$$

$$= Pr[(L_1 + \cdots + L_{n_0+1}) \geq (1 - 2^{-\omega})n | D = d] \tag{9}$$

Step 3: the event defined above can only happen when the other $d-(n_0+1)$ integers falls into the right most $2^{-\omega}n$ bins, and then we have:

$$Pr[(L_1 + \cdots + L_{n_0+1}) \geq (1 - 2^{-\omega})n | D = d] \leq (\frac{1}{2})^{\omega(d-n_0-1)} \tag{10}$$

Above is the probability that a single set is bad, we then sum it over all $\binom{d+1}{n_0+1}$

possible size $n_0 + 1$ subsets, to get:

$$Pr[B|D = d] \leq \binom{d+1}{n_0+1}(\frac{1}{2})^{\omega(d-n_0-1)} \tag{11}$$

$$\leq 2^{(1-\omega)d+\omega n_0+\omega+1} \tag{12}$$

Equipped with $Pr[B|D = d]$, we can then proceed to calculate $Pr[B, D \geq d^*]$. Note when $\omega > 1$, $Pr[B|D = d]$ is non-increasing in d. And therefore we have:

$$Pr[B, D \geq d^*] \leq 2^{(1-\omega)d^*+\omega n_0+\omega+1} \cdot \sum_{d=d^*}^{\delta m+1} Pr[D = d] \tag{13}$$

$$\leq 2^{1+\omega} \cdot 2^{(1-\omega)\frac{\delta m}{2}+\omega n_0} \tag{14}$$

Completing the proof By the calculation above, we have

$$Pr[B] \leq Pr[D < d^*] + Pr[B, D \geq d^*] \tag{15}$$

$$\leq 2^{(1-\omega)\frac{\delta m}{2}} \cdot (2^{\omega m} + 2^{1+(n_0+1)\omega}) \tag{16}$$

□

After proving the big spread lemma variant, we are interested in using it to deduce everywhere avoiding property of the sandwich graph, with the size of the subset $V' \subset V$ and the pebble constraint independent.

3.2 Everywhere avoiding property

An analysis of original strategy In the original work [3], the author proves a *consecutive avoiding property* over all possible size of m, and then for m of fixed size $(n/64)$, proves a tighter *everywhere avoiding property.* The reason of combining these two property is that the first one allows the number pebbles to be arbitrary within a given range. In particular, one can consider m to have the same number of the pebbles the adversary owns. In order to prove a tighter bound on $S \cdot T$ (like in part b of Lemma 31), one needs to first apply consecutive avoiding property and then use the everywhere avoiding property. What we want to do is to acquire the probability that given at most n_0 pebbles, any m pebbles on a standard sandwich graph is an $(n_0, m, 2/2^{\omega})$ avoiding set.

Lemma 3 (Everywhere Avoiding)**.** *Let $G = (U \bigcup V, E)$, be a δ-random sandwich graph on 2n vertices. Then for all positive integer m that satisfy the assumption of lemma 1, and n_0 such that n_0 is no bigger than the upper bound of m, we have that given n_0 pebbles at most, every subset $V' \subseteq V$ of size m, is a $(n_0, n/2^{\omega})$ avoiding set with probability at least $1 - (\frac{n}{m})^m \cdot 2^{(1-\omega)\frac{\delta m}{2}} \cdot (2^{\omega m} + 2^{1+(n_0+1)\omega})$*

Proof. Fix the size of n_0 and m, all we need to do is to apply union bound to all possible choice of V', which has $\binom{n}{m}$ in total. Apply this inequality to simply the bound:

$$\binom{n}{m} = \frac{n \times (n-1) \times \cdots \times (n-m+1)}{m \times (m-1) \times \ldots 1} \leq \frac{n^m}{m^m} = (\frac{n}{m})^m \tag{17}$$

And thus by lemma 1, we have the probability that a single bad event occurs is at most $2^{(1-\omega)\frac{\delta}{2}m} \cdot (2^{\omega m} + 2^{\omega+1}2^{\omega n_0})$, and thus the probability that for all choice of V', the bad event occurs is at most $(\frac{n}{m})^m \cdot 2^{(1-\omega)\frac{\delta m}{2}} \cdot (2^{\omega m} + 2^{1+(n_0+1)\omega})$. And thus the lemma is proved. □

3.3 A bound on the pebbling moves of random sandwich graph

Bounding pebbling moves And now for the sake of our own objective, we would like to calculate the bound of re-computation of a subset of vertices on any layer. To do this, we need to first compute the moves needed to pebble a subset of vertices. First define a variable to denote the pebbling moves needed to pebble a subset of m vertices on the condition that except on those m vertices, there can be at most n_0 pebbles on any vertices of the graph.

Definition 4. *Let G be a stack of random sandwich graphs. Then define T_r be the pebbling moves needed to pebble a subset V' of vertices on layer r, which has not received any pebble on the beginning of the moves, and at the end of the moves, the topologically last one receives a pebble. The condition is that there can only be at most n_0 pebbles on the graph.*

Now that we have defined the amount to calculate, we can use induction reasoning to calculate T_r.

Lemma 4. *Given the average case that $T_0 = \frac{m+n}{2}$, we have:*

$$T_r \geq -\frac{m}{\frac{n}{m2^\omega} - 1} + (\frac{n}{m2^\omega})^r(\frac{m+n}{2} + \frac{m}{\frac{n}{m2^\omega} - 1}) \tag{18}$$

where $0 \leq r \leq d$, d is the layer of the stack of random sandwich graph.

Proof. By the definition of T_r, we have that $T_r \geq m + \frac{n}{m2^\omega}T_{r-1}$. And thus we have:

$$T_r \geq m + \frac{n}{m2^\omega}T_{r-1} \tag{19}$$

$$T_r + \frac{m}{\frac{n}{m2^\omega} - 1} \geq \frac{n}{m2^\omega}(T_{r-1} + \frac{m}{\frac{n}{m2^\omega} - 1}) \tag{20}$$

$$\tag{21}$$

If we define $Y_r = T_r + \frac{m}{\frac{n}{m2^\omega}-1}$, and change the inequality into equality, we can have

$$Y_r = (\frac{n}{m2^\omega})^r \cdot Y_0 \tag{22}$$

And thus $T_r \leq -\frac{m}{\frac{n}{m2^\omega}-1} + (\frac{n}{m2^\omega})^r(\frac{m+n}{2} + \frac{m}{\frac{n}{m2^\omega}-1})$. □

Then we define the lower bound of re-computation needed in the process discussed above.

Definition 5. *Let Recompute(r) be the pebbling moves needed in the process to pebble m vertices on layer r. More specifically, the pebbling moves needed to pebble the direct predecessors of those m vertices.*

By the definition of *Recompute*(r) and lemma 4, one can see that *Recompute*$(r) \geq -m + T_r$, and it grows exponentially with r.

Layer	Ratio
1	0.01
2	0.04
3	0.15
4	0.62
5	2.46

Table 1: The ratio of re-computation over DRAM fetching

3.4 A comparison on re-computation and DRAM fetching

Parameter setting and result In the following comparison section, we take the block size, which the output length of the underlying cryptographic hashing function to be 256 bit, and the block in one layer of the graph to be $2^2 2^4$. We take $\omega = 3$, and the ratio of the latency of fetching a DRAM block over the ratio of the time needed to output a block from Random Oracle, $c = 1000$.

The result of the ratio of naive DRAM algorithm (only fetch from memory, never re-compute), and the lower bound proved in lemma 4 are given together with the depth of layer in figure 1. We can see from the ratio that DRAM fetching becomes advantageous once the graph gets deeper than 5 layers. However, this is a loose bound, as optimal strategy might combine these two strategies, using only the optimal moves. An intuitive idea is that when other parameters other than the depth of the graph are fixed, those re-computation optimal moves can only occur at the first few layers, and thus given the fact that r is growing, the combination boost is bounded by a constant.

A bound on the memory hardness of balloon hashing After the intuition that as the depth goes deeper, algorithm using DRAM fetching will certainly be advantageous, we are interested in finding a lower bound on the c_0 defined in definition 3. If we constrain the computational entity to only computing m adjacent blocks once at a time, then the problem would be trivial, as lemma 4 and simulation result in table 1 shows that as r gets greater, DRAM fetching will certainly be the optimal choice. Also, in shallower layers re-computation only takes up a small portion of total time (they cannot exceed $c \cdot \delta \cdot m$), then we have $c_0 > 1 - r_0/r$, where r_0 is the number of deepest layer in the group of shallow layers. Now the tricky part is that how to remove the constraint, so that forcing the algorithm to choose between these two strategies would be logical.

Consider an algorithm not using DRAM computing on a random sandwich graph, then there always exists a window of size m not computed ahead of the latest computed node (but for the last m nodes). And the computation can be considered to be computing the first node in the window. As when doing this, no nodes exist in the window, and the condition of such actions are somewhat identical. This give rise to the idea that we can consider the time required to compute each blocks in the subset V' of size m defined in lemma 3 to be identical. As the fetching strategy is scalable in nature, we can shrink m into 1 in this way. And thus any entity computing on this graph must choose between

[4]The authors of Balloon Hashing recommended the space parameter to be 256MB for authentication servers.

two strategies. And our conclusion that $c_0 > 1 - r_0/r$ suffices.

Conversely, if we can bound that for any optimal strategy computing the V' group of size m, there must exist a constant portion of blocks that are fetched from DRAM, then the memory hard property from definition 3 is satisfied.

Theorem 1. *Fix a random sandwich graph G with r layers and n nodes per layer, and given m and n_0 as defined in 3, the memory hardness parameter defined in definition 3 satisfies that*

$$c_0 > \frac{(m - \frac{n_0}{\delta}) \cdot \delta c}{\frac{n_0}{\delta} + (m - \frac{n_0}{\delta}) \cdot (\delta + 1)c} \tag{23}$$

Before we prove theorem 1, a little explanation here is necessary. The idea behind this is that when considering pebbling a group of size m haven't been pebbled, with large probability, only those nodes with parents fully pebbled can be advantageous in the case of "red pebble" solution. And indeed, if this is the case "red solution" would certainly be better than "blue solution". But due to the limitation on the total pebbles n_0, there can only exist a constant portion of such "red solutions", leaving the predecessor of the rest to be fetched from memory. And thus the memory hard property from definition 3 is satisfied.

The following lemma proves that if the predecessor of a node is partially pebbled, then using "red solution" is takes exponential time as the depth r goes further.

Lemma 5. *Let G be the random sandwich graph defined in theorem 1, if a node in layer r is not pebbled, and there exists at least one unpebbled parent of this node, then with probability at least:*

$$Pr = \frac{\binom{n_0-1}{n-m}}{\binom{n_0-1}{n-1}} \cdot \frac{n - m + 1}{n} \tag{24}$$

the moves needed to pebble this node is at least T_{r-1} as defined in definition 4.

Proof. Let x denote the unpebbled parent, and if at least m consecutively previous nodes are unpebbled, then to pebble x has the same effect as pebbling all those m nodes. Note that in this case we can choose arbitrary m, so long as the requirement from lemma 1 is satisfied. (if m is too big, the event would be unlikely) The rest could be considered as fix one bin, and drop n_0 balls into the rest $n-1$ bins. Note that we ignored the corner case where x resides in the first $m-1$ nodes of layer $r-1$. Other than the corner case, fix the position of x, and the probability is $\binom{n_0-1}{n-m}/\binom{n_0-1}{n-1}$. Taking all the probability in the corner case as zero would get the final result.

As shown, with high probability the previous m nodes of x is unpebbled. And thus we can use lemma 4 to deduce that to pebble x, at least T_{r-1} moves are required, which proves the lemma. □

Now that we see an optimal pebbling strategy would not waste pebbles on nodes with partially pebbled parents. And therefore the optimal strategy we are talking are left with two cases:

Red all parents of this node is fully pebbled, and it takes one round to pebble this node

Blue any parent of this node is not pebbled, and it takes $\delta c+1$ rounds to pebble this node

Given there are only n_0 red pebbles, one can easily calculate the portion of DRAM fetching moves used in pebbling these m nodes, and thus the memory hardness of the function.

Proof of theorem 1. From lemma 5 we can deduce that in a group of m adjacent nodes, n_0/δ of them are pebbled directly, and the rest is pebbled using the blue pebble moves. And thus the total rounds is

$$T = \frac{n_0}{\delta} + (m - \frac{n_0}{\delta}) \cdot (\delta + 1)c \tag{25}$$

where $(m - \frac{n_0}{\delta}) \cdot \delta c$ rounds are used to fetch blocks from DRAM. After r gets enough deep, the majority of the m adjacent nodes will satisfy the property. And this completes the proof. □

References

[1] Joël Alwen and Vladimir Serbinenko. High parallel complexity graphs and memory-hard functions. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 595–603. ACM, 2015.

[2] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[3] Henry Corrigan-Gibbs, Dan Boneh, and Stuart E Schechter. Balloon hashing: Provably space-hard hash functions with data-independent access patterns. *IACR Cryptology ePrint Archive*, 2016:27, 2016.

[4] Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. One-time computable self-erasing functions. In *Theory of Cryptography Conference*, pages 125–143. Springer, 2011.