Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks

Dan Boneh[1], Henry Corrigan-Gibbs[1], and Stuart Schechter[2]

[1] Stanford University, Stanford CA 94305, U.S.A.
[2] Microsoft Research, Redmond WA 98052, U.S.A.

Abstract. We present the Balloon password-hashing algorithm. This is the first practical cryptographic hash function that: (i) has proven memory-hardness properties in the random-oracle model, (ii) uses a password-independent access pattern, and (iii) meets—and often exceeds—the performance of the best heuristically secure password-hashing algorithms. Memory-hard functions require a large amount of working space to evaluate efficiently and, when used for password hashing, they dramatically increase the cost of offline dictionary attacks. In this work, we leverage a previously unstudied property of a certain class of graphs ("random sandwich graphs") to analyze the memory-hardness of the Balloon algorithm. The techniques we develop are general: we also use them to give a proof of security of the scrypt and Argon2i password-hashing functions, in the random-oracle model. Our security analysis uses a *sequential* model of computation, which essentially captures attacks that run on single-core machines. Recent work shows how to use massively parallel special-purpose machines (e.g., with hundreds of cores) to attack memory-hard functions, including Balloon. We discuss this important class of attacks, which is outside of our adversary model, and propose practical defenses against them. To motivate the need for security proofs in the area of password hashing, we demonstrate and implement a practical attack against Argon2i that successfully evaluates the function with less space than was previously claimed possible. Finally, we use experimental results to compare the performance of the Balloon hashing algorithm to other memory-hard functions.

Keywords: memory-hard functions, password hashing, pebbling arguments, time-space trade-offs, sandwich graph, Argon2, scrypt.

1 Introduction

The staggering number of password files breaches in recent months demonstrates the importance of cryptographic protection for stored passwords. In 2015 alone, attackers stole files containing users' login names, password hashes, and contact information from many large and well-resourced organizations, including LastPass [89], Harvard [53], E*Trade [70], ICANN [51], Costco [47], T-Mobile [86], the University of Virginia [84], and a large number of others [74]. In this environment, systems administrators must operate under the assumption that attackers will eventually gain access to sensitive authentication information, such as password hashes and salts, stored on their computer systems. After a compromise, the secrecy of user passwords rests on the cost to an attacker of mounting an offline dictionary attack against the stolen file of hashed passwords.

An ideal password-hashing function has the property that it costs as much for an attacker to compute the function as it does for the legitimate authentication server to compute it. Standard cryptographic hashes completely fail in this regard: it takes 100 000× more energy to compute a SHA-256 hash on a general-purpose x86 CPU (as an authentication server would use) than it does to compute SHA-256 on special-purpose hardware (such as the ASICs that an attacker would use) [23]. Iterating a standard cryptographic hash function, as is done in bcrypt [75] and PBKDF2 [49], increases the absolute cost to the attacker and defender, but the attacker's 100 000× relative cost advantage remains.

Memory-hard functions help close the efficiency gap between the attacker and defender in the setting of password hashing [8, 19, 42, 64, 68]. Memory-hard functions exploit the observation that on-chip memory is just as costly to power on special-purpose hardware as it is on a general-purpose CPU. If evaluating the password-hashing function requires large amounts of memory, then an attacker using special-purpose hardware has little cost advantage over the legitimate authentication server (using a standard x86 machine, for example) at running the password-hashing computation. Memory consumes a large amount of on-chip area, so the high memory requirement ensures that a special-purpose chip can only contain a small number of hashing engines.

An optimal memory-hard function, with security parameter n, has a space-time product that satisfies $S \cdot T \in \Omega(n^2)$, irrespective of the strategy used to compute the function [68]. The challenge is to construct a function that *provably* satisfies this bound with the largest possible constant multiple on the n^2 term.

In this paper, we introduce the Balloon memory-hard function for password hashing. This is the first practical password-hashing function to simultaneously satisfy three important design goals [64]:

– *Proven memory-hard.* We prove, in the random-oracle model [14], that computing the Balloon function with space S and time T requires $S \cdot T \geq n^2/8$ (approximately). As the adversary's space usage decreases, we prove even sharper time-space lower bounds.

  To motivate our interest in memory-hardness proofs, we demonstrate in Section 4 an attack against the Argon2i password hashing function [19], winner of a recent password-hashing design competition [64]. The attack evaluates the function with far less space than claimed without changing the time required to compute the function. We also give a proof of security for Argon2i in the random-oracle model, which demonstrates that significantly more powerful attacks against Argon2i are impossible under our adversary model.

- *Password-independent memory-access pattern.* The memory-access pattern of the Balloon algorithm is *independent* of the password being hashed. Password-hashing functions that lack this property are vulnerable to a crippling attack in the face of an adversary who learns the memory-access patterns of the hashing computation, e.g., via cache side-channels [25, 62, 87]. The attack, which we describe in Appendix G.2, makes it possible to run a dictionary attack with very little memory. A hashing function with a password-independent memory-access pattern eliminates this threat.
- *Performant.* The Balloon algorithm is easy to implement and it matches or exceeds the performance of the fastest comparable password-hashing algorithms, Argon2i [19] and Catena [42], when instantiated with standard cryptographic primitives (Section 6).

We analyze the memory-hardness properties of the Balloon function using pebble games, which are arguments about the structure of the data-dependency graph of the underlying computation [54, 65, 67, 81, 85]. Our analysis uses the framework of Dwork, Naor, and Wee [37]—later applied in a number of cryptographic works [6, 8, 38, 39, 42]—to relate the hardness of pebble games to the hardness of certain computations in the random-oracle model [14].

The crux of our analysis is a new observation about the properties of "random sandwich graphs," a class of graphs studied in prior work on pebbling [6, 8]. To show that our techniques are broadly applicable, we apply them in Appendices F and G to give simple proofs of memory-hardness, in the random-oracle model, for the Argon2i and scrypt functions. We prove stronger memory-hardness results about the Balloon algorithm, but these auxiliary results about Argon2i and scrypt may be of independent interest to the community.

The performance of the Balloon hashing algorithm is surprisingly good, given that our algorithm offers stronger proven security properties than other practical memory-hard functions with a password-independent memory access patterns. For example, if we configure Balloon to use Blake2b as the underlying hash function [10], run the construction for five "rounds" of hashing, and set the space parameter to require the attacker to use 1 MiB of working space to compute the function, then we can compute Balloon Hashes at the rate of 13 hashes per second on a modern server, compared with 12.8 for Argon2i, and 2.1 for Catena DBG (when Argon2i and Catena DBG are instantiated with Blake2b as the underlying cryptographic hash function).[3]

Caveat: Parallel Attacks. The definition of memory-hardness we use puts a lower-bound on the time-space product of computing a *single* instance of the Balloon function on a sequential (single-core) computer. In reality, an adversary mounting a dictionary attack would want to compute *billions* of instances of the Balloon function, perhaps using many processors running in parallel. Alwen and Serbinenko [8], formalizing earlier work by Percival [68], introduce a new com-

[3] The relatively poor performance of Argon2i here is due to the attack we present in Section 4. It allows an attacker to save space in computing Argon2i with no increase in computation time.

putational model—the parallel random-oracle model (pROM)—and a memory-hardness criterion that addresses the shortcomings of the traditional model. In recent work, Alwen and Blocki prove the surprising result that *no function* that uses a password-independent memory access pattern can be optimally memory-hard in the pROM [3]. In addition, they give a special-purpose pROM algorithm for computing Argon2i, Balloon, and other practical (sequential) memory-hard functions with some space savings. We discuss this important class of attacks and the relevant related work in Section 5.1.

Contributions. In this paper, we

- introduce and analyze the Balloon hashing function, which has stronger provable security guarantees than prior practical memory-hard functions (Section 3),
- present a practical memory-saving attack against the Argon2i password-hashing algorithm (Section 4), and
- explain how to ameliorate the danger of massively parallel attacks against memory-hard functions with a password-independent access pattern (Section 5.1)
- prove the first known time-space lower bounds for Argon2i and an idealized variant of scrypt, in the random-oracle model. (See Appendices F and G.)

With the Balloon algorithm, we demonstrate that it is possible to provide provable protection against a wide class of attacks without sacrificing performance.

Notation. Throughout this paper, Greek symbols (α, β, γ, λ, etc.) typically denote constants greater than one. We use $\log_2(\cdot)$ to denote a base-two logarithm and $\log(\cdot)$ to denote a logarithm when the base is not important. For a finite set S, the notation $x \overset{\text{R}}{\leftarrow} S$ indicates sampling an element of S uniformly at random and assigning it to the variable x.

2 Security Definitions

This section summarizes the high-level security and functionality goals of a password hashing function in general and the Balloon hashing algorithm in particular. We draw these aims from prior work on password hashing [68, 75] and also from the requirements of the recent Password Hashing Competition [64].

2.1 Syntax

The Balloon password hashing algorithm takes four inputs: a password, salt, time parameter, and space parameter. The output is a bitstring of fixed length (e.g., 256 or 512 bits). The password and salt are standard [59], but we elaborate on the role of the latter parameters below.

Space Parameter (Buffer Size). The space parameter, which we denote as "n" throughout, indicates how many fixed-size blocks of working space the hash function will require during the course of its computation, as in scrypt [68]. At a high level, a memory-hard function should be "easy" to compute with n blocks of working space and should be "hard" to compute with much less space than that. We make this notion precise later on.

Time Parameter (Number of Rounds). The Balloon function takes as input a parameter r that determines the number of "rounds" of computation it performs. As in bcrypt [75], the larger the time parameter, the longer the hash computation will take. On memory-limited platforms, a system administrator can increase the number of rounds of hashing to increase the cost of computing the function without increasing the algorithm's memory requirement. The choice of r has an effect on the memory-hardness properties of the scheme: the larger r is, the longer it takes to compute the function in small space.

2.2 Memory-Hardness

We say that a function f_n on space parameter n is *memory-hard* in the (sequential) random-oracle model [14] if, for all adversaries computing f_n with high probability using space S and T random oracle queries, we have that $S \cdot T \in \Omega(n^2)$. This definition deserves a bit of elaboration. Following Dziembowski et al. [39] we say that an algorithm "uses space S" if the entire configuration of the Turing Machine (or RAM machine) computing the algorithm requires at least S bits to describe. When, we say that an algorithm computes a function "with high probability," we mean that the probability that the algorithm computes the function is non-negligible as the output size of the random oracle and the space parameter n tend to infinity. In practice, we care about the adversary's concrete success probability, so we avoid asymptotic notions of security wherever possible. In addition, as we discuss in the evaluation section (Section 6), the exact value of the constant hidden inside the $\Omega(\cdot)$ is important for practical purposes, so our analysis makes explicit and optimizes these constants.

A function that is memory-hard under this definition requires the adversary to use either a lot of working space or a lot of execution time to compute the function. Functions that are memory-hard in this way are not amenable to implementation in special-purpose hardware (ASIC), since the cost to power a unit of memory for a unit of time on an ASIC is the same as the cost on a commodity server. An important limitation of this definition is that it does not take into account parallel or multiple-instance attacks, which we discuss in Section 5.1.

2.3 Password-Independent Access Pattern

A first-class design goal of the Balloon algorithm is to have a memory access pattern that is *independent* of the password being hashed. (We allow the data-access pattern to depend on the salt, since the salts can be public.) As mentioned above, employing a password-independent access pattern reduces the risk that information about the password will leak to other users on the same machine via cache or other side-channels [25, 62, 87]. This may be especially important in cloud-computing environments, in which many mutually distrustful users share a single physical host [78].

Creating a memory-hard function with a password-independent access pattern presents a technical challenge: since the data-access pattern depends only upon the salt—which an adversary who steals the password file knows—the adversary can compute the entire access pattern in advance of a password-guessing attack. With the access pattern in hand, the adversary can expend a huge amount of effort to find an efficient strategy for computing the hash function in small space. Although this pre-computation might be expensive, the adversary can amortize its cost over billions of subsequent hash evaluations. A function that is memory-hard *and* that uses a password-independent data access pattern must be impervious to *all* small-space strategies for computing the function so that it maintains its strength in the face of these pre-computation attacks. (Indeed, as we discuss in Section 5.1, Alwen and Blocki show that in some models of computation, memory-hard functions with password-independent access patterns do not exist [3].)

2.4 Collision Resistance, etc.

If necessary, we can modify the Balloon function so that it provides the standard properties of second-preimage resistance and collision resistance [57]. It is possible to achieve these properties in a straightforward way by composing the Balloon function B with a standard cryptographic hash function H as

$$H_B(\mathrm{passwd}, \mathrm{salt}) := H(\mathrm{passwd}, \mathrm{salt}, B(\mathrm{passwd}, \mathrm{salt})).$$

Now, for example, if H is collision-resistant, then H_B must also be.[4] That is because any inputs $(x_p, x_s) \neq (y_p, y_s)$ to H_B that cause $H_B(x_p, x_s) = H_B(y_p, y_s)$ immediately yield a collision for H as:

$$(x_p, x_s, B(x_p, x_s)) \qquad \text{and} \qquad (y_p, y_s, B(y_p, y_s)),$$

no matter how the Balloon function B behaves.

3 Balloon Hashing Algorithm

In this section, we present the Balloon hashing algorithm.

3.1 Algorithm

The algorithm uses a standard (non-memory-hard) cryptographic hash function $H : \mathbb{Z}_N \times \{0,1\}^{2k} \to \{0,1\}^k$ as a subroutine, where N is a large integer. For the purposes of our analysis, we model the function H as a random oracle [14].

[4] We are eliding important definitional questions about what it even means, in a formal sense, for a function to be collision resistant [17, 79].

```
func Balloon(block_t passwd, block_t salt,
    int s_cost,         // Space cost (main buffer size)
    int t_cost):        // Time cost (number of rounds)
  int delta = 3         // Number of dependencies per block
  int cnt = 0           // A counter (used in security proof)
  block_t buf[s_cost]):   // The main buffer

  // Step 1. Expand input into buffer.
  buf[0] = hash(cnt++, passwd, salt)
  for m from 1 to s_cost-1:
    buf[m] = hash(cnt++, buf[m-1])

  // Step 2. Mix buffer contents.
  for t from 0 to t_cost-1:
    for m from 0 to s_cost-1:
      // Step 2a. Hash last and current blocks.
      block_t prev = buf[(m-1) mod s_cost]
      buf[m] = hash(cnt++, prev, buf[m])

      // Step 2b. Hash in pseudorandomly chosen blocks.
      for i from 0 to delta-1:
        block_t idx_block = ints_to_block(t, m, i)
        int other = to_int(hash(cnt++, salt, idx_block)) mod s_cost
        buf[m] = hash(cnt++, buf[m], buf[other])

  // Step 3. Extract output from buffer.
  return buf[s_cost-1]
```

Fig. 1: Pseudo-code of the Balloon hashing algorithm.

The Balloon algorithm uses a large memory buffer as working space and we divide this buffer into contiguous *blocks*. The size of each block is equal to the output size of the hash function H. Our analysis is agnostic to the choice of hash function, except that, to prevent pitfalls described in Appendix B.3, the internal state size of H must be at least as large as its output size. Since H maps blocks of $2k$ bits down to blocks of k bits, we sometimes refer to H as a *cryptographic compression function.*

The Balloon function operates in three steps (Figure 1):

1. **Expand.** In the first step, the Balloon algorithm fills up a large buffer with pseudo-random bytes derived from the password and salt by repeatedly invoking the compression function H on a function of the password and salt.
2. **Mix.** In the second step, the Balloon algorithm performs a "mixing" operation r times on the pseudo-random bytes in the memory buffer. The user-specified round parameter r determines how many rounds of mixing take place. At each mixing step, for each block i in the buffer, the routine updates the contents of block i to be equal to the hash of block $(i-1) \bmod n$,

block i, and δ other blocks chosen "at random" from the buffer. (See Theorem 1 for an illustration of how the choice of δ affects the security of the scheme.)
Since the Balloon functions are deterministic functions of their arguments, the dependencies are not chosen truly at random but are sampled using a pseudorandom stream of bits generated from the user-specific salt.

3. **Extract.** In the last step, the Balloon algorithm outputs the last block of the buffer.

Multi-Core Machines. A limitation of the Balloon algorithm as described is that it does not allow even limited parallelism, since the value of the ith block computed always depends on the value of the $(i-1)$th block. To increase the rate at which the Balloon algorithm can fill memory on a multi-core machine with M cores, we can define a function that invokes the Balloon function M times in parallel and XORs all the outputs. If $\mathsf{Balloon}(p, s)$ denotes the Balloon function on password p and salt s, then we can define an M-core variant $\mathsf{Balloon}_M(p, s)$ as:

$$\mathsf{Balloon}_M(p, s) := \mathsf{Balloon}(p,\ s\|\text{“}1\text{”}) \oplus \cdots \oplus \mathsf{Balloon}(p,\ s\|\text{“}M\text{”}).$$

A straightforward argument shows that computing this function requires computing M instances of the single-core Balloon function. Existing password hashing functions deploy similar techniques on multi-core platforms [19, 42, 68, 69].

3.2 Main Security Theorem

The following theorem demonstrates that attackers who attempt to compute the Balloon function in small space must pay a large penalty in computation time. The complete theorem statement is given in Theorem 33 (Appendix E).

Theorem 1 (informal) *Let $\mathcal{A}$ be an algorithm that computes the n-block r-round Balloon function with security parameter $\delta \geq 3$, where H is modeled as a random oracle. If $\mathcal{A}$ uses at most S blocks of buffer space then, with overwhelming probability, $\mathcal{A}$ must run for time (approximately) T, such that*

$$S \cdot T \geq \frac{r \cdot n^2}{8}.$$

Moreover, under the stated conditions, one obtains the stronger bound:

$$S \cdot T \geq \frac{(2^r - 1)n^2}{8} \qquad \text{if} \qquad \begin{cases} \delta = 3 \text{ and } S < n/64 \text{ or,} \\ \delta = 4 \text{ and } S < n/32 \text{ or,} \\ \delta = 5 \text{ and } S < n/16 \text{ or,} \\ \delta = 7 \text{ and } S < n/8. \end{cases}$$

The theorem shows that, when the adversary's space usage falls below a certain threshold (parameterized by δ), the computation time increases exponentially in the number of rounds r. For example, when $\delta = 7$ and the space S

is less than $n/8$, the time to evaluate Balloon is at least 2^r times the time to evaluate it with space n. Thus, attackers who attempt to compute the Balloon function in very small space must pay a large penalty in computation time.

The proof of the theorem is given in the appendices: Appendix B introduces graph pebbling arguments, which are the basis for our memory-hardness proofs. Appendix C proves a key combinatorial lemma required for analysis of the Balloon functions. Appendix D recalls *sandwich graphs* and proves that stacks of random sandwich graphs are hard to pebble. Appendix E puts all of the pieces together to complete the proof of Theorem 1.

Here we sketch the main ideas in the proof of Theorem 1.

```
v_1 = hash(input, "0")
v_2 = hash(input, "1")
v_3 = hash(v_1, v_2)
v_4 = hash(v_2, v_3)
v_5 = hash(v_3, v_4)
return v_5
```

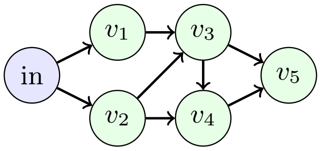


Fig. 2: An example computation (left) and its corresponding data-dependency graph (right).

Proof idea. The proof makes use of *pebbling arguments*, a classic technique for analyzing computational time-space trade-offs [48, 54, 67, 71, 81, 88] and memory-hard functions [8, 37, 38, 42]. We apply pebbling arguments to the *data-dependency graph* corresponding to the computation of the Balloon function (See Figure 2 for an example graph). The graph contains a vertex for every random oracle query made during the computation of Balloon: vertex v_i in the graph represents the response to the ith random-oracle query. An edge (v_i, v_j) indicates that the input to the jth random-oracle query depends on the response of the ith random-oracle query.

The data-dependency graph for a Balloon computation naturally separates into $r + 1$ layers—one for each round of mixing (Figure 3). That is, a vertex on level $\ell \in \{1, \dots, r\}$ of the graph represents the output of a random-oracle query made during the ℓth mixing round.

The first step in the proof shows that the data-dependency graph of a Balloon computation satisfies certain connectivity properties, defined below, with high probability. The probability is taken over the choice of random oracle H, which determines the data-dependency graph. Consider placing a pebble on each of a subset of the vertices of the data-dependency graph of a Balloon computation. Then, as long as there are "not too many" pebbles on the graph, we show that the following two properties hold with high probability:

- *Well-Spreadedness.* For every set of k consecutive vertices on some level of the graph, at least a quarter of the vertices on the prior level of the graph are on unpebbled paths to these k vertices (Lemma 28).

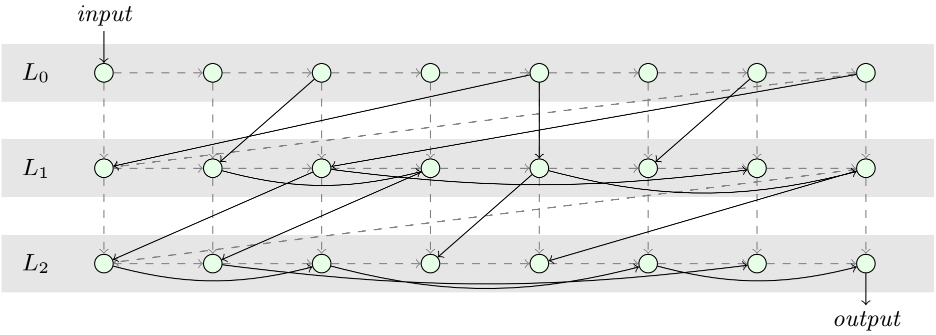


Fig. 3: The Balloon data-dependency graph on $n = 8$ blocks and $r = 2$ rounds, drawn with $\delta = 1$ for simplicity. (The real construction uses $\delta \geq 3$.) The dashed edges are fixed and the solid edges are chosen pseudorandomly by applying the random oracle to the salt.

- *Expansion.* All sets of k vertices on any level of the graph have unpebbled paths back to at least $2k$ vertices on the prior level(Lemma 29). The value of k depends on the choice of the parameter δ.

The next step is to show that every *graph-respecting* algorithm computing the Balloon function requires large space or time. We say that an adversary $\mathcal{A}$ is graph respecting if for every i, adversary $\mathcal{A}$ makes query number i to the random-oracle only after it has in storage all of the values that this query takes as input.[5]

We show, using the well-spreadedness and expansion properties of the Balloon data-dependency graph, that every graph-respecting adversary $\mathcal{A}$ must use space S and time T satisfying $S \cdot T \geq n^2/8$, with high probability over the choice of H (Lemma 31). We use the graph structure in the proof as follows: fix a set of k values that the adversary has not yet computed. Then the graph properties imply that these k values have many dependencies that a space-S adversary cannot have in storage. Thus, making progress towards computing the Balloon function in small space requires the adversary to undertake a huge amount of recomputation.

The final step uses a technique of Dwork, Naor, and Wee [37]. They use the notion of a graph *labeling* to convert a directed acyclic graph G into a function f_G (Definition 3). They prove that if G is a graph that is infeasible for time-T space-S graph-respecting pebbling adversaries to compute, then it is infeasible for time-T' space-S' arbitrary adversaries to compute the labeling function f_G,

[5] This description is intentionally informal—see Appendix B for the precise statement.

with high probability in the random-oracle model (Theorem 4), where $T' \approx T$ and $S' \approx S$.

We observe that Balloon computes the function f_G where G is the Balloon data-dependency graph (Claim 32). We then directly apply the technique of Dwork, Naor, and Wee to obtain a upper bound on the probability that an arbitrary adversary can compute the Balloon function in small time and space (Theorem 33). □

4 Attacking and Defending Argon2

In this section, we analyze the Argon2i password hashing function [19], which won the recent Password Hashing Competition [64].

An Attack. We first present an attack showing that it possible for an attacker to compute multi-pass Argon2i (the recommended version) saving a factor of $e \approx 2.72$ in space with *no increase in computation time.*[6] Additionally, we show that an attacker can compute the single-pass variant of Argon2i, which is also described in the specification, saving more than a factor of four in space, again *with no increase in computation time.* These attacks demonstrate an unexpected weakness in the Argon2i design, and show the value of a formal security analysis.

A Defense. In Appendix F, we give the first proof of security showing that, with high probability, single-pass n-block Argon2i requires space S and time T to compute, such that $S \cdot T \geq n^2/192$, in the sequential random-oracle model. Our proof is relatively simple and uses the same techniques we have developed to reason about the Balloon algorithm. The time-space lower bound we can prove about Argon2i is weaker than the one we can prove about Balloon, since the Argon2i result leaves open the possibility of an attack that saves a factor of 192 factor in space with no increase in computation time. If Argon2i becomes a standard algorithm for password hashing, it would be a worthwhile exercise to try to improve the constants on both the attacks and lower bounds to get a clearer picture of its exact memory-hardness properties.

4.1 Attack Overview

Our Argon2i attacks require a linear-time pre-computation operation that is independent of the password and salt. The attacker need only run the pre-computation phase once for a given choice of the Argon2i public parameters (buffer size, round count, etc.). After running the pre-computation step once, it is possible to compute many Argon2i password hashes, on different salts and different passwords using our small-space computation strategy. Thus, the cost of the pre-computation is amortized over many subsequent hash computations.

[6] We have notified the Argon2i designers of this attack and the latest version of the specification incorporates a design change that attempts to prevent the attack [21]. We describe the attack on the original Argon2i design, the winner of the password hashing competition [64].

The attacks we demonstrate undermine the security claims of the Argon2i (version 1.2.1) design documents [19]. The design documents claim that computing n-block single-pass Argon2i with $n/4$ space incurs a 7.3× computational penalty [19, Table 2]. Our attacks show that there is no computational penalty. The design documents claim that computing n-block three-pass Argon2i with $n/3$ space incurs a $16,384\times$ computational penalty [19, Section 5.4]. We compute the function in $n/2.7 \approx n/3$ space with no computational penalty.

We analyze a idealized version the Argon2i algorithm, which is slightly simpler than that proposed in the Argon2i v1.2.1 specification [19]. Our idealized analysis *underestimates* the efficacy of our small-space computation strategy, so the strategy we propose is actually *more effective* at computing Argon2i than the analysis suggests. The idealized analysis yields an expected $n/4$ storage cost, but as Figure 4 demonstrates, empirically our strategy allows computing single-pass Argon2i with only $n/5$ blocks of storage. This analysis focuses on the single-threaded instantiation of Argon2i—we have not tried to extend it to the many-threaded variant.

4.2 Background on Argon.

At a high level, the Argon2i hashing scheme operates by filling up an n-block buffer with pseudo-random bytes, one 1024-byte block at a time. The first two blocks are derived from the password and salt. For $i \in \{3, \ldots, n\}$, the block at index i is derived from two blocks: the block at index $(i-1)$ and a block selected pseudo-randomly from the set of blocks generated so far. If we denote the contents of block i as x_i, then Argon2i operates as follows:

$$\begin{aligned} x_1 &= H(\text{passwd},\ \text{salt} \parallel 1) \\ x_2 &= H(\text{passwd},\ \text{salt} \parallel 2) \\ x_i &= H(x_{i-1}, x_{r_i}) \quad \text{where } r_i \in \{1, \ldots, i-1\} \end{aligned}$$

Here, H is a non-memory-hard cryptographic hash function mapping two blocks into one block. The random index r_i is sampled from a non-uniform distribution over $S_i = \{1, \ldots, i-1\}$ that has a heavy bias towards blocks with larger indices. We model the index value r_i as if it were sampled from the uniform distribution over S_i. Our small-space computation strategy performs *better* under a distribution biased towards larger indices, so our analysis is actually somewhat conservative.

The single-pass variant of Argon2i computes $(x_1, \ldots, x_n)$ in sequence and outputs bytes derived from the last block x_n. Computing the function in the straightforward way requires storing every generated block for the duration of the computation—n blocks total

The multiple-pass variant of Argon2i works as above except that it computes pn blocks instead of just n blocks, where p is a user-specified integer indicating the number of "passes" over the memory the algorithm takes. (The number of passes in Argon2i is analogous to number of rounds r in Balloon hashing.) The default number of passes is three. In multiple-pass Argon2i, the contents of block

i are derived from the prior block and one of the most recent n blocks. The output of the function is derived from the value x_{pn}. When computing the multiple-pass variant of Argon2i, one need only store the latest n blocks computed (since earlier blocks will never be referenced again), so the storage cost of the straightforward algorithm is still roughly n blocks.

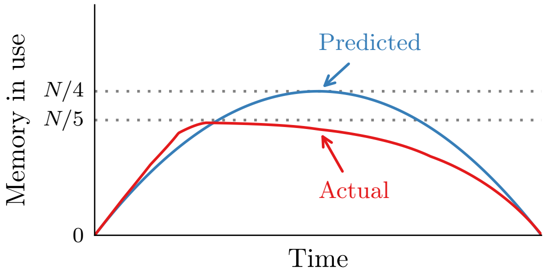


Fig. 4: Space used by our algorithm for computing single-pass Argon2i during a single hash computation.

Our analysis splits the Argon2i computation into discrete time steps, where time step t begins at the moment at which the algorithm invokes the compression function H for the tth time.

4.3 Attack Algorithm

Our strategy for computing p-pass Argon2i with fewer than n blocks of memory is as follows:

- **Pre-computation Phase.** We run the entire hash computation once—on an arbitrary password and salt—and write the memory access pattern to disk. For each memory block i, we pre-compute the time t_i after which block i is never again accessed and we store $\{t_1, \ldots, t_{pn}\}$ in a read-only array. The total size of this table on a 64-bit machine is at most $8pn$ bytes.[7]

  Since the Argon2i memory-access pattern does not depend on the password or salt, it is possible to use this same pre-computed table for many subsequent Argon2i hash computations (on different salts and passwords).

[7] On an FPGA or ASIC, this table can be stored in relatively cheap shared read-only memory and the storage cost can be amortized over a number of compute cores. Even on a general-purpose CPU, the table and memory buffer for the single-pass construction together will only require $8n + 1024(n/4) = 8n + 256n$ bytes when using our small-space computation strategy. Argon2i normally requires $1024n$ bytes of buffer space, so our strategy still yields a significant space savings.

- **Computation Phase.** We compute the hash function as usual, except that we delete blocks that will never be accessed again. After reading block i during the hash computation at time step t, we check whether the current time $t \geq t_i$. If so, we delete block i from memory and reuse the space for a new block.

The expected space required to compute n-block single-pass Argon2i is $n/4$. The expected space required to compute n-block many-pass Argon2i tends to $n/e \approx 2.7$ as the number of passes tends to infinity. We analyze the space usage of the attack algorithm in detail in Appendix A.

5 Discussion

In this section, we discuss parallel attacks against memory-hard functions and compare Balloon to other candidate password-hashing functions.

5.1 Memory Hardness Under Parallel Attacks

The Balloon Hashing algorithm achieves the notion of memory-hardness introduced in Section 2.2: an algorithm for computing Balloon must, with high probability in the random-oracle model, use (roughly) time T and space S that satisfy $S \cdot T \in \Omega(n^2)$. Using the time-space product in this way as a proxy metric for computation cost is natural, since it approximates the area-time product required to compute the function in hardware [68].

As Alwen and Serbinenko [8] point out, there are two key limitations to the standard definition of memory-hardness in which we prove security. First, the definition yields a *single-instance* notion of security. That is, our definition of memory-hardness puts a lower-bound on the ST cost of computing the Balloon function once, whereas in a password-guessing attack, the adversary potentially wants to compute the Balloon function *billions of times.*[8] Second, the definition treats a *sequential* model of computation—in which the adversary can make a single random-oracle query at each time step. In contrast, a password-guessing adversary may have access to thousands of computational cores operating *in parallel.*

To address the limitations of the conventional single-instance sequential adversary model, which we use for our analysis of the Balloon function, Alwen and Serbinenko introduce a new adversary model and security definition. Essentially, they allow the adversary to make many parallel random-oracle queries at each time step. In this “parallel random-oracle model” (pROM), they attempt to put a

[8] Bellare, Ristenpart, and Tessaro consider a different type of multi-instance security [13]: they are interested in key-derivation functions f with the property that finding $(x_1, \ldots, x_m)$ given $(f(x_1), \ldots, f(x_m))$ is roughly m times as costly as inverting f once. Stebila et al. [83] and Groza and Warinschi [46] investigate a similar multiple-instance notion of security for client puzzles [36] and Garay et al. [43] investigate related notions in the context of multi-party computation.

lower bound on the sum of the adversary's space usage over time: $\sum_t S_t \in \Omega(n^2)$, where S_t is the number of blocks of space used in the t-th computation step. We call a function that satisfies this notion of memory-hardness in the pROM an *amortized memory-hard* function.[9]

To phrase the definition in different terms: Alwen and Serbinenko look for functions f such that computing f requires a large amount of working space *at many points* during the computation of f. In contrast, the traditional definition (which we use) proves the weaker statement that the adversary computing f must use a lot of space *at some point* during the computation of f.

An impressive recent line of work has uncovered many new results in this model:

- Alwen and Blocki [3, 4] show that, in the pROM, there *does not exist* a perfectly memory-hard function (in the amortized sense) that uses a password-independent memory-access pattern. In the sequential setting, Balloon and other memory-hard functions require space S and time T to compute such that $S \cdot T \in \Omega(n^2)$. In the parallel setting, Alwen and Blocki show that the best one can hope for, in terms of amortized space usage is $\Omega(n^2/\log n)$. Additionally, they give special-case attack algorithms for computing many candidate password-hashing algorithms in the pROM. Their algorithm computes Balloon, for example, using an amortized time-space product of roughly $O(n^{7/4})$.[10]
- Alwen et al. [6] show that the amortized space-time complexity of the single-round Balloon function is at least $\tilde{\Omega}(n^{5/3})$, where the $\tilde{\Omega}(\cdot)$ ignores logarithmic factors of n. This result puts a limit on the effectiveness of parallel attacks against Balloon.
- Alwen et al. [5] construct a memory-hard function with a password-independent access pattern and that has an asymptotically optimal amortized time-space product of $S \cdot T \in \Omega(n^2/\log n)$. Whether this construction is useful for practical purposes will depend heavily on value of the constant hidden in the $\Omega(\cdot)$. In practice, a large constant may overwhelm the asymptotic improvement.
- Alwen et al. [6] prove, under combinatorial conjectures[11] that scrypt *is* near-optimally memory-hard in the pROM. Unlike Balloon, scrypt uses a data-dependent access pattern—which we would like to avoid—and the data-dependence of scrypt's access pattern seems fundamental to their security analysis.

As far as practical constructions go, these results leave the practitioner with two options, each of which has a downside:

[9] In the original scrypt paper, Percival [68] also discusses parallel attacks and makes an argument for the security of scrypt in the pROM.

[10] There is no consensus on whether it would be feasible to implement this parallel attack in hardware for realistic parameter sizes. That said, the fact that such pROM attacks exist at all are absolutely a practical concern.

[11] A recent addendum to the paper suggests that the combinatorial conjectures that underlie their proof of security may be false [7, Section 0].

Option 1. Use scrypt, which seems to protect against parallel attacks, but which uses a password-dependent access pattern and is weak in the face of an adversary that can learn memory access information. (We describe the attack in Appendix G.2.)

Option 2. Use Balloon Hashing, which uses a password-independent access pattern and is secure against sequential attacks, but which is asymptotically weak in the face of a massively parallel attack.

A good practical solution is to hash passwords using a careful composition of Balloon and scrypt: one function defends against memory access pattern leakage and the other defends against massively parallel attacks. For the moment, let us stipulate that the pROM attacks on vanilla Balloon (and all other practical password hashing algorithms using data-independent access patterns) make these algorithms less-than-ideal to use on their own. Can we somehow combine the two constructions to get a "best-of-both-worlds" practical password-hashing algorithm? The answer is yes: compose a data-independent password-hashing algorithm, such as Balloon, with a data-dependent scheme, such as scrypt. To use the composed scheme, one would first run the password through the data-independent algorithm and next run the resulting hash through the data-dependent algorithm.[12]

It is not difficult to show that the composed scheme is memory-hard against either: (a) an attacker who is able to learn the function's data-access pattern on the target password, or (b) an attacker who mounts an attack in the pROM using the parallel algorithm of Alwen and Blocki [3]. The composed scheme defends against the two attacks separately but does not defend against both of them simultaneously: the composed function does not maintain memory-hardness in the face of an attacker who is powerful enough to get access-pattern information *and* mount a massively parallel attack. It would be even better to have a practical construction that could protect against both attacks simultaneously, but the best known algorithms that do this [5, 8] are likely too inefficient to use in practice.

The composed function is almost as fast as Balloon on its own—adding the data-dependent hashing function call is effectively as costly as increasing the round count of the Balloon algorithm by one.

5.2 How to Compare Memory-Hard Functions

If we restrict ourselves to considering memory-hard functions in the sequential setting, there are a number of candidate constructions that all can be proven secure in the random-oracle model: Argon2i [21],[13] Catena BRG, Catena DBG [42], and Balloon. There is no widely accepted metric with which one measures the

[12] Our argument here gives some theoretical justification for the the Argon2id mode of operation proposed in some versions of the Argon2 specification [21, Appendix B]. That variant follows a hashing with a password-independent access pattern by hashing with a password-dependent access pattern.

[13] We provide a proof of security for single-pass Argon2i in Appendix F.

quality of a memory-hard function, so it is difficult to compare these functions quantitatively.

In this section, we propose one such metric and compare the four candidate functions under it. The metric we propose captures the notion that a good memory-hard function is one that makes the attacker's job as difficult as possible given that the defender (e.g., the legitimate authentication server) still needs to hash passwords in a reasonable amount of time. Let $\mathsf{T}_f(\mathcal{A})$ denote the expected running time of an algorithm $\mathcal{A}$ computing a function f and let $\mathsf{ST}_f(\mathcal{A})$ denote its expected space-time product. Then we define the quality Q of a memory-hard function against a sequential attacker $\mathcal{A}_S$ using space S to be the ratio:

$$Q[\mathcal{A}_S, f] = \frac{\mathsf{ST}_f(\mathcal{A}_S)}{\mathsf{T}_f(\text{Honest})}.$$

We can define a similar notion of quality in the amortized/parallel setting: just replace the quantity in the numerator (the adversary's space-time product) with the sum of a pROM adversary's space usage over time: $\sum_t S_t$ of $\mathcal{A}_S$.

We can now use the existing memory-hardness proofs to put lower bounds on the quality (in the sequential model) of the candidate memory-hard functions. We show in Appendix F that Argon2i has a sequential time-space lower bound of the form $S \cdot T \geq n^2/192$, for $S < n/24$. The n-block r-round Balloon function has a time-space lower-bound of the form $S \cdot T \geq (2^r - 1)n^2/8$ for $S < n/64$ when the parameter $\delta = 3$. The n-block Catena BRG function has a time-space lower bound of the form $S \cdot T \geq n^2/16$ (Catena BRG has no round parameter). The r-round n-block Catena DBG function has a claimed time-space lower bound of the form $S \cdot T \geq n(\frac{rn}{64S})^r$, when $S \leq n/20$. These lower-bounds yield the following quality figures against an adversary using roughly $n/64$ space:

$$Q[\mathcal{A}_S, \text{Balloon}_{(r=1)}] \geq \frac{n}{16}; \qquad Q[\mathcal{A}_S, \text{Balloon}_{(r>1)}] \geq \frac{(2^r - 1)n}{8(r+1)}$$

$$Q[\mathcal{A}_S, \text{Catena-BRG}] \geq \frac{n}{32}; \qquad Q[\mathcal{A}_S, \text{Catena-DBG}] \geq \frac{r^r}{2r \log_2 n}$$

$$Q[\mathcal{A}_S, \text{Argon2i}] \geq \frac{n}{192}$$

From these quality ratios, we can draw a few conclusions about the protection these functions provide against one class of small-space attackers (using $S \approx n/64$):

- In terms of provable memory-hardness properties in the sequential model, one-round Balloon always outperforms Catena-BRG and Argon2i.
- When the buffer size n grows and the number of rounds r is held fixed, Balloon outperforms Catena-DBG as well.
- When the buffer size n is fixed and the number of rounds r grows large, Catena-DBG provides the strongest provable memory-hardness properties in the sequential model.
- For many realistic choices of r and n (e.g., $r = 5$, $n = 2^{18}$), r-round Balloon outperforms the other constructions in terms of memory-hardness properties.

6 Experimental Evaluation

In this section, we demonstrate experimentally that the Balloon hashing algorithm is competitive performance-wise with two existing practical algorithms (Argon2i and Catena), when all are instiated with standard cryptographic primitives.

6.1 Experimental Set-up

Our experiments use the OpenSSL implementation (version 1.0.1f) of SHA-512 and the reference implementations of three other cryptographic hash functions (Blake2b, ECHO, and SHA-3/Keccak). We use optimized versions of the underlying cryptographic primitives where available, but the core Balloon code is written entirely in C. Our source code is available at https://crypto.stanford.edu/balloon/ under the ISC open-source license. We used a workstation running an Intel Core i7-6700 CPU (Skylake) at 3.40 GHz with 8 GiB of RAM for our performance benchmarks. We compiled the code for our timing results with gcc version 4.8.5 using the `-O3` option. We average all of our measurements over 32 trials. We compare the Balloon functions against Argon2i (v.1.2.1) [19] and Catena [42]. For comparison purposes, we implemented the Argon2i, Catena BRG, and Catena DBG memory-hard algorithms in C.

On the Choice of Cryptographic Primitives. The four memory-hard functions we evaluate (Argon2i, Balloon, Catena-BRG, Catena-DBG) are all essentially modes of operation for an underlying cryptographic hash function. The choice of the underlying hash function has implications for the performance and the security of the overall construction. To be conservative, we instantiate all of the algorithms we evaluate with the Blake2b as the underlying hash function [10].

Memory-hard functions going back at least as far as scrypt [68] have used reduced-round hash functions as their underlying cryptographic building block. Following this tradition, the Argon2i specification proposes using a new and very fast reduced-round hash function as its core cryptographic primitive. Since the Argon2i hash function does not satisfy basic properties of a traditional cryptographic hash function (e.g., it is not collision resistant), modeling it as a random oracle feels particularly problematic. Since our goal in this work is to analyze memory-hard functions with *provable* security guarantees, we instantiate the memory-hard functions we evaluate with traditional cryptographic hashes for the purposes of this evaluation.

That said, we stress that the Balloon construction is agnostic to the choice of underlying hash function—it is a mode of operation for a cryptographic hash function—and users of the Balloon construction may instantiate it with a faster reduced-round hash function (e.g., scrypt's BlockMix or Argon2i's compression function) if they so desire.

6.2 Authentication Throughput

The goal of a memory-hard password hash function is to *use as much working space as possible as quickly as possible* over the course of its computation. To evaluate the effectiveness of the Balloon algorithm on this metric, we measured the rate at which a server can check passwords (in hashes per second) for various buffer sizes on a single core.

Figure 5 shows the minimum buffer size required to compute each memory-hard function with high probability with no computational slowdown, for a variety of password hashing functions. We set the block size of the construction to be equal to the block size of the underlying compression function, to avoid the issues discussed in Appendix B.3. The charted results for Argon2i incorporate the fact that an adversary can compute many-pass Argon2i (v.1.2.1) in a factor of $e \approx 2.72$ less working space than the defender must allocate for the computation and can compute single-pass Argon2i with a factor of four less space (see Section 4). For comparison, we also plot the space usage of two non-memory-hard password hashing functions, bcrypt [75] (with cost = 12) and PBKDF2-SHA512 [49] (with 10^5 iterations).

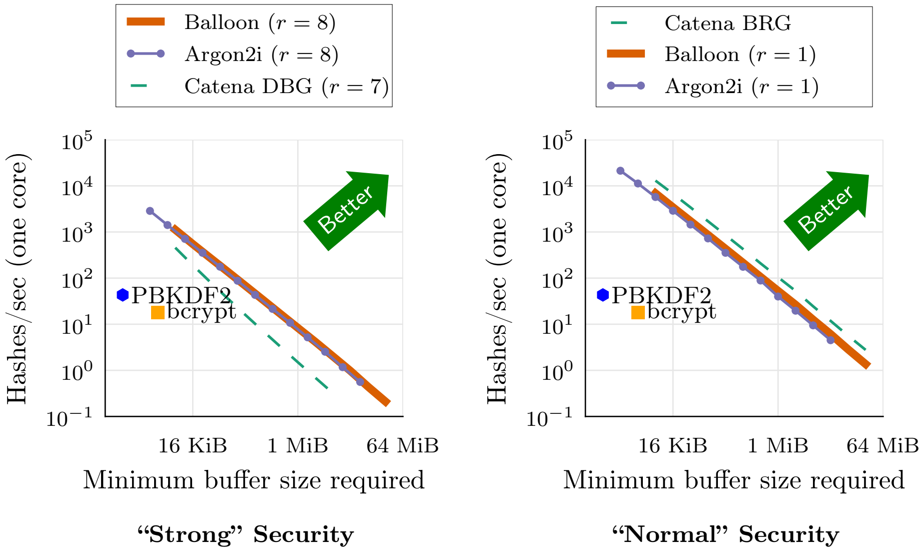


Fig. 5: The Balloon algorithm outperforms Argon2i and Catena DBG for many settings of the security parameters, and Balloon is competitive with Catena BRG. We instantiate Argon2i, Balloon, and Catena with Blake2b as the underlying cryptographic hash function.

If we assume that an authentication server must perform 100 hashes per second per four-core machine, Figure 5 shows that it would be possible to use one-round Balloon hashing with a 2 MiB buffer or eight-round Balloon hashing with a 256 KiB buffer. At the same authentication rate, Argon2i (instantiated with Blake2b as the underlying cryptographic hash function) requires the attacker to use a smaller buffer—roughly 1.5 MiB for the one-pass variant. Thus, with Balloon hashing we simultaneously get better performance than Argon2i and stronger memory-hardness guarantees.

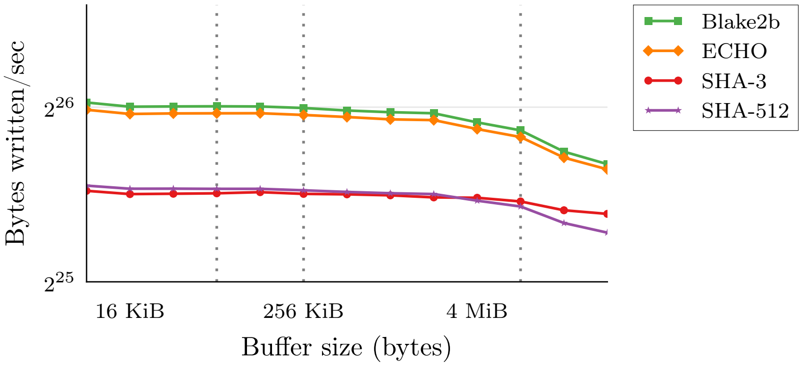


Fig. 6: Throughput for the Balloon algorithm when instantiated with different compression functions. The dotted lines indicate the sizes of the L1, L2, and L3 caches on our test machine.

6.3 Compression Function

Finally, Figure 6 shows the result of instantiating the Balloon algorithm construction with four different standard cryptographic hash functions: SHA-3 [18], Blake2b [10], SHA-512, and ECHO (a SHA-3 candidate that exploits the AES-NI instructions) [15]. The SHA-3 function (with rate = 1344) operates on 1344-bit blocks, and we configure the other hash functions to use 512-bit blocks.

On the x-axis, we plot the buffer size used in the Balloon function and on the y-axis, we plot the rate at which the Balloon function fills memory, in bytes of written per second. As Figure 6 demonstrates, Blake2b and ECHO outperform the SHA functions by a bit less than a factor of two.

7 Related Work

Password Hashing. The problem of how to securely store passwords on shared computer systems is nearly as old as the systems themselves. In a 1974 article, Evans et al. described the principle of storing passwords under a hard-to-invert function [40]. A few years later, Robert Morris and Ken Thompson presented the now-standard notion of password *salts* and explained how to store passwords under a moderately hard-to-compute one-way function to increase the cost of dictionary attacks [59]. Their DES-based "crypt" design became the standard for password storage for over a decade [55] and even has a formal analysis by Wagner and Goldberg [90].

In 1989, Feldmeier and Karn found that hardware improvements had driven the cost of brute-force password guessing attacks against DES crypt down by five orders of magnitude since 1979 [41, 52]. Poul-Henning Kamp introduced the costlier md5crypt to replace crypt, but hardware improvements also rendered that design outmoded [31].

Provos and Mazières saw that, in the face of ever-increasing processor speeds, any fixed password hashing algorithm would eventually become easy to compute and thus ineffective protection against dictionary attacks. Their solution, bcrypt, is a password hashing scheme with a variable "hardness" parameter [75]. By periodically ratcheting up the hardness, a system administrator can keep the time needed to compute a single hash roughly constant, even as hardware improves. A remaining weakness of bcrypt is that it exercises only a small fraction of the CPU's resources—it barely touches the L2 and L3 caches during its execution [56]. To increase the cost of custom password-cracking hardware, Reinhold's HEKS hash [76] and Percival's popular scrypt routine consume an adjustable amount of storage space [68], in addition to time, as they compute a hash. Balloon, like scrypt, aims to be hard to compute in little space. Unlike scrypt, however, we require that our functions' data access pattern be independent of the password to avoid leaking information via cache-timing attacks [25, 62, 87] (see also the attack in Appendix G.2). The Dogecoin and Litecoin [24] cryptocurrencies have incorporated scrypt as an ASIC-resistant proof-of-work function.

The recent Password Hashing Competition motivated the search for memory-hard password-hashing functions that use data-independent memory access patterns [64]. The Argon2 family of functions, which have excellent performance and an appealingly simple design, won the competition [19]. The Argon2 functions lack a theoretical analysis of the feasible time-space trade-offs against them; using the same ideas we have used to analyze the Balloon function, we provide the first such result in Appendix F.

The Catena hash functions [42], which became finalists in the Password Hashing Competition, are memory-hard functions whose analysis applies pebbling arguments to classic graph-theoretic results of Lengauer and Tarjan [54]. The Balloon analysis we provide gives a tighter time-space lower bounds than Catena's analysis can provide in many cases, and the Balloon algorithm outperforms the more robust of the two Catena algorithms (see Section 6). Biryokov and Khovratovich demonstrated a serious flaw in the security analysis of one of the Catena variants, and they provide a corresponding attack against that Catena variant [22].

The other competition finalists included a number of interesting designs that differ from ours in important ways. Makwa [73] supports offloading the work of password hashing to an untrusted server but is not memory-hard. Lyra [2] is a memory-hard function but lacks proven space-time lower bounds. Yescrypt [69] is an extension of scrypt and uses a password-dependent data access pattern.

Ren and Devadas [77] give an analysis of the Balloon algorithm using bipartite expanders, following the pebbling techniques of Paul and Tarjan [66]. Their results imply that an adversary that computes the n-block r-round Balloon function in $n/8$ space, must use at least $2^r n/c$ time to compute the function (for some constant c), with high probability in the random-oracle model. We prove the stronger statement that an adversary's space-time product must satisfy: $S \cdot T \in \Omega(n^2)$ for almost all values of S. Ren and Devadas also prove statements showing that algorithms computing the Balloon functions efficiently must use a certain amount of space at *many points* during their computation. Our time-space lower bounds only show that the adversary must use a certain amount of space a *some point* during the Balloon computation.

Other Studies of Password Protection. Concurrently with the design of hashing schemes, there has been theoretical work from Bellare et al. on new security definitions for password-based cryptography [13] and from Di Crescenzo et al. on an analysis of passwords storage systems secure against adversaries that can steal only a bounded number of bits of the password file [33]. Other ideas for modifying password hashes include the *key stretching* schemes of Kelsey et al. [50] (variants on iterated hashes), a proposal by Boyen to keep the hash iteration count (e.g., time parameter in bcrypt) secret [26], a technique of Canetti et al. for using CAPTCHAs in concert with hashes [28], and a proposal by Dürmuth to use password hashing to do meaningful computation [34].

Parallel Memory-Hardness. In a recent line of work [3, 4, 5, 6, 8] has analyzed memory-hard functions from a number of angles in the parallel random-oracle model, introduced by Alwen and Serbinenko [8]. We discuss these very relevant results at length in Section 5.1.

Memory-Bound Functions. Abadi et al. [1] introduced memory-bound functions as more effective alternatives to traditional proofs-of-work in heterogeneous computing environments [11, 36]. These functions require many cache misses to compute and, under the assumption that memory latencies are consistent across computing platforms, they are roughly as hard to compute on a computationally powerful device as on a computationally weak one. The theoretical analysis of memory-bound functions represented one of the first applications of pebbling arguments to cryptography [35, 37].

Proofs of Space. Dziembowski et al. [38] and Ateniese et al. [9] study proofs-of-space. In these protocols, the prover and verifier agree on a large bitstring that the prover is supposed to store. Later on, the prover can convince the verifier that the prover has stored some large string on disk, even if the verifier does not store the string herself. Spacemint proposes building a cryptocurrency based upon a proof-of-space rather than a proof-of-work [63]. Ren and Devadas propose using the problem of pebbling a Balloon graph as the basis for a proof of space [77].

Time-Space Trade-Offs. The techniques we use to analyze Balloon draws on extensive prior work on computational time-space trade-offs. We use pebbling arguments, which have seen application to register allocation problems [81], to the analysis of the relationships between complexity classes [16, 29, 48, 85], and to prior cryptographic constructions [37, 38, 39, 42]. Pebbling has also been a topic of study in its own right [54, 67]. Savage's text gives a clear introduction to graph pebbling [80] and Nordström surveys the vast body of pebbling results in depth [61].

8 Conclusion

We have introduced the Balloon password hashing algorithm. The Balloon algorithm is provably memory-hard (in the random-oracle model against sequential adversaries), exhibits a password-independent memory access pattern, and meets or exceeds the performance of the fastest heuristically secure schemes. Using a novel combinatorial pebbling argument, we have demonstrated that password-hashing algorithms can have memory-hardness proofs without sacrificing practicality.

This work raises a number of open questions:

- Are there efficient methods to defend against cache attacks on scrypt (Appendix G.2)? Could a special-purpose ORAM scheme help [44]?
- Are there *practical* memory-hard functions with password-independent access patterns that retain their memory-hardness properties under parallel attacks [8]? The recent work of Alwen et al. [4] is promising, though it is still unclear whether the pROM-secure constructions will be competitive with Balloon for concrete settings of the parameters.
- Is it possible to build hardware that effectively implements the pROM attacks [3, 4, 5] against Argon2i and Balloon at realistic parameter sizes? What efficiency gain would this pROM hardware have over a sequential ASIC at attacking these constructions? Are these parallel attacks still practical in hardware when the function's memory-access pattern depends on the salt (as Balloon's access pattern does)?

Acknowledgements. We would like to our anonymous reviewers for their helpful comments. We also thank Josh Benaloh, Joe Bonneau, Greg Hill, Ali Mashtizadeh, David Mazières, Yan Michalevsky, Bryan Parno, Greg Valiant, Riad Wahby, Keith Winstein, David Wu, Sergey Yekhanin, and Greg Zaverucha for comments on early versions of this work. This work was funded in part by an NDSEG Fellowship, NSF, DARPA, a grant from ONR, and the Simons Foundation. Opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

Bibliography

[1] Abadi, M., Burrows, M., Manasse, M., Wobber, T.: Moderately hard, memory-bound functions. ACM Transactions on Internet Technology 5(2), 299–327 (2005)

[2] Almeida, L.C., Andrade, E.R., Barreto, P.S.L.M., Simplicio Jr., M.A.: Lyra: Password-based key derivation with tunable memory and processing costs. Journal of Cryptographic Engineering 4(2), 75–89 (2014)

[3] Alwen, J., Blocki, J.: Efficiently computing data-independent memory-hard functions. In: CRYPTO (2016)

[4] Alwen, J., Blocki, J.: Towards practical attacks on Argon2i and Balloon Hashing. Cryptology ePrint Archive, Report 2016/759 (2016), http://eprint.iacr.org/2016/759

[5] Alwen, J., Blocki, J., Pietrzak, K.: The pebbling complexity of depth-robust graphs. Manuscript (Personal Communication) (2016)

[6] Alwen, J., Chen, B., Kamath, C., Kolmogorov, V., Pietrzak, K., Tessaro, S.: On the complexity of scrypt and proofs of space in the parallel random oracle model. In: EUROCRYPT 2016, pp. 358–387. Springer (2016)

[7] Alwen, J., Chen, B., Kamath, C., Kolmogorov, V., Pietrzak, K., Tessaro, S.: On the complexity of scrypt and proofs of space in the parallel random oracle model. Cryptology ePrint Archive, Report 2016/100 (2016), http://eprint.iacr.org/

[8] Alwen, J., Serbinenko, V.: High parallel complexity graphs and memory-hard functions. In: STOC. pp. 595–603 (2015)

[9] Ateniese, G., Bonacina, I., Faonio, A., Galesi, N.: Proofs of space: When space is of the essence. In: Security and Cryptography for Networks, pp. 538–557. Springer (2014)

[10] Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C.: BLAKE2: simpler, smaller, fast as MD5. In: Applied Cryptography and Network Security. pp. 119–135. Springer (2013)

[11] Back, A.: Hashcash–a denial of service counter-measure. http://www.cypherspace.org/hashcash/ (May 1997), accessed 9 November 2015

[12] Bellare, M., Boldyreva, A., Palacio, A.: An uninstantiable random-oracle-model scheme for a hybrid-encryption problem. In: EUROCRYPT 2004. pp. 171–188. Springer (2004)

[13] Bellare, M., Ristenpart, T., Tessaro, S.: Multi-instance security and its application to password-based cryptography. In: CRYPTO, pp. 312–329. Springer (2012)

[14] Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: CCS. pp. 62–73. ACM (1993)

[15] Benadjila, R., Billet, O., Gilbert, H., Macario-Rat, G., Peyrin, T., Robshaw, M., Seurin, Y.: SHA-3 proposal: ECHO. Submission to NIST (updated) (2009)

[16] Bennett, C.H.: Time/space trade-offs for reversible computation. SIAM Journal on Computing 18(4), 766–776 (1989)

[17] Bernstein, D.J., Lange, T.: Non-uniform cracks in the concrete: the power of free precomputation. In: ASIACRYPT, pp. 321–340. Springer (2013)

[18] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak sponge function family. Submission to NIST (Round 2) (2009)

[19] Biryukov, A., Dinu, D., Khovratovich, D.: Argon2 design document (version 1.2.1) (Oct 2015)

[20] Biryukov, A., Dinu, D., Khovratovich, D.: Fast and tradeoff-resilient memory-hard functions for cryptocurrencies and password hashing. Cryptology ePrint Archive, Report 2015/430 (2015), http://eprint.iacr.org/

[21] Biryukov, A., Dinu, D., Khovratovich, D.: Argon2 design document (version 1.3) (Feb 2016)

[22] Biryukov, A., Khovratovich, D.: Tradeoff cryptanalysis of memory-hard functions. In: ASIACRYPT. pp. 633–657. Springer (2015)

[23] Bitcoin wiki - mining comparison. https://en.bitcoin.it/wiki/Mining_hardware_comparison

[24] Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In: Symposium on Security and Privacy. IEEE (May 2015)

[25] Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: CHES 2006, pp. 201–215. Springer (2006)

[26] Boyen, X.: Halting password puzzles. In: USENIX Security (2007)

[27] Canetti, R., Goldreich, O., Halevi, S.: The Random Oracle Methodology, revisited. Journal of the ACM 51(4), 557–594 (2004)

[28] Canetti, R., Halevi, S., Steiner, M.: Mitigating dictionary attacks on password-protected local storage. In: CRYPTO 2006, pp. 160–179. Springer (2006)

[29] Chan, S.M.: Just a pebble game. In: IEEE Conference on Computational Complexity. pp. 133–143. IEEE (2013)

[30] Coron, J., Dodis, Y., Malinaud, C., Puniya, P.: Merkle-damgård revisited: How to construct a hash function. In: CRYPTO. pp. 430–448 (2005)

[31] CVE-2012-3287: md5crypt has insufficient algorithmic complexity. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3287 (2012), accessed 9 November 2015

[32] Damgård, I.B.: A design principle for hash functions. In: CRYPTO. pp. 416–427 (1989)

[33] Di Crescenzo, G., Lipton, R., Walfish, S.: Perfectly secure password protocols in the bounded retrieval model. In: Theory of Cryptography, pp. 225–244. Springer (2006)

[34] Dürmuth, M.: Useful password hashing: how to waste computing cycles with style. In: New Security Paradigms Workshop. pp. 31–40. ACM (2013)

[35] Dwork, C., Goldberg, A., Naor, M.: On memory-bound functions for fighting spam. In: CRYPTO, pp. 426–444. Springer (2003)

[36] Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: CRYPTO 1992. pp. 139–147. Springer (1993)

[37] Dwork, C., Naor, M., Wee, H.: Pebbling and proofs of work. In: CRYPTO. pp. 37–54 (2005)

[38] Dziembowski, S., Faust, S., Kolmogorov, V., Pietrzak, K.: Proofs of space. In: CRYPTO (2015)

[39] Dziembowski, S., Kazana, T., Wichs, D.: One-time computable self-erasing functions. In: Theory of Cryptography, pp. 125–143. Springer (2011)

[40] Evans Jr, A., Kantrowitz, W., Weiss, E.: A user authentication scheme not requiring secrecy in the computer. Communications of the ACM 17(8), 437–442 (1974)

[41] Feldmeier, D.C., Karn, P.R.: UNIX password security–ten years later. In: CRYPTO 1989. pp. 44–63. Springer (1990)

[42] Forler, C., Lucks, S., Wenzel, J.: Memory-demanding password scrambling. In: ASIACRYPT, pp. 289–305. Springer (2014)

[43] Garay, J., Johnson, D., Kiayias, A., Yung, M.: Resource-based corruptions and the combinatorics of hidden diversity. In: ITCS. pp. 415–428. ACM (2013)

[44] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the ACM 43(3), 431–473 (1996)

[45] Goldwasser, S., Kalai, Y.T.: On the (in)security of the Fiat-Shamir paradigm. In: FOCS. pp. 102–113. IEEE (2003)

[46] Groza, B., Warinschi, B.: Revisiting difficulty notions for client puzzles and DoS resilience. In: Information Security, pp. 39–54. Springer (2012)

[47] Ho, S.: Costco, Sam's Club, others halt photo sites over possible breach. http://www.reuters.com/article/2015/07/21/us-cyberattack-retail-idUSKCN0PV00520150721 (Jul 2015), accessed 9 November 2015

[48] Hopcroft, J., Paul, W., Valiant, L.: On time versus space. Journal of the ACM (JACM) 24(2), 332–337 (1977)

[49] Kaliski, B.: PKCS #5: Password-based cryptography specification, version 2.0. IETF Network Working Group, RFC 2898 (Sep 2000)

[50] Kelsey, J., Schneier, B., Hall, C., Wagner, D.: Secure applications of low-entropy keys. In: Information Security, pp. 121–134. Springer (1998)

[51] Kirk, J.: Internet address overseer ICANN resets passwords after website breach. http://www.pcworld.com/article/2960592/security/icann-resets-passwords-after-website-breach.html (Aug 2015), accessed 9 November 2015

[52] Klein, D.V.: Foiling the cracker: A survey of, and improvements to, password security. In: Proceedings of the 2nd USENIX Security Workshop. pp. 5–14 (1990)

[53] Krantz, L.: Harvard says data breach occurred in June. http://www.bostonglobe.com/metro/2015/07/01/harvard-announces-data-breach/pqzk9IPWLMiCKBl3IijMUJ/story.html (Jul 2015), accessed 9 November 2015

[54] Lengauer, T., Tarjan, R.E.: Asymptotically tight bounds on time-space trade-offs in a pebble game. Journal of the ACM 29(4), 1087–1130 (1982)

[55] Leong, P., Tham, C.: UNIX password encryption considered insecure. In: USENIX Winter. pp. 269–280 (1991)

[56] Malvoni, K., Designer, S., Knezovic, J.: Are your passwords safe: Energy-efficient bcrypt cracking with low-cost parallel hardware. In: USENIX Workshop on Offensive Technologies (2014)

[57] Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC press (1996)

[58] Merkle, R.C.: One way hash functions and DES. In: CRYPTO. pp. 428–446 (1989)

[59] Morris, R., Thompson, K.: Password security: A case history. Communications of the ACM 22(11), 594–597 (1979)

[60] Nielsen, J.B.: Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In: CRYPTO 2002, pp. 111–126. Springer (2002)

[61] Nordström, J.: New wine into old wineskins: A survey of some pebbling classics with supplemental results. http://www.csc.kth.se/~jakobn/research/PebblingSurveyTMP.pdf (Mar 2015), accessed 9 November 2015

[62] Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: CT-RSA 2006, pp. 1–20. Springer (2006)

[63] Park, S., Pietrzak, K., Alwen, J., Fuchsbauer, G., Gazi, P.: Spacemint: a cryptocurrency based on proofs of space. Tech. rep., Cryptology ePrint Archive, Report 2015/528 (2015)

[64] Password hashing competition. https://password-hashing.net/

[65] Paterson, M.S., Hewitt, C.E.: Comparative schematology. In: Record of the Project MAC conference on concurrent systems and parallel computation. pp. 119–127. ACM (1970)

[66] Paul, W.J., Tarjan, R.E.: Time-space trade-offs in a pebble game. Acta Informatica 10(2), 111–115 (1978)

[67] Paul, W.J., Tarjan, R.E., Celoni, J.R.: Space bounds for a game on graphs. Mathematical Systems Theory 10(1), 239–251 (1976)

[68] Percival, C.: Stronger key derivation via sequential memory-hard functions. In: BSDCan (May 2009)

[69] Peslyak, A.: yescrypt. https://password-hashing.net/submissions/specs/yescrypt-v2.pdf (Oct 2015), accessed 13 November 2015

[70] Peterson, A.: E-Trade notifies 31,000 customers that their contact info may have been breached in 2013 hack. https://www.washingtonpost.com/news/the-switch/wp/2015/10/09/e-trade-notifies-31000-customers-that-their-contact-info-may-have-been-breached-in-2013-hack/ (Oct 2015), accessed 9 November 2015

[71] Pippenger, N.: A time-space trade-off. Journal of the ACM (JACM) 25(3), 509–515 (1978)

[72] Pippenger, N., Fischer, M.J.: Relations among complexity measures. Journal of the ACM 26(2), 361–381 (1979)

[73] Pornin, T.: The Makwa password hashing function. http://www.bolet.org/makwa/ (Apr 2015), accessed 13 November 2015

[74] Privacy Rights Clearinghouse: Chronology of data breaches. http://www.privacyrights.org/data-breach, accessed 9 November 2015

[75] Provos, N., Mazières, D.: A future-adaptable password scheme. In: USENIX Annual Technical Conference. pp. 81–91 (1999)

[76] Reinhold, A.: HEKS: A family of key stretching algorithms (Draft G). http://world.std.com/~reinhold/HEKSproposal.html (Jul 2001), accessed 13 November 2015

[77] Ren, L., Devadas, S.: Proof of space from stacked expanders. Cryptology ePrint Archive, Report 2016/333 (2016), http://eprint.iacr.org/

[78] Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: CCS. pp. 199–212. ACM (2009)

[79] Rogaway, P.: Formalizing human ignorance. In: VIETCRYPT, pp. 211–228. Springer (2006)

[80] Savage, J.E.: Models of computation: Exploring the Power of Computing. Addison-Wesley (1998)

[81] Sethi, R.: Complete register allocation problems. SIAM journal on Computing 4(3), 226–248 (1975)

[82] Smith, A., Zhang, Y.: Near-linear time, leakage-resilient key evolution schemes from expander graphs. IACR Cryptology ePrint Archive, Report 2013/864 (2013)

[83] Stebila, D., Kuppusamy, L., Rangasamy, J., Boyd, C., Nieto, J.G.: Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols. In: CT-RSA, pp. 284–301. Springer (2011)

[84] Takala, R.: UVA site back online after chinese hack. http://www.washingtonexaminer.com/uva-site-back-online-after-chinese-hack/article/2570383 (Aug 2015), accessed 9 November 2015

[85] Tompa, M.: Time-space tradeoffs for computing functions, using connectivity properties of their circuits. In: STOC. pp. 196–204. ACM (1978)

[86] Tracy, A.: In wake of T-Mobile and Experian data breach, John Legere did what all CEOs should do after a hack. http://www.forbes.com/sites/abigailtracy/2015/10/02/in-wake-of-t-mobile-and-experian-data-breach-john-legere-did-what-all-ceos-should-do-after-a-hack/ (Oct 2015), accessed 9 November 2015

[87] Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on AES, and countermeasures. Journal of Cryptology 23(1), 37–71 (2010)

[88] Valiant, L.G.: Graph-theoretic arguments in low-level complexity. In: Gruska, J. (ed.) Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, vol. 53, pp. 162–176. Springer Berlin Heidelberg (1977)

[89] Vaughan-Nichols, S.J.: Password site LastPass warns of data breach. http://www.zdnet.com/article/lastpass-password-security-site-hacked/ (Jun 2015), accessed 9 November 2015

[90] Wagner, D., Goldberg, I.: Proofs of security for the Unix password hashing algorithm. In: ASIACRYPT 2000, pp. 560–572. Springer (2000)

A Details of the Attack on Argon2

In this section, we provide a detailed analysis of the attack on Argon2i that we introduced in Section 4.

The goal of the attack algorithm is to compute Argon2i in the same number of time steps as the naïve algorithm uses to compute the function, while using a constant factor less space than the naïve algorithm does. In this way, an attacker mounting a dictionary attack against a list of passwords hashed with Argon2i can do so at less cost (in terms of the space-time product) than the Argon2i specification claimed possible.

Argon2i has one-pass and many-pass variants and our attack applies to both; the many-pass variant is recommended in the specification. We first analyze the attack on the one-pass variant and then analyze the attack on the many-pass variant.

We are interested in the attack algorithm's expected space usage at time step t—call this function $S(t)$.[14]

Analysis of One-Pass Argon2i. At each step of the one-pass Argon2i algorithm, the expected space usage $S(t)$ is equal to the number of memory blocks generated so far minus the expected number of blocks in memory that will never be used after time t. Let $A_{i,t}$ be the event that block i is never needed after time step t in the computation. Then $S(t) = t - \sum_{i=1}^{t} \Pr[A_{i,t}]$.

To find $S(t)$ explicitly, we need to compute the probability that block i is never used after time t. We know that the probability that block i is never used after time t is equal to the probability that block i is not used at time $t+1$ *and* is not used at time $t+2$ *and* $[\dots]$ *and* is not used at time n. Let $U_{i,t}$ denote the event that block i is *unused* at time t. Then:

$$\Pr\left[A_{i,t}\right] = \Pr\left[\bigcap_{t'=t+1}^{n} U_{i,t'}\right] = \prod_{t'=t+1}^{n} \Pr[U_{i,t'}] \tag{1}$$

The equality on the right-hand side comes from the fact that $U_{i,t'}$ and $U_{i,t''}$ are independent events for $t' \neq t''$.

To compute the probability that block i is not used at time t', consider that there are $t'-1$ blocks to choose from and $t'-2$ of them are *not* block i: $\Pr[U_{i,t'}] = \frac{t'-2}{t'-1}$. Plugging this back into Equation 1, we get:

$$\Pr\left[A_{i,t}\right] = \prod_{t'=t+1}^{n} \left(\frac{t'-2}{t'-1}\right) = \frac{t-1}{n-1}$$

[14] As described in Section 4.2, the contents of block i in Argon2i are derived from the contents of block $i-1$ and a block chosen at random from the set $r_i \overset{\mathrm{R}}{\leftarrow} \{1,\dots,i-1\}$. Throughout our analysis, all probabilities are taken over the random choices of the r_i values.

Now we substitute this back into our original expression for $S(t)$:

$$S(t) = t - \sum_{i=1}^{t} \left(\frac{t-1}{n-1} \right) = t - \frac{t(t-1)}{n-1}$$

Taking the derivative $S'(t)$ and setting it to zero allows us to compute the value t for which the expected storage is maximized. The maximum is at $t = n/2$ and the expected number of blocks required is $S(n/2) \approx n/4$.

Larger in-degree. A straightforward extension of this analysis handles the case in which δ random blocks—instead of one—are hashed together with the prior block at each step of the algorithm. Our analysis demonstrates that, even with this strategy, single-pass Argon2i is vulnerable to pre-computation attacks. The maximum space usage comes at $t^* = n/(\delta+1)^{1/\delta}$, and the expected space usage over time $S(t)$ is:

$$S(t) \approx t - \frac{t^{\delta+1}}{n^{\delta}} \quad \text{so} \quad S(t^*) \approx \frac{\delta}{(\delta+1)^{1+1/\delta}} n \ .$$

Analysis of Many-Pass Argon2i. One idea for increasing the minimum memory consumption of Argon2i is to increase the number of passes that the algorithm takes over the memory. For example, the Argon2 specification proposes taking three passes over the memory to protect against certain time-space tradeoffs. Unfortunately, even after *many* passes over the memory, the Argon2i algorithm sketched above still uses many fewer than n blocks of memory, in expectation, at each time step.

To investigate the space usage of the many-pass Argon2i algorithm, first consider that the space usage will be maximized at some point in the middle of its computation—not in the first or last passes. At some time step t in the middle of its computation the algorithm will have at most n memory blocks in storage, but the algorithm can delete any of these n blocks that it will never need after time t.

At each time step, the algorithm adds a new block to the end of the buffer and deletes the first block. At any one point in the algorithm's execution, there will be at most n blocks of memory in storage. If we freeze the execution of the Argon2i algorithm in the middle of its execution, we can inspect the n blocks it has stored in memory. Call the first block "stored block 1" and the last block "stored block n."

Let $B_{i,t}$ denote the event that stored block i is never needed after time t. Then we claim $\Pr[B_{i,t}] = (\frac{n-1}{n})^i$. To see the logic behind this calculation: notice that, at time t, the first stored block in the buffer can be accessed at time $t+1$ but by time $t+2$, the first stored block will have been deleted from the buffer. Similarly, the second stored block in the buffer at time t can be accessed at time $t+1$ or $t+2$, but not $t+3$ (since by then stored block 2 will have been deleted from the buffer). Similarly, stored block i can be accessed at time steps $(t+1)$, $(t+2)$, …, $(t+i)$ but not at time step $(t+i+1)$.

The total storage required is then:

$$S(t) = n - \sum_{i=1}^{n} \mathrm{E}[B_{i,t}] = n - \sum_{i=1}^{n} \left(\frac{n-1}{n}\right)^i \approx n - n\left(1 - \frac{1}{e}\right).$$

Thus, even after many passes over the memory, Argon2i can still be computed in roughly n/e space with no time penalty.

B Pebble Games

In this section, we introduce pebble games and explain how to use them to analyze the Balloon algorithm.

B.1 Rules of the Game

The *pebble game* is a one-player game that takes place on a directed acyclic graph $G = (V, E)$. If there is an edge $(u, v) \in E$, we say that v is a *successor* of u in the directed graph and that u is a *predecessor* of v. As usual, we refer to nodes of the graph with in-degree zero as *source* nodes and nodes with out-degree zero as *sink* nodes—edges point from sources to sinks.

A pebbling game, defined next, represents a computation. The pebbles represent intermediate values stored in memory.

Definition 2 (Pebbling game). A pebbling game on a directed acyclic graph $G = (V, E)$ consists of a sequence of player moves, where each move is one of the following:

- place a pebble on a source vertex,
- remove a pebble from any pebbled vertex, or
- place a pebble on a non-source vertex *if and only if* all of its predecessor vertices are pebbled.

The game ends when a pebble is placed on a designated sink vertex. A sequence of pebbling moves is *legal* if each move in the sequence obeys the rules of the game. □

The pebble game typically begins with no pebbles on the graph, but in our analysis we will occasionally define partial pebblings that begin in a particular configuration $\mathcal{C}$, in which some some vertices are already pebbled.

The pebble game is a useful model of *oblivious* computation, in which the data-access pattern is independent of the value being computed [72]. Edges in the graph correspond to data dependencies, while vertices correspond to intermediate values needed in the computation. Source nodes represent input values (which have no dependencies) and sink nodes represent output values. The pebbles on the graph correspond to values stored in the computer's memory at a point in the computation. The three possible moves in the pebble game then correspond to: (1) loading an input value into memory, (2) deleting a value stored in memory, and (3) computing an intermediate value from the values of its dependencies.

B.2 Pebbling in the Random-Oracle Model

Dwork, Naor, and Wee [37] demonstrated that there is a close relationship between the pebbling problem on a graph G and the problem of computing a certain function f_G in the random-oracle model [14]. This observation became the basis for the design of the Catena memory-hard hash function family [42] and is useful for our analysis as well.

Since the relationship between G and f_G will be important for our construction and security proofs, we will summarize here the transformation of Dwork et al. [37], as modified by Alwen and Serbinenko [8]. The transformation from directed acyclic graph $G = (V, E)$ to function f_G works by assigning a *label* to each vertex $v \in V$, with the aid of a cryptographic hash function H. We write the vertex set of G topologically $\{v_1, \dots v_{|V|}\}$ such that v_1 is a source and $v_{|V|}$ is a sink and, to simplify the discussion, we assume that G has a unique sink node.

Definition 3 (Labeling). Let $G = (V, E)$ be a directed graph with maximum in-degree δ and a unique sink vertex, let $x \in \{0,1\}^k$ be a string, and let $H : \mathbb{Z}_{|V|} \times \left(\{0,1\}^k \cup \{\perp\}\right)^\delta \to \{0,1\}^k$ be a function, modeled as a random oracle. We define the labeling of G relative to H and x as:

$$\mathsf{label}_x(v_i) = \begin{cases} H(i, x, \perp, \dots, \perp) & \text{if } v_i \text{ is a source} \\ H(i, \mathsf{label}_x(z_1), \dots, \mathsf{label}_x(z_\delta)) & \text{o.w.} \end{cases}$$

where $z_1, \dots, z_\delta$ are the predecessors of v_i in the graph G. If v_i has fewer than δ predecessors, a special "empty" label ($\perp$) is used as placeholder input to H.

The labeling of the graph G proceeds from the sources to the unique sink node: first, the sources of G receive labels, then their successors receive labels, and so on until finally the unique sink node receives a label. To convert a graph G into a function $f_G : \{0,1\}^k \to \{0,1\}^k$, we define $f_G(x)$ as the function that outputs the label of the unique sink vertex under the labeling of G relative to a hash function H and an input x.

Dwork et al. demonstrate that any valid pebbling of the graph G with S pebbles and T placements immediately yields a method for computing f_G with Sk bits of space and T queries to the random oracle. Thus, upper bounds on the pebbling cost of a graph G yield upper bounds on the computation cost of the function f_G. In the other direction, they show that with high probability, an algorithm for computing f_G with space Sk and T random oracle queries yields a pebbling strategy for G using roughly S pebbles and T placements [37, Lemma 1].[15] Thus, lower bounds on the pebbling cost of the graph G yield lower bounds on the space and time complexity of the function f_G.

[15] This piece of the argument is subtle, since an adversarial algorithm for computing f_G could store parts of labels, might try to guess labels, or might use some other arbitrary strategy to compute the labeling. Showing that *every* algorithm that computing f_G with high probability yields a pebbling requires handling all of these possible cases.

We use a version of their result due to Dziembowski et al. [39]. The probabilities in the following theorems are over the choice of the random oracle and the randomness of the adversary.

Theorem 4 (Adapted from Theorem 4.2 of Dziembowski et al. [39]) *Let G, H, and k be as in Definition 3. Let $\mathcal{A}$ be an adversary making at most T random-oracle queries during its computation of $f_G(x)$. Then, given the sequence of $\mathcal{A}$'s random oracle queries, it is possible to construct a pebbling strategy for G with the following properties:*

1. *The pebbling is legal with probability $1 - T/2^k$.*
2. *If $\mathcal{A}$ uses at most Sk bits of storage then, for any $\lambda > 0$, the number of pebbles used is at most $\frac{Sk+\lambda}{k-\log_2 T}$ with probability $1 - 2^{-\lambda}$.*
3. *The number of pebble placements (i.e., moves in the pebbling) is at most T.*

Proof. The first two parts are a special case of Theorem 4.2 of Dziembowski et al. [39]. To apply their theorem, consider the running time and space usage of the algorithm $\mathcal{A}_{\text{small}}$ they define, setting $c = 0$. The third part of the theorem follows immediately from the pebbling they construct in the proof: there is at most one pebble placement per oracle call. There are at most T oracle calls, so the total number of placements is bounded by T. □

Informally, the lemma states that an algorithm using Sk bits of space will rarely be able to generate a sequence of random oracle queries whose corresponding pebbling places more than S pebbles on the graph or makes an invalid pebbling move.

The essential point, as captured in the following theorem, is that if we can construct a graph G that takes a lot of time to pebble when using few pebbles, then we can construct a function f_G that requires a lot of time to compute with high probability when using small space, in the random oracle model.

Theorem 5 *Let G and k be as in Definition 3 with the additional restriction that there is no pebbling strategy for G using S^* pebbles and T^* pebble placements, where T^* is less than $2^k - 1$. Let $\mathcal{A}$ be an algorithm that makes T random oracle queries and uses σ bits of storage space. If*

$$T < T^* \qquad \text{and} \qquad \sigma < S^*(k - \log_2 T^*) - k,$$

then $\mathcal{A}$ correctly computes $f_G(\cdot)$ with probability at most $\frac{T+1}{2^k}$.

Proof. Fix an algorithm $\mathcal{A}$ as in the statement of the theorem. By Theorem 4, from a trace of $\mathcal{A}$'s execution we can extract a pebbling of G that:

- is legal with probability at least $1 - T/2^k$,
- uses at most $\frac{\sigma+k}{k-\log_2 T^*} < S^*$ pebbles with probability at least $1 - 2^{-k}$, and
- makes at most T pebble placements.

By construction of G, there does not exist a pebbling of G using S^* pebbles and T^* pebble placements. Thus, whenever $\mathcal{A}$ succeeds at computing $f_G(\cdot)$ it must be that either (1) the pebbling we extract from $\mathcal{A}$ is invalid, (2) the pebbling we extract from $\mathcal{A}$ uses more than S^* pebbles, or (3) the pebbling we extract from $\mathcal{A}$ uses more than T^* moves. From Theorem 4, the probability of the first event is at most $T/2^k$, the probability of the second event is at most $1/2^k$, and the probability of the third event is zero.

By the Union Bound, we find that:

$$\begin{aligned}\Pr[\mathcal{A} \text{ succeeds}] \leq \Pr[&\text{pebbling is illegal}] \\ &+ \Pr[\text{pebbling uses} > S^* \text{ pebbles}] \\ &\qquad + \Pr[\text{pebbling uses} > T^* \text{ time steps}].\end{aligned}$$

Substituting in the probabilities of each of these events derived from Theorem 4, we find

$$\Pr[\mathcal{A} \text{ succeeds}] \leq \frac{T}{2^k} + \frac{1}{2^k} = \frac{T+1}{2^k}.$$

□

B.3 Dangers of the Pebbling Paradigm

The beauty of the pebbling paradigm is that it allows us to reason about the memory-hardness of certain functions by simply reasoning about the properties of graphs. That said, applying the pebbling model requires some care. For example, it is common practice to model an infinite-domain hash function $H : \{0,1\}^* \to \{0,1\}^k$ as a random oracle and then to instantiate H with a concrete hash function (e.g., SHA-256) in the actual construction.

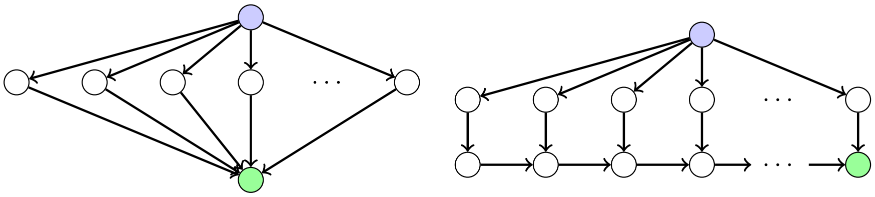


Fig. 7: A graph requiring $n+1$ pebbles to pebble in the random-oracle model (left) requires $O(1)$ storage to compute when using a Merkle-Damgård hash function (right).

When using a random oracle with an infinitely large domain in this way, the pebbling analysis can give misleading results. The reason is that Theorem 4 relies on the fact that when H is a random oracle, computing the value $H(x_1, \ldots, x_n)$

requires that the entire string $(x_1, \ldots, x_n)$ be written onto the oracle tape (i.e., be in memory) at the moment when the machine queries the oracle.

In practice, the hash function H used to construct the labeling of the pebble graph is *not* a random oracle, but is often a Merkle-Damgård-style hash function [32, 58] built from a two-to-one compression function $C : \{0,1\}^{2k} \rightarrow \{0,1\}^k$ as

$$H(x_1, \ldots, x_n) = C(x_n, C(x_{n-1}, \ldots, C(x_1, \perp) \ldots)).$$

If H is one such hash function, then the computation of $H(x_1, \ldots, x_n)$ requires at most a *constant number of blocks of storage* on the work and oracle tapes at any moment, since the Merkle-Damgård hash can be computed incrementally.

The bottom line is that pebbling lower bounds suggest that the labeling of certain graphs, like the one depicted in Figure 7, require $\Theta(n)$ blocks of storage to compute with high probability in the random oracle model. However, when H is a real Merkle-Damgård hash function, these functions actually take $\tilde{O}(1)$ space to compute. The use of incrementally computable compression functions has led to actual security weaknesses in candidate memory-hard functions in the past [20, Section 4.2], so these theoretical weaknesses have bearing on practice.

This failure of the random oracle model is one of the very few instances in which a *practical* scheme that is proven secure in the random-oracle model becomes insecure after replacing the random oracle with a concrete hash function (other examples include [12, 27, 45, 60]). While prior works study variants of the Merkle-Damgård construction that are indifferentiable from a random oracle [30], they do not factor these space usage issues into their designs.

To sidestep this issue entirely, we use the random oracle only to model compression functions with a fixed finite domain (i.e., two-to-one compression functions) whose internal state size is as large as their output size. For example, we model the compression function of SHA-512 as a random oracle, but do not model the entire [infinite-domain] SHA-512 function as a random oracle. When we use a sponge function, like SHA-3 [18], we use it as a two-to-one compression function, in which we extract only as many bits of output as the capacity of the sponge.

C A Combinatorial Lemma on "Well-Spread Sets"

In this section, we prove a combinatorial lemma that we will need for the analysis of the Balloon functions.

Let $X = \{x_1, x_2, \ldots, x_m\}$ be a multiset of integers written in non-decreasing order, such that $1 \leq x_i \leq n$ for all $x_i \in X$. The elements of X partition the integers from 0 to n into a collection of segments

$$(0, x_1],\ (x_1, x_2],\ (x_2, x_3],\ \ldots\ ,\ (x_{m-1}, x_m].$$

If we remove some subset of the segments, we can ask: what is the total length of the remaining segments? More formally, fix some set $S \subseteq X$. Then, define the *spread* of the multiset X after removing S as the sum of the lengths of the segments defined by X whose rightmost endpoints are not in S. Symbolically, we can let $x_0 = 0$ and define the spread as:

$$\text{spread}_S(X) = \sum_{x_i \in X \setminus S} (x_i - x_{i-1}).$$

For example, we have that:

$$\begin{aligned}
&\text{spread}_{\{\}}(\{2, 2, 4, 7\}) = (2 - 0) + (2 - 2) + (4 - 2) + (7 - 4) = 7 \\
&\text{spread}_{\{7\}}(\{2, 2, 4, 7\}) = (2 - 0) + (2 - 2) + (4 - 2) = 4 \\
&\text{spread}_{\{2,7\}}(\{2, 2, 4, 7\}) = (4 - 2) = 2 \\
&\text{spread}_{\{2,4,7\}}(\{2, 2, 4, 7\}) = 0
\end{aligned}$$

In computing $\text{spread}_S(\cdot)$, we remove all segments whose right endpoint falls into the set S. If there are many such segments (i.e., as when we compute $\text{spread}_{\{2\}}(\{2, 2, 4, 7\}) = 5$), we remove all of them.

We say that a multiset of integers is a *well-spread set* if the spread of the set is larger than a certain threshold, even after removing any subset of segments of a particular size.

Definition 6 (Well-Spread Set). Let X be a multiset of integers $X \subseteq \{1, \dots, n\}$ and let σ be an integer such that $\sigma \leq |X|$. Then X is a (σ, ρ)-*well-spread set* if for all subsets $S \subseteq X$ of size σ, $\text{spread}_S(X) \geq \rho$.

So, for example, a set X of integers is $(n/8, n/4)$-well-spread if, for all sets $S \subseteq X$ of size $n/8$, we have that $\text{spread}_S(X) \geq n/4$.

The following lemma demonstrates that a randomly sampled set of integers induces a well-spread set with all but negligible probability.

Lemma 7 (Big Spread Lemma) *For all constants $\delta > 1$ and $\omega > 0$, positive integers n, and positive integers $m < (1 - 2^{-\omega})n$, a multiset of δm elements sampled independently and uniformly at random from $\{1, \dots, n\}$ is an $(m, n/2^{\omega})$-well-spread set with probability at least $1 - 2^{[(1-\omega)\delta+\omega]m}$.*

Lemma 7 states (roughly) that if you throw δm balls into n bins, then remove the m longest runs of empty bins, there are still at least $n/2^{\omega}$ bins left over, with very high probability. For example, if we take $\delta = 3$ and $\omega = 2$, the lemma states that if you throw $3m$ balls into n bins (such that $m < 3n/4$) and then remove the m longest runs of empty bins, there are still at least $n/4$ bins left over, with very high probability.

To prove Lemma 7, we need one related lemma. In the hypothesis of the following lemma, we divide the integers from 1 to n into k segments at random, with the ith segment having length L_i. The lemma states that if f is an arbitrary function of the lengths of the segments, then the probability that f takes on value 1 is invariant under a reordering of the segments.

Lemma 8 *Sample a set of k integers independently and uniformly at random from $\{1, \dots, n\}$. Write the elements of the set in non-decreasing order as*

$(x_1, x_2, x_3, \ldots, x_k)$. *For all* $1 \le i \le k$, *define a random variable* $L_i = x_i - x_{i-1}$, *with* $x_0 = 0$. *Then for all functions* $f : \mathbb{Z}^k \to \{0,1\}$ *and all permutations* π *on* k *elements,*

$$\Pr[\, f(L_1, \ldots, L_k) = 1\,] = \Pr[\, f(L_{\pi(1)}, \ldots, L_{\pi(k)}) = 1\,].$$

To illustrate the meaning of the lemma, consider as an example the function $f_{(L_3 > L_4)}$ that takes the value "1" if the third segment is larger than the fourth segment and that takes the value "0" otherwise. The lemma implies that the probability that $f_{(L_3 > L_4)}$ takes on the value "1" is equal to the probability that the first segment is longer than the second segment.

Proof of Lemma 8. For convenience, write $L = (L_1, \ldots, L_k)$. For a permutation π on k elements, let $\pi(L) = (L_{\pi(1)}, \ldots, L_{\pi(k)})$. The random variable L can take on value $\ell \in \mathbb{Z}^k$ for any ℓ such that $\sum_i \ell_i \le n$. We can rewrite the probability $\Pr[f(L) = 1]$ by summing over the possible values $\ell \in \mathbb{Z}^k$:

$$\begin{aligned} \Pr[\, f(L) = 1\,] &= \sum_\ell \Pr[\, f(L) = 1 \,|\, L = \ell\,] \cdot \Pr[L = \ell], \\ &= \sum_\ell \Pr[\, f(\pi(L)) = 1 \,|\, \pi(L) = \ell\,] \cdot \Pr[L = \ell]. \end{aligned} \tag{2}$$

In the second step, we just renamed the variables on the right-hand side.

Now, we claim that $\Pr[L = \ell] = \Pr[\pi(L) = \ell]$. The claim holds because there is a one-to-one correspondence between outcomes for which $L = \ell$ and outcomes for which $\pi(L) = \ell$. In particular, for each set of integers $X = (x_1, x_2, \ldots, x_k)$ for which $L = \ell$, there is a corresponding set $X' = (x'_1, x'_2, \ldots, x'_k)$ for which $\pi(L) = \ell$, and vice versa. We can write out the correspondence as follows:

$$\begin{array}{llll} X = \{\ell_1, & \ell_1 + \ell_2, & \ell_1 + \ell_2 + \ell_3, & \ldots\} \\ X' = \{\ell_{\pi^{-1}(1)}, & \ell_{\pi^{-1}(1)} + \ell_{\pi^{-1}(2)}, & \ell_{\pi^{-1}(1)} + \ell_{\pi^{-1}(2)} + \ell_{\pi^{-1}(3)}, & \ldots\}. \end{array}$$

Thus, there are an equal number of outcomes for which $L = \ell$ and $\pi(L) = \ell$, and we conclude that $\Pr[L = \ell] = \Pr[\pi(L) = \ell]$.

Finally, we substitute this last equation into (2) to get:

$$\begin{aligned} \Pr[\, f(L) = 1\,] &= \sum_\ell \Pr[\, f(\pi(L)) = 1 \,|\, \pi(L) = \ell\,] \cdot \Pr[\pi(L) = \ell] \\ &= \Pr[\, f(\pi(L)) = 1\,]. \end{aligned}$$

□

Having proved Lemma 8, we can return to prove Lemma 7.

Proof of Lemma 7. The set X partitions the integers from 1 to n into δm segments. (We ignore the rightmost segment, since we do not count it in the spread.) Let L_i be a random variable denoting the length of the ith segment.

The bad event B that we want to avoid is the event that there exists some set of m segments whose collective length is greater than $(1-2^{-\omega})n$. To bound $\Pr[B]$, we can bound the probability that there exists a subset $S \subseteq \{1,\dots,\delta m\}$ of m segments such that

$$\sum_{i \in S} L_i \geq (1-2^{-\omega})n.$$

Fix a subset $S \subseteq \{1,\dots,\delta m\}$ of size m. Let $f : \mathbb{Z}^{\delta m} \to \{0,1\}$ be a function that outputs 1 if its first m arguments sum to at least $(1-2^{-\omega})n$, and that outputs 0 otherwise. Let π be a permutation on δm elements such that if $i \in S$, then L_i appears as one of the first m elements of $(L_{\pi(1)},\dots,L_{\pi(\delta m)})$. Then,

$$\begin{aligned}\Pr\left[\sum_{i \in S} L_i \geq (1-2^{-\omega})n\right] &= \Pr[\, f(L_{\pi(1)},\dots,L_{\pi(\delta m)}) = 1\,]\\ &= \Pr[\, f(L_1,\dots,L_{\delta m}) = 1\,]\\ &= \Pr[\, L_1 + \cdots + L_m \geq (1-2^{-\omega})n\,].\end{aligned}$$

We applied Lemma 8 to derive the second equality above.

The last probability on the right-hand side is relatively easy to compute. As long as $m < (1-2^{-\omega})n$, the event $L_1 + L_2 + ... + L_m \geq (1-2^{-\omega})n$ can only happen when $(\delta m - m)$ of the balls fall in the $2^{-\omega}n$ rightmost bins. (Otherwise the first m segments would have length less than $(1-2^{-\omega})n$.)

The probability of $(\delta m - m)$ balls falling in the rightmost $2^{-\omega}n$ bins is at most $(2^{-\omega})^{(\delta m - m)}$, so

$$\Pr[L_1 + L_2 + ... + L_m \geq (1-2^{-\omega})n] \leq \left(\frac{1}{2}\right)^{\omega(\delta m - m)}.$$

Then we apply the Union Bound over all $\binom{\delta m}{m}$ possible size-m subsets S of segments that could be large to get:

$$\begin{aligned}\Pr[B] &\leq \sum_{\text{sets } S} \Pr[L_1 + L_2 + ... + L_m \geq (1-2^{-\omega})n]\\ &\leq \binom{\delta m}{m}\left(\frac{1}{2}\right)^{\omega(\delta-1)m} \leq 2^{\delta m}\left(\frac{1}{2}\right)^{\omega(\delta-1)m} \leq 2^{[(1-\omega)\delta+\omega]m}.\end{aligned}$$

□

D Sandwich Graphs

In this section we recall the definition of *sandwich graphs* [8], and introduce a few transformations on sandwich graphs that are useful for our analysis of the Balloon functions.

D.1 Definitions

Definition 9 (Sandwich Graph [8]). A *sandwich graph* is a directed acyclic graph $G = (U \cup V, E)$ on $2n$ vertices, which we label as $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$. The edges of G are such that

- there is a (u_i, u_{i+1}) edge for $i = 1, \ldots, n-1$,
- there is a (u_n, v_1) edge,
- there is a (v_i, v_{i+1}) edge for $i = 1, \ldots, n-1$, and
- all other edges cross from U to V.

Figure 9 (left) depicts a sandwich graph. If $G = (U \cup V, E)$ is a sandwich graph, we refer to the vertices in U as the "top" vertices and vertices in V as the "bottom" vertices.

Definition 10 (Well-Spread Predecessors). Let $G = (U \cup V, E)$ be a sandwich graph and fix a subset of vertices $V' \subset V$. Write the immediate predecessors of V' in U as $P = \{u_{i_1}, u_{i_2}, \ldots, u_{i_{|P|}}\}$. Then we say that the predecessors of V' are (σ, ρ)-*well spread* if the corresponding set of integers $\{i_1, i_2, \ldots, i_{|P|}\}$ is (σ, ρ)-well spread, in the sense of Definition 6.

Definition 11 (Avoiding Set). Let $G = (U \cup V, E)$ be a directed acyclic graph. We say that the subset $V' \subset V$ is a (σ, ρ)-*avoiding set* if, after placing σ pebbles anywhere on the graph, except on V', there are at least ρ distinct vertices in U on unpebbled paths to V'.[16]

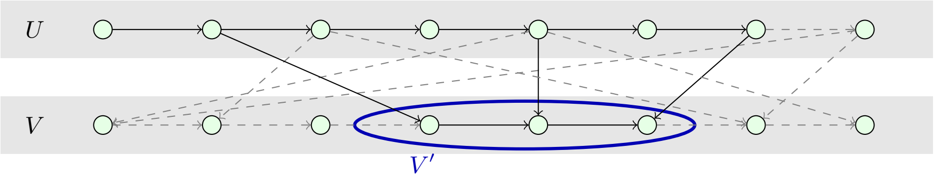


Fig. 8: The set V' is a $(1, 4)$-avoiding set: after placing any single pebble on the graph (except on vertices in V'), there will still be at least four vertices in U on unpebbled paths to V'.

Figure 8 gives an example of an avoiding set.

Lemma 12 *Let $G = (U \cup V, E)$ be a sandwich graph and let $V \subset V'$. If the predecessors of V' are (σ, ρ)-well spread, then V' is a (σ, ρ)-avoiding set.*

[16] More formally: for all possible placements of σ vertices on the graph G except on V', there exists a size-ρ set of vertices $U' \subseteq U$ such that for all vertices $u \in U'$, there exists a vertex $v \in V'$ and a u-to-v path p in G such that no vertex in the path p contains a pebble.

Proof. We can think of the p predecessors of vertices in V' as dividing the chain of vertices in U (the "top half of the sandwich") into p smaller sub-chains of vertices. If the predecessors of V' are (σ, ρ)-well spread then, after removing any σ of these sub-chains, the remaining chains collectively contain ρ vertices. We know that there can be at most σ pebbles on vertices in U, so at most σ sub-chains contain a pebbled vertex. The remaining sub-chains, collectively containing ρ vertices, contain no pebbled vertices. The ρ vertices in these unpebbled sub-chains all are on unpebbled paths to V', which proves the lemma. □

Definition 13 (Everywhere-Avoiding Graph). Let $G = (U \cup V, E)$ be a directed acyclic graph. The graph G is a (σ, ρ)-*everywhere avoiding graph* if for every subset $V' \subset V$ such that $|V'| = \sigma$, the subset V' is a (σ, ρ)-avoiding set.

Definition 14 (Consecutively-Avoiding Graph). Let $G = (U \cup V, E)$ be a directed acyclic graph, with vertices in V labeled v_1 to v_n. The graph G is a (κ, σ, ρ)-*consecutively avoiding graph* if every subset $V' \subset V$ of consecutive vertices of size κ is (σ, ρ)-avoiding.

D.2 Transformations on Sandwich Graphs

Sandwich graphs are useful in part because they maintain certain connectivity properties under a "localizing" transformation. Let G be a sandwich graph. Let $(a_1, \dots, a_n)$ be the top vertices of the graph G and let $(a_{n+1}, \dots, a_{2n})$ be the bottom vertices. The localized graph $\mathcal{L}(G)$ on vertices $(a_1, \dots, a_{2n})$ has the property that every predecessor of a vertex a_i falls into the set $\{a_{\max\{1,i-n\}}, \dots, a_{i-1}\}$. (In the unlocalized graph, a_i's predecessors fell into the larger set $\{a_{\max\{1,i-2n\}}, a_{i-1}\}$.) If we think of the set $\{a_{\max\{1,i-n\}}, \dots, a_{i-1}\}$ as the vertices "nearby" to vertex a_i, then the localizing transformation ensures that the predecessors of every vertex a_i all fall into this set of nearby vertices. Figure 9 demonstrates this transformation.

We use this localizing transformation to make more efficient use of buffer space in the Balloon algorithm. It is possible to pebble a localized sandwich graph in linear time with $n + O(1)$ pebbles, whereas a non-localized sandwich graph can require as many as $2n$ pebbles in the worst case. This transformation makes computing the Balloon function easier for anyone using n space, while maintaining the property that the function is hard to compute in much less space. (Smith and Zhang find a similar locality property useful in the context of leakage-resilient cryptography [82].)

Definition 15. Let $G = (U \cup V, E)$ be a sandwich graph with $U = (u_1, \dots, u_n)$, $V = (v_1, \dots, v_n)$. The *localized graph* $\mathcal{L}(G) = (\mathcal{L}(V) \cup \mathcal{L}(U), \mathcal{L}(E))$ has top and bottom vertex sets $\mathcal{L}(U) = \{\tilde{u}_1, \dots, \tilde{u}_n\}$ and $\mathcal{L}(V) = \{\tilde{v}_1, \dots, \tilde{v}_n\}$, and an edge set

$$\mathcal{L}(E) = \begin{array}{l} \{(\tilde{u}_i, \tilde{v}_i) \mid i \in \{1, \ldots, n\}\} \cup \\ \{(\tilde{u}_i, \tilde{u}_{i+1}) \mid i \in \{1, \ldots, n-1\}\} \cup \\ \{(\tilde{v}_i, \tilde{v}_{i+1}) \mid i \in \{1, \ldots, n-1\}\} \cup \\ \{(\tilde{u}_n, \tilde{v}_1)\} \cup \\ \{(\tilde{v}_i, \tilde{v}_j) \mid (u_i, v_j) \in E \text{ and } i \leq j\} \cup \\ \{(\tilde{u}_i, \tilde{v}_j) \mid (u_i, v_j) \in E \text{ and } i > j\}. \end{array}$$

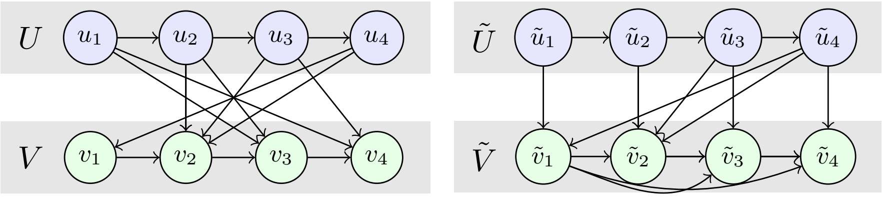


Fig. 9: A sandwich graph G (left) and the corresponding localized graph $\mathcal{L}(G)$ (right).

Claim 16 *Let $G = (U \cup V, E)$ be a sandwich graph and let $V' \subset V$ be a subset whose predecessors are (σ, ρ)-well spread. Let $\mathcal{L}(V')$ be the vertices corresponding to V' in the localized graph $\mathcal{L}(G)$. Then $\mathcal{L}(V')$ is a (σ, ρ)-avoiding set.*

Proof. Fix a pebbling of $\mathcal{L}(G)$ using σ pebbles with no pebbles on V'. For every edge $(u_i, v_j) \in U \times V$ in G, there is either (a) a corresponding edge in $\mathcal{L}(G)$, or (b) a pair of edges (u_i, v_i) and (v_i, v_j) in $\mathcal{L}(G)$. If the vertex v_i does not contain a pebble, then for analyzing the avoiding-set property, we can consider there to exist a (u_i, v_j) edge. There are now at most σ pebbled U-to-V' edges. By Lemma 12, the (σ, ρ)-avoiding set property follows. □

Corollary 17 *If G is a sandwich graph such that every subset of σ vertices $V' \subset V$ is (σ, ρ)-well-spread, then $\mathcal{L}(G)$ is a (σ, ρ)-everywhere avoiding graph.*

Corollary 18 *If G is a sandwich graph such that every subset of κ vertices $V' \subset V$ is (σ, ρ)-well-spread, then then $\mathcal{L}(G)$ is a (κ, σ, ρ)-consecutively avoiding graph.*

The next set of graph transformations we use allows us to reduce the in-degree of the graph from δ down to 2 without affecting the key structural properties of the graph. Reducing the degree of the graph allows us to instantiate our construction with a standard two-to-one compression function and avoids the issues raised in Appendix B.3. The strategy we use follows the technique of Paul and Tarjan [66].

Definition 19. Let $G = (U \cup V, E)$ be a (possibly localized) sandwich graph. We say that the *degree-reduced graph* $\mathcal{D}(G)$ is the graph in which each vertex $v_i \in V$ in G of in-degree $\delta+1$ is replaced with a path "gadget" whose vertices have in-degree at most 2. The original predecessor vertex v_{i-1} is at the beginning of the path, there are $\delta - 1$ internal vertices on the path, and the original vertex v_i is at the end of the path. The δ other predecessors of v_i are connected to the vertices of the path (see Figure 10).

By construction, vertices in $\mathcal{D}(G)$ have in-degree at most two. If G is a sandwich graph on $2n$ vertices, then $\mathcal{D}(G)$ still has n "top" vertices and n "bottom" vertices. If the graph G had out-degree at most δ, then the vertex and edge sets of $\mathcal{D}(G)$ are at most a factor of $(\delta - 1)$ larger than in G, since each gadget has at most $(\delta - 1)$ vertices. The degree-reduced graph $\mathcal{D}(G)$ has extra "middle" vertices (non-top non-bottom vertices) consisting of the internal vertices of the degree-reduction gadgets (Figure 10).

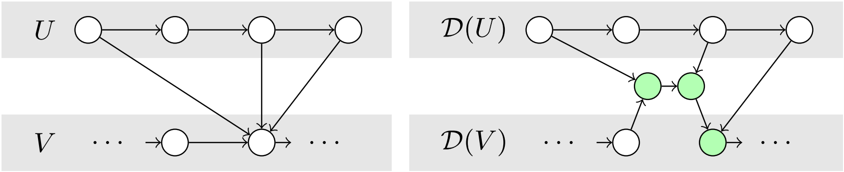


Fig. 10: A portion of a sandwich graph G (left) and the same portion of the corresponding degree-reduced graph $\mathcal{D}(G)$ (right). The shaded vertices are part of the degree-reduction gadget.

Claim 20 *Let $G = (U \cup V, E)$ be a (possibly localized) sandwich graph and let $\mathcal{D}(G)$ be the corresponding degree-reduced graph. Let $V' \subset V$ be a (σ, ρ)-avoiding set in G, let $\mathcal{D}(V')$ be the vertices corresponding to V'* and *the degree-reduction gadgets attached to vertices in V'. Then $\mathcal{D}(V')$ is a (σ, ρ)-avoiding set.*

Proof. For every pebbling of vertices in $\mathcal{D}(G)$ that violates the avoiding property of $\mathcal{D}(V')$, there is a corresponding pebbling in G that violates the avoiding property of V'. To convert a pebbling of $\mathcal{D}(G)$ into a pebbling of G, just place a pebble on every vertex in G whose corresponding degree-reduction gadget in $\mathcal{D}(G)$ has a pebble. The claim follows. □

Corollary 21 *If G is a (σ, ρ)-everywhere avoiding sandwich graph, then $\mathcal{D}(G)$ is "almost" a (σ, ρ)-everywhere avoiding graph, in the sense that every subset of σ degree-reduction gadgets is a (σ, ρ)-avoiding set.*

Corollary 22 *If G is a (κ, σ, ρ)-consecutively avoiding sandwich graph, then $\mathcal{D}(G)$ is "almost" (κ, σ, ρ)-consecutively avoiding graph, in the sense that every subset of σ degree-reduction gadgets is a (σ, ρ)-avoiding set.*

D.3 Pebbling Sandwich Graphs

Lemma 23 *Let $G = (U \cup V, E)$ be a (κ, σ, ρ)-consecutively-avoiding sandwich graph on $2n$ vertices. Let $\mathcal{M}$ be a legal sequence of pebbling moves that begins with no pebbles on the bottom half of the graph and that pebbles the topologically last vertex in G at some point. Then we can divide $\mathcal{M}$ into $L = n/(2\kappa)$ subsequences of legal pebbling moves $\mathcal{M}_1, \ldots, \mathcal{M}_L$ such that each subsequence $\mathcal{M}_i$ pebbles at least ρ unpebbled vertices in U.*

The proof follows the idea of Lengauer and Tarjan's analysis of pebbling strategies for the "bit-reversal" graph [54].

Proof. Label the vertices in V in topological order as $(v_1, \ldots, v_n)$. Divide these vertices into $\lfloor n/\kappa \rfloor$ intervals of size κ. The last vertex in each of the intervals is then: $v_\kappa, v_{2\kappa}, v_{3\kappa}, \ldots, v_{\lfloor n/\kappa \rfloor \kappa}$.

Consider any legal sequence of pebbling moves $\mathcal{M}$ of the last vertex in G. Let $t_0 = 0$ and let t_i be the time step at which vertex $v_{i\kappa}$ (the last vertex in the ith interval) first receives a pebble. We know that at t_{i-1}, there are no pebbles on the ith interval—this is because, by the structure of a sandwich graph, all of the pebbles in V must be pebbled in order. Thus, between t_{i-1} and t_i, every dependency of the vertices in interval i must receive a pebble.

These are κ consecutive vertices. Since G is (κ, σ, ρ)-consecutively avoiding, the vertices in interval i are a (σ, ρ)-avoiding set. By the definition of an avoiding set, at time t_{i-1} there must be at least ρ unpebbled dependencies in U of the vertices in the ith interval. All of these vertices must receive pebbles by time t_i. Thus, in each time interval, the $\mathcal{M}$ must pebble a set of ρ unpebbled vertices in U.

We have that $\kappa \leq n$, so $\lfloor n/\kappa \rfloor \geq n/(2\kappa)$, so there are at least $n/(2\kappa)$ sets of ρ unpebbled vertices in U that receive pebbles during the pebbling.

We now let the subsequence $\mathcal{M}_i$ defined in the lemma be the sequence of pebbling moves between t_{i-1} and t_i, and the lemma follows. □

Corollary 24 *Pebbling a (κ, σ, ρ)-consecutively avoiding (possibly localized) sandwich graph with σ pebbles, starting with no pebbles on the bottom half of the graph requires at least $\frac{\rho n}{2\kappa}$ pebbling moves.*

Proof. By Lemma 23, the pebbling pebbles at least ρ unpebbled vertices at least $n/(2\kappa)$ times, so the total time required must be at least $\frac{\rho n}{2\kappa}$. □

A key piece of our analysis involves "stacks" of everywhere-avoiding sandwich graphs. Given a sandwich graph with n top vertices and n bottom vertices, we can stack the graphs by making the bottom nodes of the zero-th copy of the graph, the top nodes of the first copy of the graph (see Figure 11).

Lemma 25 *Let G be a depth-d stack of $(\sigma, 2\sigma)$-everywhere avoiding sandwich graphs. Let V' be a set of σ level-d vertices in the graph. Fix a configuration $\mathcal{C}$ of at most σ pebbles anywhere on the graph, except that there are no pebbles on vertices in V'. Then if $\mathcal{M}$ is a sequence of legal pebbling moves such that*

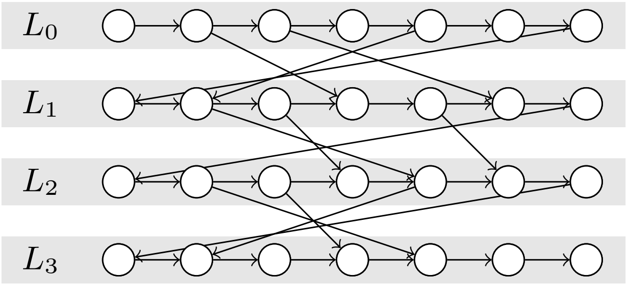


Fig. 11: A stack of $d = 3$ sandwich graphs. The top vertices of the stack are at level L_0 and the bottom vertices are at level L_3.

- *$\mathcal{M}$ beings in configuration $\mathcal{C}$,*
- *$\mathcal{M}$ at some point places a pebble on every vertex in V', and*
- *$\mathcal{M}$ never uses more than σ pebbles,*

then $\mathcal{M}$ must consist of at least $2^d\sigma$ pebbling moves.

Proof. By induction on the depth d.

Base case ($d = 1$). By the fact that the graph G is $(\sigma, 2\sigma)$-everywhere avoiding, the σ vertices in the set V' must have at least 2σ unpebbled dependencies on the top level of the graph. These unpebbled predecessors of vertices in V' all must receive pebbles during $\mathcal{M}$, so $\mathcal{M}$ must contain at least 2σ moves.

Induction Step. As in the base case, we know that there are at least 2σ unpebbled level-$(d-1)$ dependencies of V' that must receive pebbles during $\mathcal{M}$. Now we can divide $\mathcal{M}$ into two sub-sequences of consecutive pebbling moves: $\mathcal{M} = \mathcal{M}_1 \| \mathcal{M}_2$. We divide the pebbling moves such that $\mathcal{M}_1$ consists of the moves during which the first σ of the 2σ unpebbled dependencies receive pebbles, and $\mathcal{M}_2$ consists of the rest of the moves.

Note now that the induction hypothesis applies to both $\mathcal{M}_1$ and $\mathcal{M}_2$:

- Each set of moves begins in a configuration of at most σ pebbles on the graph.
- Each set of moves pebbles a set of σ initially unpebbled level-$(d-1)$ vertices.
- Each set of moves never uses more than σ pebbles.

Thus the total number of pebbling moves required in $\mathcal{M}$ is at least $2 \cdot (2^{d-1}\sigma)$, which proves the lemma. □

Remark 26 *The arguments of Lemmata 23 and 25 apply also to localized and degree-reduced sandwich graphs, by Corollaries 17, 18, 21, and 22.*

D.4 Random Sandwich Graphs

Now we introduce two special types of sandwich graphs that we use in the analysis of the Balloon function. Alwen and Blocki [3] use the same type of sandwich graphs in their analysis (indeed, their work has inspired our use of sandwich graphs), though the results we prove here are new.

Definition 27 (δ-Random Sandwich Graph). A *δ-random sandwich graph* is a sandwich graph $G = (U \cup V, E)$ such that each vertex in V has δ predecessors sampled independently and uniformly at random from the set U.

Lemma 28 *Fix integers $\delta \geq 3$ and $\omega \geq 2$. Let $G = (U \cup V, E)$ be a δ-random sandwich graph on $2n$ vertices. Then for all positive integers n_0, G is an $(m, m, n/2^{\omega})$-consecutively avoiding graph for all m such that $n_0 \leq m < (1 - 2^{-\omega})n$, except with probability $2n \cdot c_{28}(\delta, \omega)^{n_0}$, where the value of $c_{28}(\cdot, \cdot) < 1$ depends only on δ and ω.*

Proof. The predecessors of a set of $m = |V'|$ vertices are δm vertices i.u.r. sampled from the set of n "top" vertices. The probability that a single consecutive set of m vertices induces a poorly spread set of predecessors is at most $2^{[(1-\omega)\delta+\omega]|V'|}$, by Lemma 7. We then apply the Union Bound over all sets of m *consecutive vertices* to find the probability that there exists a bad set. For each choice of m, there are at most n sets V' of consecutive vertices, so

$$\Pr[\exists \text{ bad set}] \leq \sum_{i=1}^{n} \sum_{m=1}^{n} (2^{[(1-\omega)\delta+\omega]})^m \leq n \sum_{m=n_0}^{\infty} (2^{[(1-\omega)\delta+\omega]})^m.$$

This is just a geometric series with ratio $r = 2^{[(1-\omega)\delta+\omega]}$.

$$\Pr[\exists \text{ bad set}] \leq n \cdot \frac{r^{n_0}}{1-r} \leq n \cdot \frac{2^{[(1-\omega)\delta+\omega]n_0}}{1 - 2^{[(1-\omega)\delta+\omega]}}$$

Since $\delta \geq 3$ and $\omega \geq 2$, the denominator is at least $1/2$, so

$$\Pr[\exists \text{ bad set}] \leq 2n \cdot 2^{[(1-\omega)\delta+\omega]n_0}.$$

Taking $c_{28}(\delta, \omega) = 2^{[(1-\omega)\delta+\omega]}$ proves the lemma. □

Lemma 29 *For every $\omega \geq 2$, there exists constants $\delta \geq 1$ and $c < 1$ such that a δ-random sandwich graph on $2n$ vertices is an $(n/2^{\omega+1}, n/2^{\omega})$-everywhere avoiding graph, with probability at least $1 - c_{29}(\delta, \omega)^n$, for some function $c_{29}(\cdot, \cdot) < 1$.*

Proof. The probability that a single set $V' \subset V$ in G induces a set that is not $(n/2^{\omega+1}, n/2^{\omega})$-well spread is at most $2^{[(1-\omega)\delta+\omega]n/2^{\omega+1}}$, by Lemma 7. We then apply the Union Bound over sets V' of size $n/2^{\omega+1}$ to find the probability that there exists a bad set V':

$$\begin{aligned}
\Pr[\exists \text{ bad set}] &\leq \binom{n}{n/2^{\omega+1}} \cdot 2^{[(1-\omega)\delta+\omega]\frac{n}{2^{\omega+1}}} \\
&\leq \left(\frac{n}{n/2^{\omega+1}}\right)^{\frac{n}{2^{\omega+1}}} \cdot 2^{[(1-\omega)\delta+\omega]\frac{n}{2^{\omega+1}}} \\
&\leq \left(e \cdot 2^{\omega+1}\right)^{\frac{n}{2^{\omega+1}}} \cdot 2^{[(1-\omega)\delta+\omega]\frac{n}{2^{\omega+1}}} \\
&\leq \left[\left(e \cdot 2^{\omega+1}\right) \cdot 2^{[(1-\omega)\delta+\omega]}\right]^{\frac{n}{2^{\omega+1}}} \\
&\leq \left[2^{(1-\omega)\delta+2\omega+\log_2 e+1}\right]^{\frac{n}{2^{\omega+1}}}.
\end{aligned}$$

Here, we used the standard inequality $\binom{n}{k} \leq (\frac{ne}{k})^k$. If we select δ such that:

$$\frac{2\omega + \log_2 e + 1}{\omega - 1} < \delta,$$

then we have that the probability that there exists a bad set is at most $c_{29}(\delta, \omega)^n$ for

$$c_{29}(\delta, \omega) = 2^{\frac{(1-\omega)\delta+2\omega+\log_2 e+1}{2^{\omega+1}}}.$$

This proves the lemma. □

Lemma 30 *Let $G_{n,d}$ be a depth-d stack of δ-random sandwich graphs with n vertices on each level. Then for any integer $n_0 > 0$,*

- *every sandwich graph in the stack is an $(m, m, n/4)$-consecutively avoiding graph, for all $n_0 \leq m \leq 3n/4$, and*
- *every sandwich graph in the stack is $(n/2^{\omega+1}, n/2^{\omega})$-everywhere avoiding,*

except with probability:

$$p_{fail}(\delta, \omega, n_0, n, d) \leq 2nd \cdot c_{28}(\delta, 2)^{n_0} + d \cdot c_{29}(\delta, \omega)^n.$$

Proof. The probability that $G_{n,d}$ does not satisfy the first property is at most $p_1 \leq 2nd \cdot c_{28}(\delta, 2)^{n_0}$ by Lemma 28 and a Union Bound over the d levels of the graph. The probability that $G_{n,d}$ does not satisfy the second property is at most $p_2 \leq d \cdot c_{29}(\delta, \omega)^n$ by Lemma 29 and an application of the Union Bound over the d levels of the graph. The probability that either bad event occurs is at most $p_{\text{fail}} \leq p_1 + p_2$, by the Union Bound. □

Lemma 31 *Let $G_{n,d}$ be a depth-d stack of δ-random sandwich graphs with n vertices on each level. Then, pebbling the topologically last vertex of $G_{n,d}$ with at most S pebbles requires time T such that:*

(a) $S \cdot T \geq dn^2/8$ for space usage $n_0 \leq S$, and
(b) $S \cdot T \geq (2^d - 1)n^2/8$ for space usage $n_0 \leq S < n/2^{\omega+1}$,

except with probability $p_{fail}(\delta, \omega, n_0, n, d)$, defined in Lemma 30.

Proof. Lemma 30 demonstrates that, except with probability p_{fail} defined in the lemma, the last sandwich graph in the stack is an $(S, S, n/4)$-consecutively avoiding graph, for all $n_0 \leq S \leq 3n/4$, and every sandwich graph in the stack is $(n/2^{\omega+1}, n/2^{\omega})$-everywhere avoiding,

Conditioned on this failure event not happening, we can prove the theorem by induction on d, the depth of the stack of sandwich graphs.

Base Case ($d = 1$). When $d = 1$, part (a) of the theorem implies part (b). Part (a) follows immediately by by Corollary 24: pebbling an $(S, S, n/4)$-consecutively avoiding graph with $n_0 \leq S \leq 3n/4$ pebbles requires at least $\frac{n^2}{8S}$ pebbling moves. Thus $S \cdot T \geq n^2/8$. Since for $S \geq n/8$, $T \geq n$, we have that $S \cdot T \geq n^2/8$ holds for all $S \geq n_0$.

Induction Step. Assume the theorem holds for stacks of depth at most $d - 1$. At the start of a pebbling of $G_{n,d}$, there are no pebbles on the graph.

To prove Part (a): To place a pebble the first pebble on level d of the graph, we must first place a pebble on the last vertex of level $d-1$ of the graph. By the induction hypothesis, this requires at least $(d-1)n^2/(8S)$ pebbling moves. Now there are no pebbles on the last level of the graph, by Corollary 24, pebbling an $(S, S, n/4)$-consecutively avoiding graph with $n_0 \leq S \leq 3n/4$ pebbles requires at least $\frac{n^2}{8S}$ pebbling moves. Thus

$$T \geq \frac{n^2}{8S} + \frac{(d-1)n^2}{8S} = \frac{dn^2}{8S},$$

and Part (a) is proved.

To prove Part (b): To place a pebble the first pebble on the last level of the graph, we must first place a pebble on the last vertex of level $d-1$ of the graph. By the induction hypothesis, this requires at least $T_{d-1} \geq (2^{d-1} - 1)n^2/(8S)$ pebbling moves, if $S < n/2^{\omega+1}$.

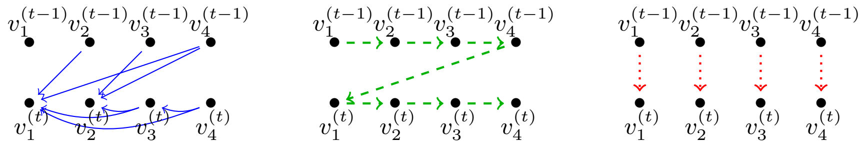
By Corollary 24, pebbling an $(S, S, n/4)$-consecutively avoiding graph that has no pebbles on the bottom half of the graph with $n_0 \leq S \leq 3n/4$ pebbles requires pebbling at least $n/(2S)$ subsets of $n/4$ vertices on the "top half" of the sandwich graph.

These $n/4$ vertices are the bottom vertices of a depth-$(d-1)$ stack of $(n/2^{\omega+1}, n/2^{\omega})$-everywhere avoiding graphs. By Lemma 25, pebbling any $n/2^{\omega+1}$ unpebbled vertices of a depth-$(d-1)$ stack of such graphs with at most $n/2^{\omega+1}$ pebbles requires at least $2^{d-1}n/2^{\omega+1}$ moves. Since there are $(n/4)/(n/2^{\omega+1}) = 2^{\omega+1}/4 = 2^{\omega-1}$ segments of such vertices, the total time required to pebble them is $2^{\omega-1}(2^{d-1}n/2^{\omega+1}) = 2^d n/8$.[17]

The total time to pebble each of the $n/(2S)$ segments of such vertices is then:

$$T_d \geq \frac{n}{2S} \cdot \frac{2^d n}{8} = \frac{2^{d-1}n^2}{8S}.$$

[17] More formally, we can divide any sequence of pebbling moves $\mathcal{M}$ that pebbles all of these $2^{\omega-1}$ segments into $2^{\omega-1}$ distinct sub-sequences of consecutive pebbling moves. By Lemma 25, each of these sub-sequences must contain at least $2^{d-1}n/2^{\omega+1}$ moves.



$v_1^{(t-1)}$ $v_2^{(t-1)}$ $v_3^{(t-1)}$ $v_4^{(t-1)}$
$v_1^{(t)}$ $v_2^{(t)}$ $v_3^{(t)}$ $v_4^{(t)}$
$v_1^{(t-1)}$ $v_2^{(t-1)}$ $v_3^{(t-1)}$ $v_4^{(t-1)}$
$v_1^{(t)}$ $v_2^{(t)}$ $v_3^{(t)}$ $v_4^{(t)}$
$v_1^{(t-1)}$ $v_2^{(t-1)}$ $v_3^{(t-1)}$ $v_4^{(t-1)}$
$v_1^{(t)}$ $v_2^{(t)}$ $v_3^{(t)}$ $v_4^{(t)}$

(a) Pseudo-random edges. (b) Backwards edges. (c) In-place edges.

Fig. 12: Components of the data-dependency graph for one Balloon mixing round. Here, $v_i^{(t)}$ represents the value stored in the ith block in the main memory buffer at the tth mixing round.

So the total time to place a pebble on the last vertex of the d-th level of the graph is:

$$T \geq T_d + T_{d-1} \geq \frac{2^{d-1}n^2}{8S} + \frac{(2^{d-1}-1)n^2}{8S} = \frac{(2^d-1)n^2}{8S},$$

and Part (b) is proved. □

E From Pebbling to Memory-Hardness

In this section, we complete the security analysis of the Balloon function. We present the formal proof of the claim that the space S and time T required to compute the r-round Balloon function satisfies (roughly) $S \cdot T \geq rn^2/8$. The other pieces of Informal Theorem 1 follow from repeated application of the same technique.

Claim 32 *The output of the n-block r-round Balloon construction is the labeling, in the sense of Definition 3, of a depth-r stack of localized and degree-reduced δ-random sandwich graphs.*

Proof. Follows by inspection of the Balloon algorithm (Figure 1). □

Theorem 33 (Formal statement of first part of Informal Theorem 1) *Let k denote the block size (in bits) of the underlying cryptographic hash function used in the Balloon constructions. Fix an integer $n_0 > 0$. Any algorithm $\mathcal{A}$ that computes the output of the n-block r-round Balloon construction (with security parameter $\delta = 7$), makes T random-oracle queries, and uses fewer than σ bits of storage space, such that*

$$T < \frac{rn^2}{8S^*} \quad \text{and} \quad \sigma < S^*\left(k - \log_2\left(\frac{rn^2}{8S^*}\right)\right) - k,$$

for some $n_0 \leq S^* < 3n/4$, then the probability that $\mathcal{A}$ succeeds is at most

$$\frac{T+1}{2^k} + p_{fail}(7, 2, n_0, n, r),$$

where $p_{fail}(\cdot,\cdot,\cdot,\cdot,\cdot)$ is defined in Lemma 30.

To make the failure probability concrete, we give an example: if the adversary can make $T = 2^{60}$ random-oracle queries, we use a random oracle with a $k = 512$-bit output size, we fix $n_0 = 128$, we use a buffer with $n = 2^{14}$ blocks, and we run the Balloon hashing algorithm for $r = 8$ rounds, then adversary $\mathcal{A}$'s success probability in Theorem 33 is much smaller than 2^{-256}.

Proof of Theorem 33. Fix an algorithm $\mathcal{A}$. By Claim 32, $\mathcal{A}$ outputs the labeling of a depth-r stack of δ-random sandwich graphs. By Lemma 31, shows that pebbling these graphs with $S \geq n_0$ pebbles takes time at least $n^2/(8S)$, except with some small probability. Let B denote this event that the pebbling bound does not hold.

Then we have:

$$\begin{aligned}\Pr[\mathcal{A}\text{ succeeds}] = \Pr[\mathcal{A}\text{ succeeds}|B]\cdot\Pr[B]+&\\ \Pr[\mathcal{A}\text{ succeeds}|\neg B]\cdot\Pr[\neg B]&\\ \leq \Pr[\mathcal{A}\text{ succeeds}|\neg B]+\Pr[B].&\end{aligned}$$

From Lemma 31, $\Pr[B] \leq p_{\text{fail}}(\delta, 2, n_0, n, r)$.

Conditioned on $\neg B$, we may use Lemma 31 in conjunction with Theorem 5 to show that $\Pr[\mathcal{A}\text{ succeeds}|\neg B]$ is small. In particular, for any $S^* \geq n_0$, there does not exist a pebbling of the graph that uses than $T^* = \frac{rn^2}{8S^*}$ pebbling moves.[18] We then apply Theorem 5 to conclude that $\Pr[\mathcal{A}\text{ succeeds}|\neg B] \leq (T+1)/2^k$. This completes the proof. □

The other parts of Informal Theorem 1 follow from a similar analysis.

F Argon2i Proof of Security

In this section, we show that the proof techniques we introduce for the analysis of the Balloon algorithm can also apply to prove the first known memory-hardness results on the Argon2i algorithm.

We focus here on the single-pass variant of Argon2i, described in Section 4, and do not attempt to generalize our results to the multi-pass variant. As in Section 4, we analyze an idealized version of Argon2i, in which the randomly chosen predecessor of a vertex is chosen using the uniform distribution over the topologically prior vertices.

Theorem 34 *Let A_n denote the single-pass Argon2i data-dependency graph on n blocks. Fix a positive integer n_0. Then pebbling A_n with $n_0 < S < n/24$ pebbles requires time T such that:*

$$S \cdot T \geq \frac{n^2}{192},$$

[18] The graph used in the Balloon construction is degree-reduced and localized, so it is not precisely a δ-random sandwich graph, but by Remark 26, the pebbling time-space lower bounds still apply to the degree-reduced and localized graph.

except with probability $n^2 2^{-n_0-1}$.

Note that the memory-hardness theorem that we are able to prove here about single-pass Argon2i is much weaker than the corresponding theorem we can prove about the Balloon algorithm (Informal Theorem 1). Essentially, this theorem says that an attacker computing single-pass Argon2i cannot save more than a factor of 192× in space without having to pay with some increase in computation cost.

For the purposes of the security analysis, it will be more convenient to look at the graph A_{2n}. Take the graph A_{2n} with and write its vertex set in topological order as: $(u_1, \dots, u_n, v_1, \dots, v_n)$. We can now think of the u vertices as the "top" vertices of the graph, and the v vertices as the "bottom" vertices in the graph. Then we have the following claim:

Claim 35 *Consider any set of* $12m$ *bottom vertices of the graph* A_{2n}*. These vertices are an* $(m, n/4)$*-avoiding set, in the sense of Definition 11, except with probability* 2^{1-m}*.*

Proof. Let v_i be some bottom vertex in A_{2n}. The vertex v_i has one predecessor (call it $\mathsf{pred}(v_i)$) chosen i.u.r. from the set $\{u_1, \dots, u_n, v_1, \dots, v_{i-1}\}$. Conditioned on the event that that $\mathsf{pred}(v_i) \in \{u_1, \dots, u_n\}$, this predecessor is chosen i.u.r. from the set of top vertices $\{u_1, \dots, u_n\}$. Let T_i be an indicator random variable taking on value "1" when vertex v_i's randomly chosen predecessor is a top vertex. Then $\Pr[T_i = 1] \geq 1/2$.

Fix a set V' of κm vertices on the bottom half of the graph. We want to show that, with high probability, V' has at least $3m$ predecessors in the top-half of the graph. The expected number of top-half predecessors of the set V' is at least $\kappa m/2$, and we want to bound the probability that there are at most $3m$ top-half predecessors. A standard Chernoff bound gives that, for κm independent Poisson trials with success probability at least $1/2$, the probability that fewer than $3m$ of them succeed is bounded by

$$\Pr\left[\sum_{v_i \in V'} T_i < 3m\right] < \exp\left(-\frac{(\frac{\kappa m}{2})(1-\frac{6}{\kappa})^2}{2}\right),$$

for $\kappa > 6$. Now taking $\kappa = 12$ gives:

$$\Pr\left[\sum_{v_i \in V'} T_i < 3m\right] < \exp\left(-\frac{3m}{4}\right) < 2^{-m}.$$

Now, by Lemma 7 (with $\delta = 3$ and $\omega = 2$), the probability that the predecessors of set V' are not $(m, n/4)$-well-spread over the top vertices is at most 2^{-m}. By the Union Bound, the probability that either V' has fewer than $3m$ i.u.r. top-half predecessors or that these predecessors are poorly spread is at most $2 \cdot 2^{-m} = 2^{1-m}$. By Lemma 12, the set V' is then an $(m, n/4)$-avoiding set, except with probability 2^{1-m}. □

Proof of Theorem 34. Fix an integer n_0. We show that A_{2n} is a $(12m, m, n/4)$-consecutively avoiding graph when $n_0 \leq m \leq n/12$, except with probability $n^2 2^{-n_0}$. The probability that any consecutive set of $12m$ vertices on the bottom level of the graph is not an $(m, n/4)$-avoiding set is at most $\sum_{i=1}^{n} \sum_{m=128}^{n} 2^{1-m} \leq 2n^2 2^{-n_0}$, using Claim 35 and the Union Bound.

Now, we apply Corollary 24 (with $\kappa = 12m$, $\sigma = m$, $\rho = n/4$) to conclude that pebbling A_{2n} with at most S pebbles requires at least $T \geq \frac{n^2}{48S}$ pebbling moves, when $n_0 < S < n/12$, except with probability $2n^2 2^{-n_0}$. Alternatively, we can write: $S \cdot T \geq n^2/48$.

Now, since we are interested in the complexity of pebbling A_n (not A_{2n}), we can divide the ns by two to find that pebbling A_n with at most S pebbles requires $S \cdot T \geq n^2/192$ pebbling moves, when $n_0 < S < n/24$, except with probability $n^2 2^{-n_0-1}$. □

We can convert the pebbling lower bound into a time-space lower bound in the random-oracle model using the techniques of Appendix E.

G Analysis of Scrypt

In this section, we show that the proof techniques we have used to analyze the memory-hardness of the Balloon algorithm are useful for analyzing scrypt.

G.1 Scrypt is Memory-Hard

In particular, we give a very simple proof that a simplified version of the core routine in the *scrypt* password hashing algorithm (“ROMix”) is memory-hard in the random-oracle model [68]. Although scrypt uses a password-dependent access pattern, we show that *even if* scrypt used a password-independent access pattern, it would be still be a memory-hard function in the sense of Section 2.2.

Alwen et al. [6] give a proof memory-hardness of scrypt in the stronger parallel random-oracle model (Section 5.1) under combinatorial conjectures (see Footnote 11). Here we prove memory-hardness in the traditional sequential random-oracle model. Our goal is not to prove a more powerful statement about scrypt—just to show that our techniques are easy to apply and are broadly relevant to the analysis of memory-hard functions.

Simplified scrypt. The simplified variant of scrypt we consider here—which operates on a buffer $(x_1, \dots, x_n)$ of n memory blocks using random oracles H_1, H_2, and H_3—operates as follows:

```
1. Set $x_1 \leftarrow H_1(1, \text{passwd}, \text{salt})$.
2. For $i = 2, \dots, n$: Set $x_i \leftarrow H_1(i, x_{i-1})$.
3. Set $y \leftarrow x_n$.
4. For $i = 2, \dots, n$:
     – Set $r \leftarrow H_2(i, \text{salt}) \in \{1, \dots, n\}$.
     – Set $y \leftarrow H_3(i, y, x_r)$.
```

5. Output y.

This variant of scrypt uses a password-independent access pattern, so our time-space lower bound applies not only to conventional scrypt but also to this cache-attack-safe scrypt variant. In Step 4 above, if we selected r by also hashing in the value of y, then we would have a hash function with a password-dependent scheme that behaves more like traditional scrypt.

Theorem 36 *Fix a positive integer n_0. Let G_n denote the data-dependency graph for the core scrypt function (ROMix) on n memory blocks. Then any strategy for pebbling G_n using at most S pebbles requires at least T pebbling moves, such that:*

$$S \cdot T \geq \frac{n^2}{24},$$

for $n_0 \leq S < n/4$, with probability $n^2 2^{-n_0}$.

Proof. By inspection of the ROMix algorithm, one sees that G_n is a 1-random sandwich graph with n top and n bottom vertices.

We now use an argument similar to that of Lemma 28 to show that G_n is hard to pebble with few pebbles. Fix an integer m such that $128 \leq m \leq n/4$. Then any set of $3m$ vertices on the bottom half of G_n have $3m$ predecessors on the top half of G_n, chosen independently and uniformly at random (using the random oracle). By Lemma 7 (applied with $\delta = 3$ and $\omega = 2$), these predecessors are $(m, n/4)$-well spread over the top vertices, except with probability 2^{-m}.

Fix an integer n_0. Using a Union Bound, the probability that any set of $3m$ consecutive vertices for $m \geq n_0$ induces a poorly spread set of predecessors is at most: $\sum_{i=1}^{n} \sum_{m=n_0}^{n/4} 2^{-m} \leq n^2 2^{-n_0}$. So, except with probability $n^2 2^{-n_0}$, the graph G_n is $(3m, m, n/4)$-consecutively avoiding for $m \geq n_0$. Now we can apply Corollary 24 (with $\kappa = 3S$, $\sigma = S$, and $\rho = n/4$) to conclude that pebbling G_n with at most S pebbles, for $n_0 \leq S \leq n/4$, requires at least $T \geq \frac{n^2}{24S}$ pebbling moves with high probability. Thus, we have $S \cdot T \geq n^2/24$ when $n_0 \leq S \leq n/4$. □

As in Appendix E, we can now turn the pebbling lower-bound into a time-space lower bound in the random-oracle model. We omit the details, since the conversion is mechanical.

G.2 Attacking Scrypt Under Access Pattern Leakage

We show that if an attacker can obtain just a few bits of information about the data-access pattern that scrypt makes when hashing a target user's password, the attacker can mount a very efficient dictionary attack against that user's scrypt-hashed password in *constant space*. To demonstrate this, we recall a folklore attack on scrypt (described by Forler et al. [42]).[19]

[19] See Appendix G for a sketch of the core scrypt algorithm.

Say that the attacker observes the memory access pattern of scrypt as it operates on a target user's password and that the attacker gets ahold of the target user's password hash and salt (e.g., by stealing the `/etc/shadow` file on a Linux system). Say that scrypt, when hashing on the target's password, accesses the memory blocks at addresses $(A_1, A_2, A_3, \dots)$. The attacker can now mount a dictionary attack against a stolen password file as follows:

- Guess a candidate password p to try.
- Run scrypt on the password p and the salt (which is in the password file), but discard all blocks of data generated except the most recent one. Recompute the values of any blocks needed later on in the computation.
- Say that scrypt on the candidate password accesses block A'_1 first. If $A_1 \neq A'_1$, terminate the execution and try another candidate password.
- Say that scrypt on the candidate password accesses block A'_2 next. If $A_2 \neq A'_2$, terminate the execution and try another candidate password.
- *Continue in this way until recovering the target's password...*

The probability that a candidate password and the real password agree on the first k memory addresses is n^{-k}, and the expected time required to run each step of the algorithm is $n/2$ for n-block scrypt, since recomputing each discarded memory block takes this much time on average. The expected time to compute one candidate password guess is then:

$$\mathrm{E}[T] = n + \sum_{k=1}^{n} \frac{k(n/2)}{n^k} \leq n \left(1 + \frac{1}{2} \sum_{k=1}^{n} \frac{k}{n^k}\right) \leq n \left(1 + \frac{1}{2} n(1/n)\right) \in O(n).$$

Since the space usage of this attack algorithm is constant, the time-space product is only $O(n)$ instead of $\Omega(n^2)$, so scrypt essentially loses its memory-hardness completely in the face of an attacker who learns the first few bits of the memory access pattern.[20]

Of course, a valid question is whether a real-world attacker could ever learn any bits of the access pattern of scrypt on a user's password. It is certainly plausible that an attacker could use cache-timing channels—such as exist for certain block ciphers [25]—to extract memory-access pattern information, especially if a malicious user and an honest user are colocated on the same physical machine (e.g., as in a shared compute cluster). Whether or not these attacks are practical today, it seems prudent to design our password-hashing algorithms to withstand attacks by the strongest possible adversaries.

[20] Here we have assumed that the attacker knows the entire access pattern, but a similarly devastating attack applies even if the attacker only knows the first few memory address locations.