An Improved Analysis On Balloon Hashing

Cui Hongrui, Sha Jinrui

February 1, 2018

1 Introduction

Hashing functions serve as the backbone in many modern cryptographic applications, such as crypto-currency and authentication of legitimate users on a shared host. Due to the invertible nature of cryptographic hashing function, the only way to get the message from its fingerprint is to mount a dictionary attack. Unfortunately, as the asymmetry between the computing ability of general purpose machine and application-specific hardware gets bigger[1] , applications that use traditional hashing functions may become either inefficient or insecure. In particular, an algorithm infeasible to be brute attacked on a general-purpose computer may be possible to be compromised on a piece of application-specific hardware. On the other hand, if the algorithm is hard enough against application-specific hardware, it would be too inefficient for legitimate users who would mostly use general purpose machines.

Memory Hard functions come to rescue Memory hard hashing functions overcome this disadvantage by using that fact that memory units on hardware that can match the speed of other components are costly in terms of space and energy, and on a particular chip one can not place too many memory units. An ideal memory hard hashing function can be executed efficiently with more than $|S|$ bits of space and if available space gets smaller than that amount, the performance drops dramatically. And as there are always sufficient memory (though slow) on general purpose machine, legitimate uses will not be affected.

Our Idea Many suggestions have been purposed, especially in the past password hashing competition. We discuss some of the results in section 1.1. Although some of these works have strictly proved the security (often in the form of time-space tradeoff) of their algorithms under sequential or parallel models, none of them have mentioned the possibility that an adversary may try to integrate the slow memories in their attacking scheme. In particular, these algorithms tend to achieve a state where the lack of memory will most certainly cause re-computation of previously calculated values, and such re-computation will take very long time. But consider if an adversary has cached all of the previous results needed in slow memory, then if re-computation takes longer time than a fetch cycle, it can use constant fetch time instead of re-compute the previous results.

[1] In [1], the author argues that ASIC has a 100,000 times advantage against x86 servers on energy cost in executing SHA-256 hashing function.

Our idea is to formalize this intuition and make a security (memory-hardness) definition under the new model. Then by making a simplified (but sufficient enough) analysis on balloon hashing, we can use the new model to test if balloon hashing could fit in this definition.

1.1 Related Works

There are a number of works focusing on designing and analysing memory hard hashing functions.

Although algorithms like balloon hashing has been proved to be secure under the sequential computation model, we find adapting this analysis in order to fit in our model is rather difficult. And thus we calculated a different bound on the average case from where the expected result can be easily drawn.

1.2 Our contribution

In this work, we introduced a new model to capture the DRAM ability of adversary and made a new analysis of balloon hashing that incorporate this model. In section 2 we briefly introduced the balloon hashing algorithm defined in section 3 in [1]. In section 3 we defined the model and some definitions that will be used throughout the analysis. In section 4 we did the analysis and showed that even if the ability of adversary is enlarged, the main portion of an optimal (takes the least time) execution of a balloon hashing instance will still spend most of its time on fetching blocks from memory.

2 Balloon Hashing Algorithm

The pseudo-code of balloon hashing can be found in [1]. We will present it here for sake of clarity.

Now we are ready to present our model and security definition in the next section.

3 Model And Definition

There are primarily two models in the analysis of memory-hard functions: the sequential model and the parallel model. In the sequential model only one request to the random oracle is allowed whereas in the parallel setting, multiple request can be handled at the same time. Also there are some difference in how to quantify the time-space tradeoff. The simpler one is to measure in terms of ST where S is the space used (roughly) and T is the time required. This is used in [1] and a number of other work. Alwen et al. purposed another metric *cumulative complexity* to capture the averaged result when multiple instances of the algorithm are running in parallel.

In this work we will stick to the sequential model. We present our model in the form of a set of pebbling rules. The rules are defined as follows.

3.1 Pebbling Rules

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 $T\prime \subset V$ 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*

Additional constraints are at any time, there can be only at most S red pebbles, and blue pebbles are removed automatically one move after placement. Move 1, 2, and 4 takes one round and move 3 takes c rounds where c is a constant equal to the ratio of the time of fetching a block from DRAM over the time needed to compute a block. A legal game is valid only if it begins with no node on the graph and end with all nodes in $T\prime$ 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 other work,

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.

3.2 Security Definition

In this section we give the security definition that will use in the proof section.

Definition 2. *We consider an algorithm memory hard if the fastest algorithm will take non-negligible ratio of total time to fetch block from memory. If we denote r to be this ratio, we have $\exists \delta > 0$ s.t. $\lim_{n \to \inf} r > \delta$.*

4 Recursive Analysis for the Execution Time

Previous work in this field typically use "pebbling reduction" to simplify the analysis. In essence, that means to analyse calculating strategy of the algorithm (function) , one only need to analyse the pebbling strategy of the underlying data-dependency graph. To do so, one need to prove that if a graph is hard to pebble, then its corresponding function is hard to compute, and vise versa. Luckily, it has been proved by Dwork, Naor, and Wee in [2]. As we have added a new rule and used a new security definition, we will modify the proof a little bit.

4.1 Proof of Pebbling Reduction

After proving that, we can move to the main section of the proof, the recursive reasoning.

4.2 Recursive Reasoning

Our goal here is to get a function f mapping from the layer (or depth) of the current target block to the red pebble placement moves needed to achieve this goal. We do this without allowing the adversary to use the blue pebble rules in order to prove that as the calculation goes deeper, slow memories will most certainly be used in the optimal strategy. If the graph has n nodes on each layer, and the number of red pebbles S is greater than n, then such function would be constant ($f = 1$).

The more practical case is when $S < n$. In this case, we enlarge the ability of the adversary by allowing them to have S red pebbles on each layer of the graph, but the adversary must not use them in a single layer because that would violate reality. If we consider the placement of red pebbles to be independent of the the predecessors (δ in total) needed to compute the next block, then the number of nodes that have red pebbles on them satisfies that:

$$X \sim H(\delta, n, S) \tag{1}$$

$$Pr[X = p] = \frac{C_S^p \cdot C_{n-S}^{\delta-p}}{C_n^S} \tag{2}$$

$$E(X) = \frac{\delta \cdot S}{n} \tag{3}$$

And this means that in the average case, only $E(X)$ of them are already available in memory, whereas the rest need to be re-computed. As we have allowed the adversary to own another S red pebbles in the previous layer, the only difference is the previous layer is one layer shallower than the current target. If we denote T_r as the amount of red pebble placement moves needed to compute the current target, then we have:

$$T_r = 1 + \frac{\delta \cdot S}{n} + (\delta - \frac{\delta \cdot S}{n}) \cdot T_{r-1} \tag{4}$$

Given the base case that on average, $T_0 = n/2$. Therefore we have

$$T_r = -\frac{1 + \frac{\delta \cdot S}{n}}{\delta - \frac{\delta \cdot S}{n} - 1} + (\delta - \frac{\delta \cdot S}{n})^r \cdot (\frac{n}{2} + \frac{1 + \frac{\delta \cdot S}{n}}{\delta - \frac{\delta \cdot S}{n} - 1}) \tag{5}$$

Analyse the result As result shown in equation 5, if we limit the pebbling strategy to using only the red pebbles (not rely on additional slow memory), the moves needed to compute an un-visited block grows exponentially with the depth of the block. Out of T_r, $\frac{\delta \cdot S}{n} \cdot T_{r-1}$ of them are spent on the re-computation. This will certainly not be the optimal choice as c is a constant, and a trivial means to use blue pebble moves in every nodes unpebbled would take $\frac{\delta \cdot S}{n} \cdot c$ moves. By the simple inequality:

$$\frac{\delta \cdot S}{n} \cdot T_{r-1} > \frac{\delta \cdot S}{n} \cdot c \tag{6}$$

we can easily get when using the trivial blue pebble strategy would become more advantageous.

4.3 Worst Case And Average Case

4.4 Balloon Hashing is Secure

In this section we prove that Balloon hashing does satisfy the security (memory hard) definition defined in definition 2.

Lemma 1. *Set Parameter c, the Balloon Hashing algorithm defined in section 2 is secure (memory hard) according to definition 2.*

Proof. What to do here??? □

5 Now What?

References

[1] 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.

[2] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In *Annual International Cryptology Conference*, pages 37–54. Springer, 2005.