Balloon Hashing: Provably Space-Hard Hash Functions with Data-Independent Access Patterns

Henry Corrigan-Gibbs
Stanford University

Dan Boneh
Stanford University

Stuart Schechter
Microsoft Research

January 14, 2016

Abstract. We present the Balloon family of password hashing functions. These are the first cryptographic hash functions with proven space-hardness properties that: (i) use a password-independent access pattern, (ii) build exclusively upon standard cryptographic primitives, and (iii) are fast enough for real-world use. Space-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. The central technical challenge of this work was to devise the graph-theoretic and linear-algebraic techniques necessary to prove the space-hardness properties of the Balloon functions (in the random-oracle model). To motivate our interest in security proofs, we demonstrate that it is possible to compute Argon2i, a recently proposed space-hard function that lacks a formal analysis, in less than the claimed required space with no increase in the computation time.

1 Introduction

The theft of personal data from websites and online services has become routine. In 2015 alone, attackers stole files containing users' login names, password hashes, and contact information from LastPass [97], Harvard [54], E*Trade [75], ICANN [51], Costco [47], T-Mobile [92], the University of Virginia [90], and a staggering number of other organizations [80]. 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 of this sort, the security of a user's password rests on the cost to an attacker of "cracking" the password's encoding. Well-designed systems store the cryptographic hash of each password with a user-specific salt. (Salting prevents attackers from using pre-computed *rainbow tables* to invert the password hashes with little computational effort [46, 67].) To crack a salted password hash, the attacker repeatedly guesses candidate passwords and calculates the hash of each guessed password with the user's salt. The attacker then checks if the resulting hash matches the hash obtained from the user's compromised account record. If the two values do not match, the attacker repeats the process with another candidate password.

The cost to crack each user's password is thus the number of guesses required times the cost of calculating the cryptographic hash function. This formulation suggests two ways to increase the attacker's cost: either increase the average number of guesses required, by strengthening the user-chosen passwords [18, 20, 39, 52, 53, 85], or increase the cost of each guess, by strengthening the cryptographic hash function. We focus on the latter goal.

Ideally, the attackers' cost per guess would be equal to the cost that the authentication system pays to validate a user's password. However, authentication systems typically run on general-purpose computing hardware, whereas attackers can use special-purpose machines designed for password-cracking. Attackers computing traditional hash functions using specialized circuits need not pay for components unrelated to hashing, such as DRAM, disks, and high-speed I/O systems. By stripping off this superfluous hardware, attackers can obtain a massive cost savings over using general-purpose hardware. Dedicated hardware for SHA-256 hashing, for example, yields roughly a million-fold cost savings over commodity servers [32]. Even off-the-self GPUs yield formidable speed-ups over general-purpose CPUs in parallelizable hash computations [89].

To minimize attackers' potential economic advantage, hash-function designers now augment the resources required for computation to include random-access memory [1, 34]. Like logic circuits, high-speed random-access memory circuits are expensive to build and power. *Space-hard* hash functions, functions that require access to large memory buffers during their execution, thus steeply increase the cost of password-cracking attacks.

A popular space-hard password-hashing function is *scrypt*, designed by Percival in 2009 [73]. We seek to improve on three shortcomings of scrypt. First, scrypt's memory-access pattern depends on the value being hashed. If an attacker can use a cache-timing side channel attack to learn the first few cache lines accessed when scrypt hashes a user's password, the attacker could cheaply rule out candidate passwords by inspecting the memory behavior of the first few steps of the hash computation on those candidates. Second, there are time-space trade-offs against scrypt that may give a cost advantage to attackers using GPUs [30]. Third, scrypt's security analysis relies on properties of BlockMix, a non-standard and otherwise unstudied hash function. A password-hashing design competition [70] has yielded interesting alternatives to scrypt [3, 15, 42], though these new space-hard designs do not offer sort of rigorous security guarantees we provide (and prove).

In this paper, we introduce the Balloon family of space-hard cryptographic hash functions. These are the first password hashing functions to simultaneously satisfy four important design goals [70]:

- **[Proven] Space-Hardness Properties.** The Balloon hash functions are moderately hard to compute with N bits of space but are prohibitively expensive to compute with much less space than that (e.g., $N/8$ bits). We *prove*, in the random-oracle model [10], the precise space-hardness properties of the Balloon functions. Most prior constructions have no formal security analysis, in the random oracle model or otherwise.
- **Built from Standard Primitives.** The Balloon hashes require only a standard cryptographic hash function (e.g., SHA-3 or SHA-512) as a subroutine so they need not rely on new and unstudied cryptographic primitives.
- **Resistant to Cache Attacks.** The memory access pattern of each Balloon hash function is *independent* of the password being hashed. Thus, an adversary who can observe the memory access patterns of a Balloon computation, e.g. via cache side-channels [22, 68, 93] on a multi-user system, learns no information about the password.
- **Practical.** The Balloon hash functions are easy to implement and are fast enough to support hundreds of authentications per second while using using all of the high-speed (L2) memory on a modern CPU core.

Achieving all of these properties at once is surprisingly difficult. Argon2 [15], the winner of the recent Password Hashing Competition [70], achieves the latter three design goals but lacks a formal analysis of its claimed space-hardness properties. To motivate our interest in space-hardness proofs, we show in Appendix A that it is possible for an adversary to reduce the space usage of single-pass Argon2i by a factor of five without paying any penalty in computation time. A similar analysis of multiple-pass Argon2i (the recommended mode) yields a factor of $e \approx 2.72$ space savings. By providing a formal security analysis of the Balloon functions, we preclude similar small-space attacks against them.

We introduce three distinct Balloon functions, each of which sits at a different point on a security-speed curve (see Table 1). Our fastest function consumes the most working space per unit time but offers relatively weak formal space-hardness guarantees. Our strongest function (in terms of its space-hardness guarantees) fills memory at a slightly slower rate.

The three Balloon functions all use a similar design principle: they first fill up a large buffer with pseudo-random bytes derived from the password and salt. Next, they "mix" the buffer contents by reading pseudo-randomly selected bytes out of the buffer, hashing them together, and writing them back in. Finally, they extract a few bytes from the buffer as output. As this description indicates, the Balloon functions are essentially "modes of operation" for an underlying non-space-hard cryptographic hash function.

We prove the security of the first two Balloon variants using pebble games, which are arguments about the structure of the data-dependency graph of the underlying computation [56, 72, 86]. To prove the security of the third construction, we use a novel analysis that relates the structure of certain matrices to the space-hardness of computing the linear transformations they represent (reminiscent of Valiant's classic depth lower bounds [96]). Both analyses are in the random-oracle model [10].

To demonstrate that the Balloon functions are fast enough for real-world use, we implement each of the three variants and compare their performance to that of existing schemes. A four-core server using our fastest construction with a 1 MiB buffer can compute 115 hashes per second and can compute our strongest construction (in terms of the security bounds) with a 1 MiB buffer at the rate of 19.5 hashes per second. For comparison, our server computes PBKDF2-HMAC-SHA512 hashes (using 10^5 iterations) at the rate of 17.3 hashes per second.

Contributions. Our primary contribution in this work is the design and analysis of the Balloon functions, the first space-hard hash functions that use a password-independent access pattern, build upon standard cryptographic primitives, and run at speeds competitive with existing password-hashing functions. We also advance the state of the art in space-hard hash function design by: (1) introducing and studying a new class of directed acyclic graphs, which may be useful in future time-space analyses, (2) applying novel linear-algebraic techniques to analyze time-space trade-offs in the random-oracle model, and (3) investigating the space-hardness of Argon2i (see Appendix A). With this work, we aim to show that space-hard functions can be at once practically efficient and provably secure.

2 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 [64]. Their DES-based *crypt* design became the standard for password storage for over a decade [57] and even has a formal analysis by Wagner and Goldberg [98].

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 [28].

Provos and Mazières realized 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 [81]. 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 [61]. To increase the cost of custom password-cracking hardware, Reinhold's *HEKS* hash [82] and Percival's popular *scrypt* routine consume an adjustable amount of storage *space* [73], in addition to time, as they compute a hash.

The Balloon functions, like scrypt, aim 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 [22, 68, 93]. Our functions also prevent certain time-space trade-offs against scrypt [30] and use only standard cryptographic primitives, whereas scrypt's analysis relies on a new primitive built from the Salsa20 core [13].

The recent Password Hashing Competition motivated the search for space-hard password-hashing functions that use data-independent memory access patterns [70]. The Argon2 family of functions, which have excellent performance and an appealingly simple design, won the competition [15]. The Argon2 functions lack a theoretical analysis of the feasible time-space trade-offs against them. As we discuss in Appendix A, without such an analysis it is difficult to know what the exact space-hardness properties of the Argon functions are.

The Catena family of hash functions [42], which became finalists in the Password Hashing Competition, are space-hard functions whose analysis applies pebbling arguments to classic graph-theoretic results of Lengauer and Tarjan [56]. The faster of the two Catena constructions had a flawed security analysis and corresponding attack [17] and the other has some practical limitations: when using N blocks of working space, the Catena hash requires $\Theta(N \log N)$ invocations of a cryptographic compression function per round, compared with $\Theta(N)$ for our schemes. In addition, the Catena security theorems do not rule out many interesting small-space attacks (e.g., an attacker using $N/16$ space).

The other competition finalists included a number of interesting designs that differ from ours in important ways. Makwa [79] supports offloading the work of password hashing to an untrusted server but is not space-hard. Lyra [3] is a space-hard function but lacks proven space-time lower bounds. Yescrypt [74] is an extension of scrypt and uses a password-dependent data access pattern.

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 [9] 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 [31]. 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 [23], a technique of Canetti et al. for using CAPTCHAs in concert with hashes [25], and a proposal by Dürmuth to use password hashing to do meaningful computation [33].

In recent theoretical work, Alwen and Serbinenko introduce the notion of parallel-space hardness [4]: functions that use a lot of space not only at *some* point during a computation but during *most* points in a computation. Applying their analysis techniques to the Balloon functions presents an interesting and important challenge for future work.

Proofs of Space. Abadi et al. [1] introduced "proofs of space" as more effective alternatives to traditional proofs-of-work in heterogeneous computing environments [7, 35]. There exist theoretically analyzed proof-of-space schemes [5, 34, 37] (many relying also on pebbling arguments) and others with heuristic hardness [58, 94]. Applications of proofs-of-space to crypto-currencies, like Dogecoin, Litecoin [21], and the newly proposed Spacecoin [69] have renewed interest in the area. By making the currency mining process space-hard, these currencies aim to decrease the economic advantage of industrial-scale miners using custom mining hardware.

Two features distinguish a proof-of-space from a space-hard password hashing function. First, a proof-of-space may access memory in a pattern that depends on the input to the proof. In the context of password hashing, we prefer to avoid data-dependent addressing to prevent cache attacks. Second, proofs-of-space must be efficiently checkable—the verifier of the proof should be quite efficient while the prover need not be. We need no such asymmetry in our setting: computing a hash should be moderately hard for all parties.

Time-Space Trade-Offs. The techniques we use to analyze the Balloon functions draw on extensive prior work on computational time-space trade-offs. We use pebbling arguments, which have seen application to register allocation problems [86], to the analysis of the relationships between complexity classes [12, 26, 48, 91], and to prior cryptographic constructions [36, 37, 38, 42]. Pebbling has also been a topic of study in its own right [56, 72]. Savage's text gives a clear introduction to graph pebbling [84] and Nordström surveys the vast body of pebbling results in depth [66]. In the latter part of the paper, we use properties of linear transformations to prove time-space lower bounds. This connection is reminiscent of the techniques of Leslie Valiant [96] and Yaacov Yesha [99].

3 Goals

This section summarizes the high-level security and functionality goals of a password hashing function in general and the Balloon functions in particular. We draw these aims from prior work on password hashing [73, 81] and also from the requirements of the recent Password Hashing Competition [70].

3.1 Syntax

Each of our password hash functions takes four inputs: a password, salt, time parameter, and space parameter. The output of each function is a bitstring of fixed length (e.g., 256 or 512 bits). The password and salt are standard [64], but we elaborate on the role of the latter parameters below.

Time Parameter (*Number of Rounds*). Our functions take as input a parameter r that determines the number of "rounds" of computation they perform. As in bcrypt [81], the larger the time parameter, the longer the hash computation will take. As computational power increases, users can increase this time parameter to keep the number of wall-clock seconds required to compute each hash near-constant.

Space Parameter (*Buffer Size*). The space parameter indicates how many bytes of working space the hash function will require during the course of its computation, as in scrypt [73]. Computing the

Balloon functions with much less than than the specified amount of buffer space incurs a huge slow-down in the computation time. We make the exact time-space trade-offs precise later on.

For execution on multi-core processors, two of the three Balloon functions allow a limited and configurable amount of parallelism. By using all available compute cores, the defender (i.e., legitimate authentication server) can increase the amount of space the Balloon functions consume per unit time. Allowing for limited parallelism is a common feature of space-hard password hashing functions [15, 42, 73, 74].

3.2 Security Properties

The high-level security goals of the Balloon functions are as follows.

Space-Hardness. The Balloon functions are *space hard*. Informally, that means that they are "easy" to compute with $N + O(1)$ blocks of storage space but are "hard" to compute with "many fewer than" N blocks of space. To make this definition precise, we use the number of random oracle queries needed to compute a function as a proxy for the function's time complexity [10]. A computer with $N + O(1)$ blocks of working space available can compute the Balloon functions with roughly $c \cdot r \cdot N$ random-oracle queries, where r is the number of rounds and c is a small constant—less than 20, say. For algorithms using a constant fraction less space (e.g., $N/8$ block of working space), we prove lower bounds on the running time of any algorithm computing these functions with high probability (Table 1). Thus, we can make statements of the form: "Computing the r-round single-buffer Balloon function with $N/8$ k-bit blocks of space requires time at least $2^r N$."

In many cases, small-space strategies for computing our functions result in a computational slowdown that is *exponential* in the number of rounds r. From a theoretical perspective, if one chooses the number of rounds r to be a function of N, then the time required to compute the Balloon functions in small space can be super-polynomial (e.g., if $r = \log^2 N$) or even sub-exponential in N (e.g., if $r = \sqrt{N}$) [71].

Password-Independent Access Pattern. A first-class design goal of the Balloon functions is that their memory access patterns be *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 [22, 68, 93]. This may be especially important in cloud-computing environments, in which many mutually distrustful users share a single physical host [83].

Creating a space-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 space-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.

Collision Resistance, etc. If necessary, we can modify the Balloon hash functions so that they provide the standard properties of second-preimage resistance and collision resistance [62]. It is possible to achieve these properties in a straightforward way by composing a Balloon function B with a standard cryptographic hash function H as $H(\text{password}, \text{salt}, B(\text{password}, \text{salt}))$, so we focus our attention on the space-hardness properties of the Balloon functions.

4 Constructions

In this section, we present the three Balloon functions and we defer the security analysis of the constructions to latter sections. Each of the Balloon functions uses a standard (non-space-hard) cryptographic hash function $H : \{0,1\}^{2k} \to \{0,1\}^k$ as a subroutine. All constructions use 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 issues described later on (in Section 5.3), 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 functions operate in three steps:

1. **Expand.** In the first step, the Balloon functions fill up a large buffer with pseudo-random bytes derived from the password and salt. The current implementation uses a stream cipher (e.g., AES-CTR on machines with hardware AES support) to fill this buffer initially. Repeatedly invoking the compression function H with the password, salt, and a counter as input would also work.

2. **Mix.** In the second step, the Balloon functions perform a "mixing" operation on the pseudo-random bytes in the memory buffer. The user-specified round parameter r determines how many rounds of mixing take place.

3. **Extract.** In the last step, the Balloon functions extract the final output value from the memory buffer, either by outputting the last block of the buffer or by outputting some function of the entire buffer (this steps differs with each construction).

Notation. Throughout this paper, Greek symbols (α, β, γ, λ, etc.) typically denote constants greater than one. For convenience, we often implicitly assume that a certain number (e.g., n/α) is integral. We use $\log_2(\cdot)$ to denote a base-two logarithm and $\log(\cdot)$ to denote a logarithm when the base is irrelevant. For strings x and y, we write their concatenation as $x \| y$. We use $\mathbb{F}$ to denote an arbitrary finite field with $|\mathbb{F}|$ elements, $\mathbb{F}_k$ to denote the unique field of k elements, and $\mathbb{F}^{m \times n}$ to denote an m-by-n matrix whose elements are in $\mathbb{F}$. The notation $\operatorname{polylog} x$ indicates a fixed polynomial in $\log x$. For a finite set S, the notation $x \xleftarrow{\text{R}} S$ indicates sampling an element of S uniformly at random and assigning it to the variable x.

Construction I: Single-Buffer

The first member of the Balloon function family operates on a single buffer of N memory blocks (Figure 1). The function first fills up the buffer with pseudo-random bytes derived from the inputs. Next, it runs the mixing operation r times. 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 $\delta = 20$ other blocks chosen "at random" from the buffer. (The choice of the constant δ is important in the security analysis, as we explain in Section 7.) 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. After mixing the buffer for r rounds, the routine returns the last block of the buffer as its output.

```
func balloon_single(char passwd[], char salt[],
    int s_cost,   // Space cost (main buffer size)
    int t_cost):  // Time cost (num. of rounds)
  int delta = 20; // Num. of dependencies
  int cnt = 0;    // A counter (used in sec. proof)
  block_t buf[s_cost];  // The main buffer

  // Step 1. Expand input into buffer.
  for m in range(s_cost):
    buf[m] = hash(cnt++, passwd, salt);

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

      // Step 2b. Hash in pseudorandomly
      // chosen blocks.
      for i in range(delta):
        int other = random_index(t, m, i, salt);
        buf[m] = hash(cnt++, buf[m], buf[other]);

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

Figure 1: Pseudo-code of the single-buffer Balloon hash function.

This construction provides very strong protection against time-space trade-offs: computing the function's output with high probability in N blocks of space requires a number of calls to the underlying hash function that is linear in r and N. Computing the function with much less than N blocks of space, e.g., $N/8$ blocks of space, causes the time required to compute the function to increase by a factor of 2^r.

A limitation of the single-buffer construction 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. The next member of the Balloon function family weakens this restriction to increase the rate at which the Balloon functions can fill memory on multi-core machines.

	Parallel	Hash Calls	**Small-Space Computation Time (r rounds)** $S < N/4$	$S < N/8$	$S < N/16$	$S < N/32$	$S < N/128$
Argon2i (one pass)	✓	N	N	?	?	?	?
Single-Buffer ($\delta = 20$)		$20N$	$r^2 N/4$	$2^r N$	$3^{r+1} N/4$	$4^{r+1} N/6$	$4^{r+1} N/6$
Double-Buffer ($\delta = 20$)	✓	$10N$	$r^2 N/4$	$r^2 N/4$	$2^r N$	$3^{r+1} N/4$	$4^{r+1} N/6$
Linear (Thm. 27, $w = 10$)	✓	$N/2$					$2^r N$

Table 1: Comparison of the three hash constructions when used with an N-block buffer. The values in the right half of the table indicate the minimum computation time for each function, measured in terms of number of random oracle calls. (Per Remark 14, it is possible to strengthen the single- and double-buffer bounds with extra analysis.)

Construction II: Double-Buffer

The double-buffer construction modifies the construction above to allow all three steps of the hash computation to be parallelized, to more fully take advantage of the processing power on multi-core machines. Although it would be possible to achieve p-fold parallelism by just running p parallel instances of Construction I, allowing different threads of execution to read from the same buffer yields stronger space-hardness properties with the same speed-up.

The double-buffer variant modifies the single-buffer construction by splitting the main N-block memory buffer into two pieces of $N/2$ blocks each. We call one of these buffers the "source" buffer and one the "destination" buffer.

In the expansion step of the computation, which is already parallelizable, the routine fills up the source buffer with pseudorandom bytes as before (see Figure 1, Step 1). The mixing step of the computation proceeds differently. Rather than update the buffer in place, each thread now treats the "source" buffer as read-only and writes into a thread-private segment of the "destination" buffer. For each block in the destination buffer, the routine picks $\delta = 20$ neighboring blocks in the source buffer, hashes them together, and writes the result into the destination buffer. Many threads can perform this mixing operation concurrently since they read from a read-only buffer and their writes never conflict. Once the destination buffer is full, the routine swaps the source and destination buffers. Finally, in the extraction step, the routine outputs the hash of all of the blocks in the last-written buffer as the output of the function overall.

The construction we implement is a bit more complicated, in that we force the value of ith block in each thread-private segment to depend on the value of the $(i-1)$th block. We also force the first block in each segment in the "destination" buffer to depend on the last block in each segment of the "source" buffer. These tweaks *limit* the amount of parallelism (so that the attacker's parallelism does not exceed the defender's).

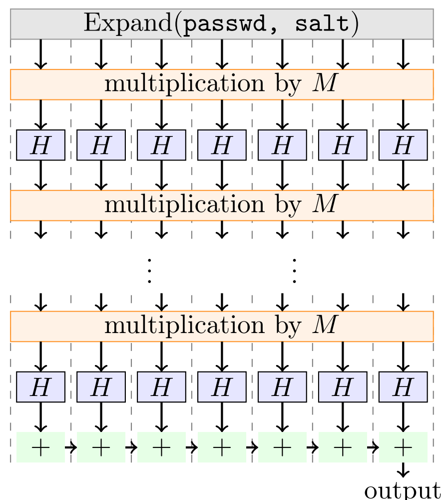


Figure 2: The linear construction alternates a linear transformation on the blocks of the buffer with a hashing step.

The primary limitation of the double-buffer construction is still performance: each mixing iteration requires making roughly $20(N/2) = 10N$ calls to the underlying hash function.

Construction III: Linear

The main performance bottleneck of the double-buffer construction is that for each block of memory written, the function must invoke the underlying cryptographic hash function roughly 20 times. That

is, we hash together the 20 input blocks from the source buffer to produce the single output block that we will write into the destination buffer.

A simpleminded way to speed this step up would be to read the 20 blocks of memory from the source buffer, XOR them together, and run the resulting value through the cryptographic compression function to obtain the output block that we write to the destination buffer. This optimization would reduce the number of cryptographic compression function calls down to one—instead of 20—per block written, but would the resulting scheme still be secure?

With some work, we show in Section 9 that it is. Indeed, we show that the resulting scheme is secure for a large class of linear transformations, including the transformation represented by XORing 20 blocks together per block written. As in the double-buffer design, this construction uses two parallel buffers. The expansion step (Step 1) proceeds exactly as in the double-buffer construction. The mixing and extraction steps (Steps 2 and 3) differ, as we describe below.

Let us first introduce a bit of notation. If each block of memory in the buffer is k bits long, we can view each block as an element of the finite field $\mathbb{F}_{2^k}$. The source and destination buffers contain N blocks total, or $n = N/2$ blocks each. Thus we can treat the state of each buffer as a vector in the space $\mathbb{F}_{2^k}^n$.

The mixing step requires a public matrix $M \in \mathbb{F}_{2^k}^{n\times n}$, the properties of which we describe later on. In each step of mixing, the routine computes the product of the matrix M with the vector consisting of the state of the source buffer, and stores the result in the destination buffer. The routine hashes each value in the destination buffer in place and swaps the two buffers (Figure 2). To extract the output, the algorithm computes the block-wise XOR of all blocks in the final buffer (Figure 3).

The finesse comes in choosing the matrix M. For now, let us say that a random $n \times n$ matrix over $\mathbb{F}_{2^k}$ in which 10 independently chosen entries per row are set to 1, and all other entries are 0, is a suitable choice. We will return to this topic in Section 9.

The design of our linear construction bears resemblance to the design of certain block ciphers that alternate a linear transformation with a non-linear step [2]. Exploring whether there are deeper connections between these two areas of cryptography may be worthy of further study.

```
matrix_t M;   // A public matrix
int cnt = 0;  // A counter (used in sec. proof)
block_t src[s_cost/2], dst[s_cost/2];
...
// Step 2. Mix from src to dst buffer.
for t in range(t_cost):
  // Perform a matrix-vector mulitply.
  dst = matrix_multiply(M, src)
  // Write result back into src buffer.
  for m in range(s_cost/2):
    src[m] = hash(cnt++, dst[m]);
...
```

Figure 3: Pseudo-code for the mixing step of the linear construction.

5 Pebble Games

In this section, we introduce pebble games and explain how to use them to analyze the first two Balloon functions. Pebble games are a technique from the theoretical computer science literature for analyzing computational time-space trade-offs [48, 56, 72, 77, 86, 96] and space-hard functions [4, 36, 37, 42].

5.1 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. 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. At each time step in the pebble game, the player may:

- place a pebble on a sink vertex,
- remove a pebble from any pebbled vertex, or
- place a pebble on a non-sink vertex *if and only if* all of its successor vertices are pebbled.[1]

The goal of the game is to place a pebble on a particular source node of the graph in as few moves as possible.

Definition 1 (Legal Pebbling)**.** A sequence of pebbling moves is *legal* if each move in the sequence obeys the rules of the game.

[1] The pebble game is often defined in the opposite direction, in which pebbles move from sources to sinks, but this modified version simplifies our discussion later on.

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 [78]. Edges in the graph correspond to data dependencies, while vertices correspond to intermediate values needed in the computation. Sink nodes represent input values (which have no dependencies) and source 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.

5.2 Pebbling in the Random-Oracle Model

Dwork, Naor, and Wee [36] 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 [10]. This observation became the basis for the design of the Catena space-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. [36], as modified by Alwen and Serbinenko [4]. 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 sink and $v_{|V|}$ is a source and, to simplify the discussion, we assume that G has a unique source node.

Definition 2 (Labeling)**.** Let $G = (V, E)$ be a directed graph with maximum out-degree δ and a unique source vertex, let $x \in \{0,1\}^k$ be a string, and let $H : \mathbb{Z}_{|V|} \times \{0,1\}^{k\delta} \to \{0,1\}^k$ be a function, modeled as a random oracle. We define the labeling of G relative to H and x as:

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

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

The labeling of the graph G proceeds from the sinks to the unique source node source: first, the sinks of G receive labels, then their predecessors receive labels, and so on until finally the unique source 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 source 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 [36, Lemma 1].[2] 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.

We use a version of their result due to Dziembowski et al. [38]. The probabilities in the following theorems are over the choice of the random oracle and the randomness of the adversary.

Theorem 3 (Adapted from Theorem 4.2 of Dziembowski et al. [38])**.** *Let G, H, and k be as in Definition 2. 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.*

[2] 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.

Proof. The first two parts are a special case of Theorem 4.2 of Dziembowski et al. [38]. 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 4. *Let G and k be as in Definition 2 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 3, 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 $\frac{\sigma+k}{k-\log_2 T^*}$ 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 3, 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.

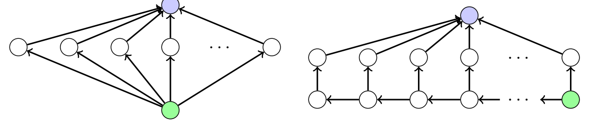


Figure 4: A graph requiring $n + 1$ pebbles to pebble in the random-oracle model (left) requires $O(1)$ storage to compute when using a Merkle-Damgård hash function (right).

By the Union Bound, we find that:

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

Substituting in the probabilities of each of these events derived from Theorem 3, we find

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

□

5.3 Dangers of the Pebbling Paradigm

The beauty of the pebbling paradigm is that it allows us to reason about the space-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.

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 3 relies on the fact that when H is a random oracle, computing the value $H(x_1, \dots, x_n)$ requires that the entire string $(x_1, \dots, x_n)$ 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 [29, 63] built from a two-to-one compression function $C : \{0,1\}^{2k} \to \{0,1\}^k$ as

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

If H is one such hash function, then the computation of $H(x_1, \dots, x_n)$ 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 4, 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 space-hard functions in the past [16, 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 [8, 24, 45, 65].) While prior works study variants of the Merkle-Damgård construction that are indifferentiable from a random oracle [27], 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.

6 Avoider Graphs

In this section, we introduce *avoider graphs*, a type of graph that has a very rich connectivity structure and yet can be quite sparse. We show that "stacks" of avoider graphs are difficult to pebble with few pebbles and few moves. As a result, if the data dependency graph for a particular computation is a stack of avoiders, then executing that computation with a small amount of storage space will inevitably require a lot of time. We use arguments along these lines to prove the space-hardness of Constructions 1 and 2. Throughout this section, we use n to indicate the number of distinguished source and sink nodes in the graph—n does *not* refer to the total number of vertices in the graph.

6.1 Expander Graphs

We will use expanders in the rest of this section, so we recall their definition.

Definition 5. A directed bipartite graph $G = (A \cup B, E)$ with $n = |A| = |B|$ is an (α, β)*-bipartite expander* with source nodes A, sink nodes B, and left-degree δ if:

1. every vertex in A has out-degree at most δ, every vertex in B has out-degree zero, and
2. for every subset $A' \subseteq A$ such that $|A'| = n/\alpha$, the set A' has more than n/β successors in B.

Consider a bipartite graph generated by choosing, for every vertex $v \in A$, δ successors independently and uniformly at random from B, collapsing parallel edges. Pinsker [76] demonstrated that, as long as the degree δ is a large enough constant, a graph generated according to this process will be an expander with good probability.

Theorem 6 (Pinsker [76])**.** *For all $\alpha, \beta, \lambda, n > 1$, a random bipartite graph with left-degree δ, n sources, and n sinks is an (α, β)-bipartite expander with probability greater than $1 - 2^{-\lambda}$ if:*

$$\delta \geq \frac{\alpha}{\log_2 \beta}\left[H_b(1/\alpha) + H_b(1/\beta) + \frac{\lambda}{n}\right].$$

Here, $H_b(\cdot)$ denotes the binary entropy function.

Theorem 6 implies a randomized construction of expanders that is extremely simple to implement, provides good expansion, and fails with exponentially small probability. For these reasons, the randomized construction is likely the best choice for a real-world implementation. That said, we use bipartite expanders only in a "black box" way, so it would be possible to use a deterministically constructed expander instead in all of our constructions (see Goldreich [44]).

6.2 Defining Avoiders

An avoider graph is a directed graph with n sources, n sinks, and potentially many non-source non-sink ("middle") vertices. The key property of an avoider graph is that every small number of sources has paths to a large number of sinks, *even if* some of these paths are blocked. The name "avoider" comes from the notion that there is always a set of source-sink paths that avoids the blocking vertices. Our definition of avoiders is related to, but weaker than, the definition of Valiant's $f(r)$-grates [96].

The avoider property is useful to us because it is invariant under certain transformations of the graph. For example, we show that an avoider graph with out-degree at most δ can be transformed into an equally good avoider graph with out-degree at most 2. To our knowledge, no existing class of switching graphs (expanders, concentrators, super-concentrators, etc. [76, 95, 96]) maintains the properties we need under the transformations we use.

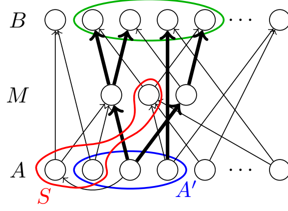


Figure 5: A graph with sources A, sinks B, and middle nodes M. There are four S-avoiding sink-distinct source-sink paths beginning in A' (illustrated with thickened edges).

We say that a path $p = (v_1, \ldots, v_k)$ in a graph *avoids* a set of vertices S if no vertex in p is in the set S. For a graph with distinguished source and sink vertices, we say that a *source-sink path* is a path beginning at a source node and terminating at a sink node. Since the sink nodes in our graphs have out-degree zero, there are never two sink nodes along a path in these graphs. We say that a set of source-sink paths is *sink-distinct* if no two paths in the set terminate at the same sink (see Figure 5 for an example).

Definition 7. An (α, β, γ)-*avoider* is a directed acyclic graph $G = (A \cup B \cup M, E)$, with source nodes A, sink nodes B, and middle nodes M such that:

1. G has has n sources and n sinks, with all sinks having out-degree zero,

2. for every set $A' \subseteq A$ of size n/α and for every set $S \subseteq (A \cup M)$ of size at most n/γ, there exists a set of at least n/β sink-distinct source-sink paths from A' to B, avoiding the set of vertices S.

For an avoider graph to exist, we must have $\alpha < \gamma$. If not, then the set of vertices S "to avoid" is larger than the set of sources nodes A'. In this case, for every set $A' \subseteq A$ of size n/α there exists a set $S \subseteq A$ such that $A' \subseteq S$. In this case, there can never be a set of S-avoiding paths from A' to B.

A first point to see is that a bipartite expander is also an avoider, for an appropriate setting of the parameters.

Claim 8. *Let $\alpha > 2$. If $G = (A \cup B, E)$ is an (α, β)-bipartite expander with sources A and sinks B, then G is an $(\alpha/2, \beta, \alpha)$-avoider with sources A, sinks B, and no middle nodes ($M = \emptyset$).*

Proof. The first property of an avoider holds by construction. To show the second property, fix a subset $A' \subseteq A$ of size $2n/\alpha$ and a subset $S \subseteq A$ of size n/α (since G has no middle nodes, the blocking set S consists only of source nodes). There are at least $2n/\alpha - n/\alpha = n/\alpha$ nodes of A' outside of S and since G is an (α, β)-expander, these nodes must have at least n/β successors. Thus, there are least n/β sink-distinct S-avoiding source-sink paths from A' to B in G. ☐

6.3 Transformations on Avoiders

One useful aspect of the definition of avoiders is that a bipartite expander graph is still an avoider under a "localizing" transformation. Consider a bipartite graph G on $2n$ vertices $a_1, \ldots, a_{2n}$, such that edges flow from the last n vertices to the first n vertices. The localized graph $\mathcal{L}(G)$ we construct has the property that (1) if G is an expander, then $\mathcal{L}(G)$ is an avoider and (2) the successors of vertex a_i are a subset of the set $\{a_{i-1}, \ldots, a_{i-n}\}$. In this sense, G's edges are "local"—no edge of G points to a vertex too far from its origin. As Figure 6 demonstrates, the localized graph $\mathcal{L}(G)$ is not necessarily bipartite, even though the original graph G was.

We use this localizing transformation to make more efficient use of buffer space in Construction 1. It is possible to pebble a localized bipartite expander in linear time with $n + O(1)$ pebbles, whereas a non-localized bipartite expander can require as many as $2n$ pebbles in the worst case. This transformation makes computing the space-hard hash 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 context of leakage-resilient cryptography [88].)

Definition 9. Let $G = (A \cup B, E)$ be a bipartite graph with $A = (a_1, \ldots, a_n)$, $B = (b_1, \ldots, b_n)$ and all edges flowing from A to B. The *localized graph* $\mathcal{L}(G) = (U \cup V, E')$ is a graph (not necessarily bipartite) with $|U| = |V| = n$ and with the edge set:

$$E' = \begin{array}{l} \{(u_i, v_i) \mid i \in \{1, \ldots, n\}\} \cup \\ \{(u_i, v_j) \mid (a_i, b_j) \in E \text{ and } i \leq j\} \cup \\ \{(u_i, u_j) \mid (a_i, b_j) \in E \text{ and } i > j\} \end{array}$$

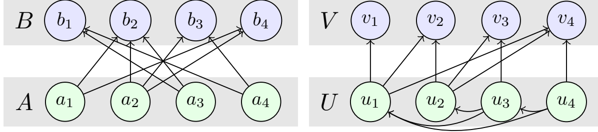


Figure 6: A bipartite graph G (left) and the corresponding localized graph $\mathcal{L}(G)$ (right).

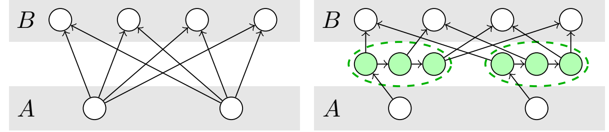


Figure 7: A bipartite graph G (left) and the corresponding degree-reduced graph $\mathcal{D}(G)$ (right). We indicate the gadgets with dashed ovals.

Claim 10. *If G is an (α, β)-bipartite expander with source nodes A and sink nodes B with $\alpha > \beta$ and $\alpha > 2$, then the localized graph $\mathcal{L}(G)$ is an $(\alpha/2, \beta', \alpha)$-avoider with source nodes U and sink nodes V, for $\beta' = \alpha\beta/(\alpha - \beta)$.*

Proof. The first property of an avoider holds by construction. To show the second property, we must show that for every subset $U' \subseteq U$ of size $2n/\alpha$ and for every subset $S \subseteq U$ of size at most n/α, there are at least n/β' S-avoiding sink-distinct paths from U' to V. Fix subsets U' and S. At most n/α of the vertices of U' are in S, which leaves n/α vertices of U' outside of S. By the expansion of G, these vertices of U' must have at least n/β distinct sinks of G as successors.

Thus, there is a set of indices $I \subseteq \{1, \ldots, n\}$ of size at least n/β such that u_i or v_i (or both) have a predecessor in the set U'. At most n/α of the vertices u_i can be in S. Thus, the set $I \setminus \{i \mid u_i \in S\}$ must have size at least $n/\beta - n/\alpha$. In other words, there are at least $n/\beta - n/\alpha$ distinct integers $i \in \{1, \ldots, n\}$ such that v_i has a predecessor in U' or $u_i \notin S$ and u_i has a predecessor in U'. Since there is a (u_i, v_i) edge in $\mathcal{L}(G)$ for all i, this implies that there are at least $(1/\beta - 1/\alpha)n = \frac{\alpha-\beta}{\alpha\beta}n = n/\beta'$ S-avoiding paths from vertices in U' to distinct sinks in V. □

The last bit of preliminary work we must do is to show that by adding auxiliary nodes to an avoider graph with out-degree δ we can produce an equally good avoider with out-degree two (similar to the technique of Paul and Tarjan [71]).

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 Section 5.3.

Definition 11. Let $G = (V, E)$ be a directed acyclic graph. We say that the *degree-reduced graph* $\mathcal{D}(G)$ is the graph in which each vertex v in G of out-degree δ is replaced with a path "gadget" that has out-degree at most 2. The original vertex v is at the head of the path, there are $\delta - 1$ internal nodes on the path, and the δ original successors of v are connected to the internal nodes of the path (see Figure 7).

By construction, nodes in $\mathcal{D}(G)$ have out-degree at most two and if G has n sources and n sinks, then $\mathcal{D}(G)$ also has n sources and n sinks. 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 $(\delta-1)$ nodes. The degree-reduced graph $\mathcal{D}(G)$ has extra "middle" nodes (non-source non-sink nodes) consisting of the internal nodes of the degree-reduction gadgets (Figure 7).

Claim 12. *If G is an (α, β, γ)-avoider with sources A and sinks B then $\mathcal{D}(G)$ is an (α, β, γ)-avoider with sources A and sinks B.*

Proof. The first property of an avoider holds by construction, so we focus on the second property. By way of contradiction, assume that there exists some set $A'_{\mathcal{D}}$ of size n/α and some set $S_{\mathcal{D}}$ of size at most n/γ in $\mathcal{D}(G)$ such that there are fewer than n/β sink-distinct paths from A' to B in $\mathcal{D}(G)$ avoiding $S_{\mathcal{D}}$. Since $|S_{\mathcal{D}}| \leq n/\gamma$, the vertices in the set $S_{\mathcal{D}}$ consist of vertices on at most n/γ distinct degree-reduction gadgets in $\mathcal{D}(G)$. Let S be the set of vertices in G corresponding to the degree-reduction gadgets in $\mathcal{D}(G)$ that $S_{\mathcal{D}}$ touches.

Now we know that for every set of vertices $A' \subseteq A$ in G of size n/α, there are at least n/β sink-distinct source-sink paths in G avoiding S (since G is an avoider). Every such set A' in G corresponds to an equivalent set A' in $\mathcal{D}(G)$ and every set of sink-distinct source-sink paths avoiding S in G corresponds to a set of sink-distinct source-sink paths in $\mathcal{D}(G)$ avoiding the gadgets in $S_{\mathcal{D}}$. Thus, for every set $A'_{\mathcal{D}}$ and $S_{\mathcal{D}}$ in $\mathcal{D}(G)$ as above, there must be at least n/β $S_{\mathcal{D}}$-avoiding sink-distinct source-sink paths in $\mathcal{D}(G)$. This is a contradiction. □

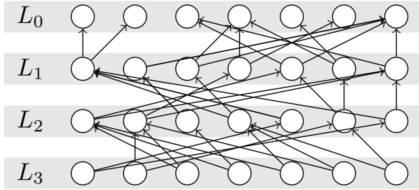


Figure 8: A stack of $d = 3$ bipartite graphs. The sinks of the stack are at level L_0 and the sources are at level L_3.

6.4 Hard-to-Pebble Graphs from Avoiders

The core of our analysis involves stacks of avoider graphs. Given an avoider graph with n sources and n sinks, we denote a *depth-d stack* of such avoiders by $G_{n,d}$. We can view the graph $G_{n,d}$ as consisting of $d+1$ layers, each of n nodes, possibly with the avoiders' middle nodes sitting between the layers. The sink nodes of the level-zero avoider sit at the very top of the stack and the source nodes of last avoider sit at the very bottom of the stack (Figure 8).

Following Paul and Tarjan [71], we show that if the graph at each level of the stack is a good avoider, then pebbling a large subset of the vertices on the last level of stack with few pebbles requires many moves.

Theorem 13. *Let $G_{n,d}$ be a graph constructed from a stack of $(\alpha, \alpha/\omega, \gamma)$-avoiders for some integer $\omega \geq 1$. Let $\mathcal{C}$ be some configuration of at most n/γ pebbles on $G_{n,d}$ and let A be a set of n/α vertices on layer i of the graph. Then every legal sequence of pebbling moves $\mathcal{M}$ that*

- *begins in configuration $\mathcal{C}$,*
- *never uses more than n/γ pebbles, and*
- *places a pebble on each vertex in the set A at some point*

must contain at least $(\frac{1}{\alpha} - \frac{1}{\gamma})(\omega^i + i)n$ pebble placements on levels $\{0, \dots, i\}$ of the graph, irrespective of the starting configuration $\mathcal{C}$.

Before proving the theorem, let us consider an instantiation of it, to make its meaning slightly more concrete.

Example. Let B be an $(8, 2)$-bipartite expander. By Claim 8, B is a $(4, 2, 8)$-avoider. Construct a graph $G_{n,d}$ as a stack of d copies of B. Now applying Theorem 13 with $\omega = 2$ asserts that every sequence of moves that pebbles some $n/4$ nodes on level d of $G_{n,d}$ using at most $n/8$ pebbles must require making at least $(2^d + d)n/8$ pebble placements on the graph.

Now we return to prove the theorem.

Proof of Theorem 13. The proof proceeds by induction on i, the level of the graph.

Base Case ($i = 0$). At the start of the sequence of moves, there are at most n/γ pebbles on the graph and thus at most n/γ pebbles on level zero. If each of n/α vertices on level zero receives a pebble during the sequence of moves, then there must be at least $n/\alpha - n/\gamma = (1/\alpha - 1/\gamma)n$ level-zero placements.

Induction Step. Let $\mathcal{M}$ be a legal sequence of pebbling moves and let A be a set of n/α vertices on level i. At the start of the sequence of moves, there are at most n/γ pebbles on the graph and thus there are at most n/γ pebbles on level i of the graph.

Since each layer of $G_{n,d}$ consists of an $(\alpha, \alpha/\omega, \gamma)$-avoider graph, every set of n/α nodes on the ith level of $G_{n,d}$ must connect to at least $\omega n/\alpha$ nodes on level $i-1$ of the graph by paths avoiding the initial positions of the n/γ pebbles on the graph. Since $\mathcal{M}$ is a sequence of legal pebbling moves, at some point during $\mathcal{M}$, all of these $\omega n/\alpha$ dependencies on level $i-1$ must receive pebbles. We can now write $\mathcal{M}$ as the concatenation of ω legal sequences of pebbling moves: $\mathcal{M} = (\mathcal{M}_1 \| \dots \| \mathcal{M}_\omega)$.

We define $\mathcal{M}_1$ to be the sequence of moves beginning in configuration $\mathcal{C}$ during which the first n/α dependencies on level $i-1$ receive pebbles. At the end of $\mathcal{M}_1$, the pebbles on the graph are in some configuration $\mathcal{C}_1$ with at most n/α pebbles on the graph. We define $\mathcal{M}_2$ to be the sequence of moves beginning in configuration $\mathcal{C}_1$ during which the next n/α pebbles on level $i-1$ receive pebbles. At the end of $\mathcal{M}_2$, the pebbles on the graph are in some configuration $\mathcal{C}_2$. Continuing in this fashion, we can divide $\mathcal{M}$ into ω valid sequences of pebbling moves $(\mathcal{M}_1, \dots, \mathcal{M}_\omega)$.

Each sub-sequence $\mathcal{M}_j$ is a legal sequence of pebbling moves beginning in a configuration of at most n/γ pebbles. Further, $\mathcal{M}_j$ never uses more than n/γ pebbles (since the entire sequence $\mathcal{M}$ never does) and during $\mathcal{M}_j$ at least n/α vertices in level $i-1$ of the graph receive pebbles. Thus the induction hypothesis applies to each of the ω sequences.

By the induction hypothesis, each sequence of moves $(\mathcal{M}_1, \dots, \mathcal{M}_\omega)$ must contain at least $(1/\alpha - 1/\gamma)(\omega^{i-1} + i - 1)n$ pebble placements on levels $\{1, \dots, i-1\}$ of the graph. In total, the entire

sequence of moves $\mathcal{M}$ must then contain at least

$$\omega \cdot [(1/\alpha - 1/\gamma)\,(\omega^{i-1} + i - 1)n]$$

placements on levels $\{1, \ldots, i-1\}$ of the graph.

In addition, since there are n/α vertices on level i that receive pebbles during $\mathcal{M}$ and there are at most n/γ pebbles on the graph in the initial configuration, $\mathcal{M}$ must contain at least $(1/\alpha - 1/\gamma)n$ pebblings of vertices on level i. Thus $\mathcal{M}$ pebbles at least

$$(1/\alpha - 1/\gamma)\,(\omega^{i} + \omega(i-1) + 1)n$$

vertices on levels $\{1, \ldots, i\}$ of the graph. Since $i, \omega \geq 1$, this quantity is at least $(1/\alpha - 1/\gamma)\,(\omega^{i} + i)n$. □

Remark 14. The lower bound of Theorem 13 is not the best possible when G is a stack of degree-reduced graphs (see Definition 11). By keeping track of which degree-reduction gadgets are pebbled, instead of which vertices are pebbled, in the proof of the theorem we can improve the lower-bound of Theorem 13 to $(\delta-1)(1/\alpha-1/\gamma)(\omega^{i}+i)n$, where δ is the degree of the graph G before degree-reduction. This factor of $(\delta-1)$ is important in practice, but we omit the analysis for the sake of brevity.

Finally, we show that adding a few extra edges to a stack of avoiders strengthens the lower-bound.

Corollary 15. *Let $G_{n,d}$ be a stack of $(\alpha, \alpha/\omega, \gamma)$-avoiders. If edges are added to the graph such that there is a path from every vertex on level i to every vertex on level $i-1$, for all $i \in \{1, \ldots, d\}$, then the number of moves required to pebble all vertices on the last level of the graph with at most n/γ pebbles is at least:*

$$T \geq \left(1 - \frac{\alpha}{\gamma}\right) n \sum_{i=0}^{d} (\omega^{i} + i).$$

Proof. For $i \in \{0, \ldots, d\}$, every vertex on level $i-1$ of the graph must be pebbled before every vertex on level i of the graph. To pebble all n vertices on the last level of the graph thus requires completely pebbling each vertex on each layer of the graph at least once. By Theorem 13, we know that pebbling a set of n/α vertices on level i using at most n/γ pebbles takes time at least: $T_i \geq (1/\alpha - 1/\gamma)(\omega^{i}+i)n$ so pebbling all n vertices on level i takes time αT_i. Thus pebbling all n vertices on all d levels takes time at least $T \geq \sum_{i=0}^{d} \alpha T_i$. Substituting in the expression for T_i proves the corollary. □

7 From Pebbling to Space-Hardness

In this section, we complete the security analysis of the single- and double-buffer Balloon constructions. The space-hardness results of Table 1 follow from repeated application of a single proof technique. To illustrate the general principle behind the analysis, we walk through the proof of the claim there is no strategy for computing the N-block r-round single-buffer Balloon function that (1) uses much less than $N/16$ space, (2) makes many fewer than $3^{r+1}N/4$ random oracle queries, and (3) succeeds with high probability. The other results of Table 1 for the single-buffer constructions (for space $N/4$, $N/8$, etc.) follow from an almost identical analysis.

Theorem 16. *Let k denote the block size (in bits) of the underlying cryptographic hash function used in the Balloon constructions. Any algorithm $\mathcal{A}$ that computes the output of the single-buffer N-block r-round Balloon construction, makes $T < \frac{3^{r+1}N}{4}$ random oracle queries, and uses fewer than*

$$\tfrac{N}{16}(k - \log_2(\tfrac{3^{r+1}N}{4})) - k$$

bits of storage space succeeds with probability at most $\frac{T+1}{2^k} + \frac{r}{2^{80}}$ over the choice of the random oracle, provided that $N \geq 2^{10}$ and $\frac{3^{r+1}N}{4} < 2^k - 1$.

As Theorem 16 demonstrates, computing the single-buffer Balloon function with high probability with a very small amount of space causes the time required to blow up *exponentially* in the number of mixing rounds. Even for a small constant number of mixing rounds (e.g., 5 or 8), the time penalties can be prohibitive.

Before proving Theorem 16, we define the *data-dependency graph* of a Balloon computation and then lay out a sequence of claims towards proving the theorem.

Definition 17 (Data-Dependency Graph)**.** We let $G_{\text{single},x} = (V, E)$ denote the directed *data-dependency graph* for the N-block r-round single-buffer Balloon function computed on input x. The vertex set V contains a vertex for each of the N values stored in the main memory buffer at each of the r rounds of mixing: $\{v_1^{(t)}, \ldots, v_N^{(t)} \mid t = 0, \ldots, r\}$. There is an edge $(v_i^{(t)}, v_{i'}^{(t')})$ in $G_{\text{single},x}$ if the ith value in the buffer at mixing round t depends on the i'th value in the buffer in mixing round t'.

Claim 18. *When $N \geq 2^{10}$, the data-dependency graph of one round of the single-buffer Balloon construction is an $(8, \frac{8}{3}, 16)$-avoider with probability at least $1 - 2^{-80}$.*

Proof. To prove the claim, write out the data-dependency graph for a round of mixing. There are three types of edges in the graph. See Figure 9 for a visual depiction of the dependencies and see the referenced steps in Figure 1 for a programmatic depiction. The types are:

- edges pointing to pseudo-randomly chosen neighboring blocks (Step 2b),
- edges pointing to the prior state of the current block (Step 2a), and
- edges pointing to the prior buffer block (Step 2a).

The value generated during ith step of the tth round of mixing depends on $\delta = 20$ other values in the main memory buffer. Now, the union of the first two sets of edges in Figure 9 are just the edges of a localized bipartite graph with left-degree δ. (We introduced localized bipartite graphs in Section 6.3.)

By Claim 10, as long as δ is large enough to ensure that a random bipartite graph with left-degree δ is a good expander, the data dependency graph for one round of mixing will be a good avoider with high probability. By Theorem 6, a random bipartite graph with left-degree 20 is a $(16, \frac{16}{7})$-expander with probability at least $1 - 2^{-80}$ for $N \geq 2^{10}$, so by Claim 10 one such graph is an $(8, \frac{8}{3}, 16)$-avoider with probability at least $1 - 2^{-80}$. Thus, the graph representing the tth mixing round fails to be a good avoider with probability at most $\frac{1}{2^{80}}$. □

We cheated a bit in our definition of the data-dependency graph since we did not account for the fact that each node in the true data dependency graph actually has out-degree two. This is so because the random oracle we use in the construction takes only two data blocks as input so each value computed during the Balloon computation depends only on two other values. We can modify the data-dependency graph of Figure 9 to account for the intermediate values generated while hashing each of the δ blocks together using our two-to-one compression function. This process corresponds exactly to the degree-reduction transformation of Definition 11. Claim 12 demonstrates that the avoider property of a graph is invariant under degree-reduction, so the degree-reduced data-dependency graph is also a good avoider.

Claim 19. *The graph $G_{single,x}$ is a depth-r stack of $(8, \frac{8}{3}, 16)$-avoiders with probability at least $1 - \frac{r}{2^{80}}$.*

Proof. By Claim 18, the probability that the data-dependency graph for a single layer of mixing fails to be an avoider is at most $\frac{1}{2^{80}}$. The entire data-dependency graph consists of a stack of r of these graphs, and the probability that there exists one of them that fails to be a good avoider is, by the Union Bound, at most $\frac{r}{2^{80}}$. □

Claim 20. *Each vertex on level t of the $G_{single,x}$ stack of avoiders has a path to every vertex on level $t-1$ of the stack.*

Proof. Each value stored in the buffer depends on the prior value generated during the computation. These edges are depicted in the center panel of Figure 9. These edges create a path from every vertex on level t of the graph to every vertex on level $t-1$ of the graph. □

Claim 21. *Conditioned on the event that the graph $G_{single,x}$ is a depth-r stack of $(8, \frac{8}{3}, 16)$-avoiders, there is no pebbling strategy for $G_{single,x}$ that uses $S^* < N/16$ pebbles and $T^* < 3^{r+1}N/4$ pebbling moves.*

Proof. By Claims 19 and 20, $G_{\text{single},x}$ is a stack of avoiders satisfying the hypothesis of Corollary 15. Applying the corollary with $\omega = 3$ lets us conclude that pebbling $G_{\text{single},x}$ graphs with at most $N/16$ pebbles requires at least

$$T \geq \left(1 - \frac{8}{16}\right) N \sum_{i=0}^{r} (3^i + i)$$
$$= \left(3^{r+1} + r(r+1) - 1\right) \frac{N}{4}$$

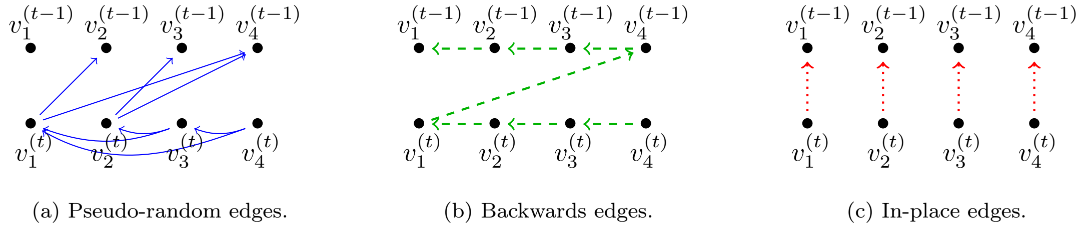
moves, which is greater than $3^{r+1}N/4$ for $r \geq 1$. □

Claim 22. *The output of the single-buffer Balloon construction is the labeling, in the sense of Definition 2, of $G_{single,x}$.*

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

Now we return to prove Theorem 16.

Proof of Theorem 16. Fix an algorithm $\mathcal{A}$ as in the statement of Theorem 16. Conditioned on the event that the graph $G_{\text{single},x}$ is a depth-r stack of $(8, \frac{8}{3}, 16)$-avoiders, Claim 21 indicates that there is no pebbling strategy for $G_{\text{single},x}$ using $S^* < N/16$ pebbles and



(a) Pseudo-random edges.

(b) Backwards edges.

(c) In-place edges.

Figure 9: Components of the data-dependency graph for one single-buffer mixing round. Here, $v_i^{(t)}$ represents the value stored in the ith block in the main memory buffer at the tth mixing round.

$T^* < 3^{r+1}N/4$ pebbling moves. By Claim 22, $\mathcal{A}$ outputs the labeling of $G_{\text{single},x}$. By hypothesis of Theorem 16, $\mathcal{A}$ makes at most $T < 3^{r+1}N/4 < 2^k - 1$ random-oracle queries and uses at most $(N/16)(k - \log_2(3^{r+1}N/4)) - k$ bits of storage space. Theorem 4 then implies that $\mathcal{A}$ succeeds with probability at most $(T+1)/2^k$.

Let E denote the event that $G_{\text{single},x}$ is a stack of $(8, \frac{8}{3}, 16)$-avoiders. Then we have:

$$\Pr[\mathcal{A} \text{ succeeds}] = \Pr[\mathcal{A} \text{ succeeds}|E] \cdot \Pr[E] + \Pr[\mathcal{A} \text{ succeeds}|\neg E] \cdot \Pr[\neg E].$$

Using the fact that probabilities are at most 1,

$$\Pr[\mathcal{A} \text{ succeeds}] \le \Pr[\mathcal{A} \text{ succeeds}|E] + \Pr[\neg E].$$

From Claim 19, $\Pr[\neg E] \le r/2^{80}$ and from the discussion of the prior paragraph, $\Pr[\mathcal{A} \text{ succeeds}|E] \le (T+1)/2^k$. So we have:

$$\Pr[\mathcal{A} \text{ succeeds}] \le \frac{T+1}{2^k} + \frac{r}{2^{80}},$$

which proves the theorem. □

Double-Buffer Construction. The analysis of the double-buffer construction proceeds exactly as above, except that we skip the localizing step in our analysis of the data-dependency graph.

8 Inflexible Matrices

In this section, we introduce *inflexible matrices*, which we will need for our analysis of the linear Balloon functions (Construction III). As the special properties of avoider graphs allowed us to prove pebbling lower bounds in Section 7, the special properties of inflexible matrices will allow us to prove time-space lower bounds for the linear construction.

The characteristic property of an inflexible matrix is that all of its submatrices of a particular size have relatively large rank. This property will be crucial in proving that using the linear transformation defined by an inflexible matrix as the core of Construction 3 yields a space-hard function.

Terminology. In the discussion that follows, when $v = (v_1, \dots, v_n) \in \mathbb{F}^n$ is a vector and $i \in \{1, \dots, n\}$ is an index, we say that v_i is the *ith component of* v. We say that a *standard basis vector* in $\mathbb{F}^n$ is a vector that has value one in a single coordinate and is zero in all other coordinates. For example, the vectors (010) and (100) are two standard basis vectors in $\mathbb{F}_2^3$.

8.1 Definition and Properties

Definition 23 (Inflexible Matrix)**.** For constants $\alpha, \beta, \rho > 1$, we say that a matrix $M \in \mathbb{F}^{n \times n}$ is (α, β, ρ)-inflexible if all submatrices formed from n/α rows and $n - n/\beta$ columns of M are of rank greater than n/ρ over the field $\mathbb{F}$.

For example, a $(4, 2, 8)$-inflexible matrix $M \in \mathbb{F}^{n \times n}$ has the property that every submatrix formed from some $n/4$ rows and $n/2$ columns of M has rank greater than $n/8$ (Figure 10). Similar properties, like the (m, α)-mixing property of Yesha [99], have been useful in prior work on time-space trade-offs.

Inflexible matrices are useful because they represent linear transformations that are "not too lossy," even in the presence of side-information. To see what this means, consider the matrix-vector product $y = Mx$, for a matrix $M \in \mathbb{F}^{n \times n}$ and vector $x \in \mathbb{F}^{n \times 1}$. If M is any matrix in $\mathbb{F}^{n \times n}$ with no special properties, then seeing some set of components of y does not necessarily reveal any information about x.

For example, if M is the all-zeros matrix, the vector $y = Mx$ reveals nothing about x.

In contrast, if M is of full rank, then seeing some $n/2$ components of y reveals $n/2$ linear relations on the components of x. However, if you *already* have some partial information about x, seeing $n/2$ components of y may not give you any extra information about x. For example, if M is the identity matrix and if you already know the first $n/2$ components of x, then seeing the first $n/2$ components of y gives you no new information about x.

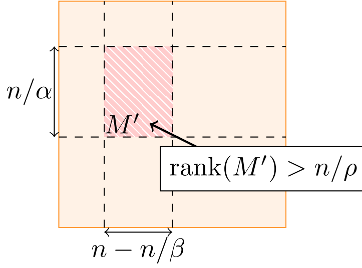


Figure 10: Inflexible matrix.

This is where inflexible matrices help: if M is a $(4, 2, 8)$-inflexible matrix, then *no matter which* $n/2$ components of x you already know, you are guaranteed that *all* sets of $n/4$ components of $y = Mx$ will give you some extra information about x. In particular, you are guaranteed to learn at least $n/8$ additional linear relations on the components of x, beyond those you already know. The following lemma, which we use in the proof of security of the linear hash construction, formalizes this notion.

Lemma 24. *Let $(b_1, \ldots, b_{n/\beta})$ be a set of n/β distinct standard basis vectors over $\mathbb{F}^n$ and let B be an $n/\beta \times n$ matrix formed from these rows. Let R be a matrix formed from some n/α rows of an (α, β, ρ)-inflexible matrix. Then the $(n/\beta + n/\alpha) \times n$ matrix $A = \binom{B}{R}$ has rank greater than $(n/\beta + n/\rho)$.*

Proof. Any set of distinct standard basis vectors is linearly independent, so the first n/β rows of A form a linearly independent set. There are $n - n/\beta$ columns in which the standard basis vectors in B are all zero; let R' be the submatrix formed from these columns of R. Pick some set of $n/\rho + 1$ linearly independent row vectors of R'—such vectors are guaranteed to exist by the properties of an inflexible matrix—and let $\{r_1, \ldots, r_{(n/\rho)}, r_{(n/\rho+1)}\}$ be the rows of the wider matrix R corresponding to these rows of R'.

Now, we claim that for all $i \in \{1, \ldots, (n/\rho + 1)\}$, the set of row vectors $B \cup \{r_1, \ldots, r_i\}$ is a linearly independent set. To prove this claim, append each vector r_i to B one at a time. Towards a contradiction, assume that adding some vector r_i introduces a linear dependency into the set. Then r_i can be written as a linear combination of the B vectors and the vectors $\{r_1, \ldots, r_{i-1}\}$. It is always possible to zero out n/β coordinates of r_i using some combination of the basis vectors in B. However, this still leaves the other $n - n/\beta$ coordinates of r_i to zero out. If there were some linear combination of the vectors $\{r_1, \ldots, r_{i-1}\}$ that, when added to r_i, summed to zero, then there would be a corresponding linear dependency in R' involving the vector r_i. This contradicts the construction of r_i. □

We claimed above that inflexible matrices represent linear transformations that are "not too lossy," even in the presence of side-information. To restate this claim in term of the lemma: let x be a length-n vector and let M be an (α, β, ρ)-inflexible n-by-n matrix. We now consider the linear transformation $y = Mx$. The lemma asserts that even if you know the values of some n/β coordinates of x, every subset of n/α coordinates of y will give you more than n/ρ *additional* independent linear relations on the remaining components of x. In the statement of the lemma, the standard basis vectors represent the indices of the n/β known components of x and the rows R represent the n/α components you are given of y. The rank of the resulting linear system, which we have proven is greater than $n/\beta + n/\rho$, represents the total number of independent linear relations you have after seeing the extra components of $y = Mx$.

8.2 A Simple Probabilistic Construction

The simplest construction of inflexible matrices is a probabilistic one: a random matrix over $\mathbb{F}_2$ is inflexible with high probability.

Lemma 25 ([59], Section 6.3.1)**.** *For large enough n and for all constants $\alpha, \beta \geq 4$, a uniformly random matrix in $\mathbb{F}_2^{n \times n}$ is $(\alpha, \beta, 2\alpha)$-inflexible with high probability.*

The proof of the lemma uses the incompressibility method (see Li and Vitányi for the proof [59, Lemma 6.1.3]). If a random matrix contained a large submatrix of low rank with good probability, then it would be possible to describe a random $n \times n$ matrix in fewer than n^2 bits (asymptotically).

Although these inflexible matrices are easy to generate, they yield an inefficient hash construction overall. Our space-hard function repeatedly multiplies

an inflexible matrix M by a vector x. If we measure the cost of a matrix-vector product by the number of reads an algorithm must make from the main memory buffer to compute it, a completely random matrix yields a $\Theta(n^2)$-time matrix-vector product. By choosing M as a structured or sparse matrix, we can decrease the computation time from quadratic to $O(n \operatorname{polylog} n)$ or even linear time.

8.3 Constructions from Error-Correcting Codes

Generator matrices for certain types of error-correcting codes yield inflexible matrices that are deterministically constructible and allow for sub-quadratic-time matrix-vector multiplication. We cannot do justice to the theory of error-correcting codes here, so we simply state the relevant theorem:

Theorem 26 (MacWilliams and Sloane, p. 321 [60])**.** *Let I be the $n \times n$ identity matrix over a finite field $\mathbb{F}$ and let $G = (I|M)$ be an $n \times 2n$ matrix over $\mathbb{F}$. The matrix G is the generator matrix of a maximum distance separable code if and only if every square submatrix of M is of full rank.*

Thus, if the matrix M generates the parity-check bits for a systematic maximum distance separable code (e.g., a Reed-Solomon code), M is an $(\alpha, \frac{\alpha}{\alpha-1}, \alpha)$-inflexible matrix for all $\alpha > 1$. One way to construct such codes is via Cauchy matrices [19]. Let $\mathbb{F}$ be a finite field with $|\mathbb{F}| \geq 2n$ and let $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$ be $2n$ distinct components of the field. Then the ijth entry of the $n \times n$ Cauchy matrix is:

$$C(a, b)_{ij} = \frac{1}{a_i + b_j}.$$

Another way to construct such codes is via the product of Vandermonde matrices [55]. Further, Shokrollahi et al. show that the generator matrices of good algebraic-geometric codes give rise to matrices with similar properties over a fixed field (one whose size does not vary with n) [87]. Cauchy and Vandermonde matrices have an $O(n \operatorname{polylog} n)$-time matrix-vector product [43], but with sparse matrices, we can get a strictly linear-time construction.

8.4 Fast Probabilistic Construction

With a more nuanced analysis, we can show that a very sparse random binary matrix—one with $O(n)$ non-zero entries—is still inflexible. Computing a matrix-vector product with these matrices takes expected linear time, which is much better than the quadratic-time product given by random matrices.

Let $\mathcal{P}_{n,w}$ denote the distribution that samples a vector v in $\mathbb{F}_2^n$ by choosing w random indices, with replacement, in $\{1, \ldots, n\}$, setting those components in v to 1, and setting all other components to 0.

Then we have the following theorem:

Theorem 27. *Let $w, \rho > 0$ be scalars satisfying*

$$\rho^{1/\rho} \left(\frac{1}{2} + \frac{1.1}{\rho} \right)^{w(1/4 - 1/\rho)} < 2^{-1.82}. \qquad (1)$$

For $n > 20w\rho$, let M be an $n \times n$ matrix over $\mathbb{F}_2$ whose rows are sampled independently from $\mathcal{P}_{n,w}$. Then M is $(4, 2, \rho)$ inflexible with probability at least $1 - c^{-n}$, for some absolute constant $c > 1.005$.

The proof of this theorem appears in Appendix B.

We give a few examples of (w, ρ) pairs satisfying the hypothesis of Theorem 27:

w	8	9	10	11	12	...	21	28	97
ρ	117	49	32	25	20	...	10	8	5

Note that the inequality can never hold when $w < 8$. The theorem gives us a way to construct an $n \times n$ matrix with roughly $10n$ non-zero entries that is still quite inflexible. With all but negligible probability, a matrix sampled from the distribution of Theorem 27 with $w = 10$ will be such that all $n/4$-by-$n/2$-size submatrices have rank at least $n/32$.

Since this construction gives relatively inflexible matrices with a linear-time matrix-vector product, we primarily use this class of matrix for instantiating Construction 3.

8.5 Working over Extension Fields

Later in the analysis, we will work over $\mathbb{F}_{2^k}$, where k is the block length of our underlying compression function. Since we have constructed our inflexible matrices over $\mathbb{F}_2$, we first show that moving to the extension field does not weaken the inflexibility property of a binary matrix.

Definition 28 (Extension of a Matrix)**.** Let M be a matrix over $\mathbb{F}_2$. For every integer $k \geq 1$, the *degree-k extension* of M is the unique 0-1 matrix $\tilde{M}$ over $\mathbb{F}_{2^k}$ such that $\tilde{M}$ has a one in every cell in which M has a one and is zero everywhere else.

Lemma 29. *For all $k \geq 1$, if M is a matrix over $\mathbb{F}_2$ and $\tilde{M}$ is the degree-k extension of M, then* $\text{rank}(M) = \text{rank}(\tilde{M})$.

Proof. Towards a contradiction, assume that there exists a set of rows $(\vec{r}_1, \dots, \vec{r}_\ell)$ that are linearly dependent in $\tilde{M}$ but not in M. There must exist a vector of coefficients $(\alpha_1, \dots, \alpha_\ell) \in \mathbb{F}_{2^k}^\ell$ such that $\sum_i \alpha_i \vec{r}_i = \vec{0} \in \mathbb{F}_{2^k}^n$, where n denotes the number of columns in M. Since the elements of $\vec{r}_i$ are either zero or one by construction, the jth component of the sum is equal to $\sum_{(i \text{ s.t. } \vec{r}_i[j]=1)} \alpha_i = 0 \in \mathbb{F}_{2^k}$. By considering only the least significant bit of the α's we obtain a linear dependency in the rows $(\vec{r}_1, \dots, \vec{r}_\ell)$ over $\mathbb{F}_2$. Thus, the linear dependency in $\tilde{M}$ yields a corresponding linear dependency in M, which is a contradiction.

In the other direction: every linear dependency in M is also a dependency in $\tilde{M}$. This is because a dependency of the rows in M is just a subset of the rows that, when summed together in $\mathbb{F}_2$, equals the zero vector. For any such linear dependency in M, the same property holds in $\tilde{M}$ and thus the set of vectors is also a linear dependency in $\tilde{M}$. □

Corollary 30. *Let M be an (α, β, ρ)-inflexible matrix over $\mathbb{F}_2$. For all $k \geq 1$, the degree-k extension of M is an (α, β, ρ)-inflexible matrix over $\mathbb{F}_{2^k}$.*

Note that the binary matrix resulting from such a transformation still has a fast matrix-vector product: since each of its entries is either zero or one, computing the product simply requires XOR operations.

9 Analysis of the Linear Construction

In this section, we analyze the linear Balloon function. The mixing step of the linear Balloon function alternates between two operations: a linear transformation and a hashing step. As depicted in Figure 2, we can think of these alternating computation steps in terms of layers. Our general strategy is to argue that if an algorithm computes the output of linear Balloon function with little space, then the algorithm cannot possibly store all of the information it needs to produce the vector at the ith layer of the construction without recomputing some values it needs at the $(i-1)$th layer. By the same argument, it must also need to recompute values at the $(i-2)$th layer and the $(i-3)$th layer, and so on, all the way down to the first layer. Using an argument along the lines of Theorem 13, we can show that these recomputations incur a time cost that is exponential in r, the number of rounds.

9.1 Preliminary Lemmata

We will need the following lemma of Dwork et al. [36], which we state without proof. The lemma generalizes the fact that any algorithm that tries to guess the value of an n-bit string with an $(n-1)$-bit "hint" about the string can never succeed with probability better than 1/2. Without loss of generality, we require the hint function h to be deterministic since h can always output the hint that maximizes the predictor's success probability. In the lemma, $\mathcal{H}$ is the space of all possible hints.

Lemma 31 ([36, 38])**.** *Let x be a uniformly random vector in $\mathbb{F}^n$. For all deterministic functions $h : \mathbb{F}^n \to \mathcal{H}$ and for all randomized algorithms $\mathcal{P} : \mathcal{H} \to \mathbb{F}^n$, the probability, over the choice of x and randomness of $\mathcal{P}$, that $\mathcal{P}(h(x)) = x$ is at most $|\mathcal{H}|/|\mathbb{F}|^n$.*

The following lemma, which underpins the security of our construction, states that if M is an inflexible matrix and x is a random vector, producing many components of the matrix-vector product Mx requires making many queries to the components of x.

Lemma 32. *Let $M \in \mathbb{F}^{n \times n}$ be an (α, β, ρ)-inflexible matrix and let $x \xleftarrow{\text{R}} \mathbb{F}^{n \times 1}$. Let $\mathcal{A}$ be a randomized algorithm that takes as input an "hint" in some hint space $\mathcal{H}$ (the hint may depend on x and M), makes at most n/β adaptive queries for the components of x, and finally outputs:*

- *a matrix R formed from n/α distinct rows of M and*
- *a guess $\hat{y}$ of the value of the vector $y = Rx$,*

then $\Pr[y = \hat{y}] \leq |\mathcal{H}|/|\mathbb{F}|^{(n/\rho)+1}$*, over x and $\mathcal{A}$.*

Proof. Towards a contradiction, assume that such an algorithm $\mathcal{A}$ exists. Then we can use $\mathcal{A}$ to build a predictor that guesses some components of x with impossibly high probability, contradicting Lemma 31.

The predictor takes as input a hint in $\mathcal{H}$, which it passes to $\mathcal{A}$. Algorithm $\mathcal{A}$ requests at most n/β components of x and produces a vector $\hat{y} \in \mathbb{F}^{n/\alpha}$ such that $y = \hat{y}$ with probability $p > |\mathcal{H}|/|\mathbb{F}|^{(n/\rho)+1}$. We can assume $\mathcal{A}$ requests exactly n/β components of x, since the predictor can make the remaining queries on $\mathcal{A}$'s behalf.

Assuming that $\mathcal{A}$'s output is correct (i.e., that $y = \hat{y}$), let us write out the linear relations that the predictor now has on the components of x. The answer a_i to $\mathcal{A}$'s ith query for a component of x can be written as the inner product of a standard basis vector $(000\cdots010\cdots0)$ with x. We can collect the n/β query vectors into an (n/β)-by-n matrix B.

Finally, the predictor has the following linear system, where the unknown is the random vector $x \in \mathbb{F}^{n\times 1}$, the vector $a \in \mathbb{F}^{(n/\beta)\times 1}$ consists of the responses to the adversary's queries, and the vector $y \in \mathbb{F}^{(n/\alpha)\times 1}$ consists of the adversary's output: $\binom{B}{R} \cdot x = \binom{a}{y}$. By Lemma 24, the matrix on the left side of this equation has rank at least $(1/\rho+1/\beta)n+1$.

The predictor now has a linear system with $(1/\rho + 1/\beta)n + 1$ independent linear equations and n unknowns. If we assume that $\mathcal{A}$'s output is correct, then the predictor can make a random choice for the remaining components of x, consistent with the linear constraints it already has. The probability that $\mathcal{A}$ succeeds will be:

$$\Pr[\mathcal{P} \text{ succeeds} \mid y = \hat{y}] = \frac{|\mathbb{F}|^{(n/\rho)+(n/\beta)+1}}{|\mathbb{F}|^n}.$$

The predictor's probability of success overall is:

$$\begin{aligned}\Pr[\mathcal{P} \text{ succeeds}] &= \Pr[y = \hat{y}] \cdot \Pr[\mathcal{P} \text{ succeeds} \mid y = \hat{y}] \\ &> \frac{|\mathcal{H}|}{|\mathbb{F}|^{(n/\rho)+1}} \cdot \frac{|\mathbb{F}|^{(n/\rho)+(n/\beta)+1}}{|\mathbb{F}|^n} \\ &= \frac{|\mathcal{H}| \cdot |\mathbb{F}|^{n/\beta}}{|\mathbb{F}|^n}.\end{aligned}$$

But this contradicts Lemma 31, since the predictor takes a hint in $\mathcal{H}\times\mathbb{F}^{n/\beta}$ (the hint along with the query responses) and guesses the value of the vector x with probability better than allowed by the lemma. ☐

9.2 The Main Theorem

Recall that the computation of the linear Balloon function involves repeated operations on a vector $x \in \mathbb{F}^n$. Each round of mixing involves computing the matrix-vector product $y \leftarrow Mx$ and then hashing the resulting vector. In practice, the matrix M is the degree-k extension of an inflexible matrix constructed over $\mathbb{F}_2$. That is, M is a 0/1 matrix defined over $\mathbb{F}_{2^k}$. By Corollary 30, M is an inflexible matrix. Note that the matrix-vector product Mx computed in each step of the algorithm requires only XORs, since M is a 0/1 matrix over $\mathbb{F}_{2^k}$. If M is sparse (as the matrices we construct are), computing Mx will be quite fast.

Let the operator $\mathbf{h}_i : \mathbb{F}^n_{2^k} \to \mathbb{F}^n_{2^k}$ denote the operation of hashing each component in a vector: $\mathbf{h}_i(x) = (H(i, 1, x_1), \dots, H(i, n, x_n))$. To facilitate the analysis, define:

$$\begin{aligned} & x^{(0)} \leftarrow \mathbf{h}_0(IV) \\ y^{(i)} \leftarrow Mx^{(i)} \quad ; \quad & x^{(i+1)} \leftarrow \mathbf{h}_i(y^{(i)}) \end{aligned}$$

Here $IV \in \mathbb{F}^n_{2^k}$ is the initial vector derived from the password and salt and $x^{(0)}$ is the output of the "extract" step of the computation. The sequence of vectors generated during the course of the computation is then: $x^{(0)}, y^{(0)}, x^{(1)}, y^{(1)}, x^{(2)}, \dots$.

Fix a random oracle H and an input to the construction x. At the ith mixing iteration, the jth invocation of the random oracle has the form $H(i, j, y^{(i)}_j)$, where $y^{(i)}_j$ is the jth coordinate of the vector $y^{(i)}$. We say that a random oracle query is *correct* (with respect to the oracle H and input x) if it has the form $H(i, j, \sigma)$, where $\sigma = y^{(i)}_j$ and we say that a random oracle query is *incorrect* otherwise. We call a query of the form $H(i, \cdot, \cdot)$, a *level-i query* to the random oracle. We say that a set of level-i queries $\{H(i, j_1, \cdot), H(i, j_2, \cdot), \dots\}$ is *distinct* if the values $\{j_1, j_2, \dots\}$ are distinct.

The intuition behind the security of our construction is that every small-space algorithm that can make many correct level-i queries will have to make many correct level-$(i-1)$ queries with high probability. Using an argument along the lines of that in Section 7, we show that every strategy for computing the outputs at the last level with small-space will cause the time cost to blow up.

We model the adversarial algorithm $\mathcal{A}$ as a Turing machine with oracle access to a hash function $H : \{0,1\}^k \to \{0,1\}^k$. When we say that the state of $\mathcal{A}$ can be described in a certain number of bits, we mean that the entire configuration of the Turing machine fits into this many bits (as in the analysis of Dziembowski et al. [38]).

The statement of the theorem is necessarily probabilistic: there is always some chance that an algorithm will get lucky and guess the output to the linear Balloon function without doing much computation.

Theorem 33. *Let k denote the output size of the random oracle, in bits. Let the linear Balloon function be instantiated with an $(\alpha, \alpha/\omega, \rho)$-inflexible matrix $M \in \mathbb{F}^{n\times n}_{2^k}$, for some integer $1 \le \omega < \alpha$ such that $\alpha\cdot\omega < n$. Let $\mathcal{A}$ be an algorithm whose state at every step can be described in at most $\frac{nk}{2\rho}$ bits.*

For all $i \in \{0, \ldots, d\}$, let $\mathcal{Q}_i$ be a random variable representing the ordered sequence of random oracle queries that $\mathcal{A}$ makes on levels $\{0, \ldots, i\}$ in some sequence of consecutive time steps. Then for all integers q_i such that (a) $q_i < \omega^i n/\alpha$, (b) $q_i < 2^{\frac{\alpha k}{2\rho}}$, and (c) $\Pr\left[|\mathcal{Q}_i| = q_i\right] > 0$, we have that

$$\Pr\left[\begin{array}{c}\mathcal{Q}_i \text{ contains at} \\ \text{least } n/\alpha \text{ correct} \\ \text{distinct level-}i \text{ queries}\end{array} \;\middle|\; |\mathcal{Q}_i| = q_i\right] \leq \frac{q_i}{2^k},$$

over the randomness of $\mathcal{A}$ and the random oracle H.

Proof. By induction on i.

Base Case ($i = 0$). In this case, the statement is vacuously true, since if $\mathcal{A}$ makes $q_0 < n/\alpha$ oracle queries during some time period, its probability of making n/α queries during this same time period is zero.

Induction Step. Let C_i denote the event that the adversary makes n/α correct distinct level-i queries. By way of contradiction, assume that there exists an adversary $\mathcal{A}$ and a constant q_i as in the theorem such that $\mathcal{A}$ makes q_i queries at or below level i (such that at least n/α of these queries are correct and distinct) with non-zero probability p_{win} and that

$$p_{\text{win}} = \Pr\left[C_i \;\middle|\; |\mathcal{Q}_i| = q_i\right] > \frac{q_i}{2^k}. \tag{2}$$

In the following discussion, all probabilities are implicitly conditioned on $|\mathcal{Q}_i| = q_i$.

Let C_{i-1} denote the event that $\mathcal{Q}_i$ contains $\omega n/\alpha$ correct distinct queries on level-$(i-1)$. We can then write the probability that the adversary succeeds as:

$$\begin{aligned} p_{\text{win}} = \Pr\left[C_i | C_{i-1}\right] \cdot \Pr[C_{i-1}] & \\ + \Pr\left[C_i | \neg C_{i-1}\right] \cdot \Pr[\neg C_{i-1}]. & \quad (3)\end{aligned}$$

Towards bounding the probability p_{win}, we prove the following claim:

Claim. *The probability $\Pr[C_i | \neg C_{i-1}]$ is bounded by*

$$\Pr\left[C_i | \neg C_{i-1}\right] \leq 2^{-k}. \tag{4}$$

Proof of Claim. To prove the claim, we show that if there exists a small-space algorithm $\mathcal{A}$ for which $\Pr[C_i | \neg C_{i-1}] > 2^{-k}$, then there exists an algorithm $\mathcal{B}$ that contradicts Lemma 32.

To apply Lemma 32, define the vector x as the vector in $\mathbb{F}_{2^k}^n$ consisting of the n strings that the random oracle would output in response to the n correct level-$(i-1)$ queries. (The correct queries are always distinct, since we prepend each query with a counter indicating the index of the query.) Define the vector y as the vector in $\mathbb{F}_{2^k}^{n/\alpha}$ consisting of the n/α responses to the correct level-i queries that $\mathcal{A}$ makes of the random oracle. Since the vector x consists of outputs of the random oracle on distinct queries, it is a uniformly random vector over $\mathbb{F}_{2^k}^n$. By the construction of x and y, we can write $y = Rx$, where R is a matrix in $\mathbb{F}_{2^k}^{(n/\alpha) \times n}$ consisting of the n/α rows of the inflexible matrix M corresponding to the n/α outputs of $\mathcal{A}$.

The algorithm $\mathcal{B}$ takes as a hint (1) the state of $\mathcal{A}$ at the beginning of the sequence of time steps defined in the statement of the theorem, and (2) a list of n/α integers in the range $\{1, \ldots, q_i\}$. These integers indicate to the predictor which n/α of the adversary's level-i queries are correct (without these hints, the predictor could not distinguish correct level-i queries from incorrect ones).

By the hypothesis of the theorem, the state of $\mathcal{A}$ fits into $\frac{nk}{2\rho}$ bits. Also by the hypothesis of the theorem, and the number of queries q_i has an upper bound of $2^{\frac{\alpha k}{2\rho}}$. The entire hint space thus has size at most

$$2^{\frac{nk}{2\rho}} \cdot q_i^{\frac{n}{\alpha}} = 2^{\frac{nk}{2\rho}} \cdot (2^{\frac{\alpha k}{2\rho}})^{\frac{n}{\alpha}} = 2^{\frac{nk}{\rho}}.$$

The predictor $\mathcal{B}$ runs by first executing the algorithm $\mathcal{A}$. Whenever $\mathcal{A}$ makes a random oracle query, $\mathcal{B}$ forwards the query to the random oracle. Since $\mathcal{B}$ faithfully simulates $\mathcal{A}$'s real interaction with the random oracle, $\mathcal{A}$ will, over the course of its execution, make n/α correct level-i queries with probability greater than 2^{-k}.

The algorithm $\mathcal{B}$ then uses the n/α integers in the hint to determine which n/α of $\mathcal{A}$'s level-i queries were correct. Given these integers, $\mathcal{B}$ learns both the indices at which $\mathcal{A}$ made the correct level-i queries and the values of these queries. The algorithm $\mathcal{B}$ then can produce a subset R of n/α rows of the inflexible matrix M and a guess $\hat{y} \in \mathbb{F}^{n/\alpha}$ of the value Rx. The algorithm $\mathcal{B}$ succeeds with exactly the probability that algorithm $\mathcal{A}$ does, which is greater than 2^{-k}.

To derive a contradiction, we apply Lemma 32 with $|\mathbb{F}| = 2^k$ and $|\mathcal{H}| \leq 2^{\frac{nk}{\rho}}$ to bound the success probability of $\mathcal{B}$. Lemma 32 applies because the algorithm $\mathcal{A}$, and thus the predictor $\mathcal{B}$, makes at most wn/α correct level-$(i-1)$ queries. (The predictor $\mathcal{B}$ may make many more incorrect level-$(i-1)$ queries than this, but incorrect queries give $\mathcal{B}$ no information about the bits it is trying to predict.) By Lemma 32, $\mathcal{B}$'s success

probability is at most

$$\frac{|\mathcal{H}|}{|\mathbb{F}|^{(n/\rho)+1}} = \frac{2^{\frac{nk}{\rho}}}{2^{\frac{nk}{\rho}} 2^k} = \frac{1}{2^k}.$$

But this is a contradiction since we assumed that $\mathcal{A}$, and therefore $\mathcal{B}$, have success probability greater than 2^{-k}.

Thus we conclude that $\Pr[C_i|\neg C_{i-1}] \le 2^{-k}$ and this completes the proof of the claim. □

Now we return to the proof of Theorem 33. By Equations (3) and (4), we now have:

$$p_{\text{win}} \le \Pr[C_i|C_{i-1}] \cdot \Pr[C_{i-1}] + 2^{-k}$$

and by applying Bayes' Theorem, we get:

$$\begin{aligned} &= \Pr[C_{i-1}|C_i] \cdot \Pr[C_i] + 2^{-k} \\ &\le \Pr[C_{i-1}|C_i] + 2^{-k}. \end{aligned} \tag{5}$$

To complete the argument we need only to bound $\Pr[C_{i-1}|C_i]$. We do so with the following claim:

Claim. *The probability* $\Pr[C_{i-1}|C_i]$ *is bounded by:*

$$\Pr[C_{i-1}|C_i] \le \frac{q_i - n/\alpha}{2^k}. \tag{6}$$

Proof of Claim. Our strategy is to split $\mathcal{Q}_i$ into pieces in such a way that we can apply the induction hypothesis to each piece. Towards this goal, we first observe that, conditioned on C_i, at least n/α of the queries in $\mathcal{Q}_i$ are correct distinct level-i queries.

Conditioned on C_i, the set of queries $\mathcal{Q}_{i-1}$ on levels $\{0, \ldots, i-1\}$ then has size at most

$$\begin{aligned} |\mathcal{Q}_{i-1}| = q_{i-1} &\le q_i - n/\alpha \\ &< \omega^i n/\alpha - (n/\alpha) \end{aligned}$$

after removing the level-i queries.

We can define integers $(q_{i-1}^{(1)}, \ldots, q_{i-1}^{(\omega)})$ such that

(i) $q_{i-1}^{(1)} + \cdots + q_{i-1}^{(\omega)} = q_{i-1}$ and

(ii) $q_{i-1}^{(j)} < \omega^{i-1} n/\alpha$ for all $j \in \{1, \ldots, \omega\}$. (Here we used the fact that $\alpha\omega < n$.)

Towards applying the induction hypothesis, we now define a set of ω random variables $(\mathcal{Q}_{i-1}^{(1)}, \ldots, \mathcal{Q}_{i-1}^{(\omega)})$. We define $\mathcal{Q}_{i-1}^{(1)}$ to be the random variable representing the first $q_{i-1}^{(1)}$ random oracle queries in $\mathcal{Q}_{i-1}$. We define $\mathcal{Q}_{i-1}^{(2)}$ to be the next $q_{i-1}^{(2)}$ random oracle queries in $\mathcal{Q}_{i-1}$. We continue the definitions this way until defining $\mathcal{Q}_{i-1}^{(\omega)}$ to be the last $q_{i-1}^{(\omega)}$ random oracle queries in $\mathcal{Q}_{i-1}$.

We know that the total size of all subsets ($q_{i-1} = \sum_{j=1}^{\omega} |\mathcal{Q}_{i-1}^{(j)}|$) is at most $q_i - n/\alpha$.

Let S_j be the event that $\mathcal{Q}_{i-1}^{(j)}$ contains at least n/α correct distinct level-$(i-1)$ queries. *At least* one of the events $(S_1, \ldots, S_\omega)$ must occur for C_{i-1} to occur, so we can use the Union Bound to state:

$$\Pr[C_{i-1}|C_i] \le \Pr[S_1|C_i] + \cdots \Pr[S_\omega|C_i].$$

We now apply the induction hypothesis to bound the sum on the right-hand side. For any set $\mathcal{Q}_{i-1}^{(j)}$, we can apply the induction hypothesis on level $i-1$ to conclude that: $\Pr[S_j|C_i] \le q_{i-1}^{(j)}/2^k$, so

$$\Pr[C_{i-1}|C_i] \le \frac{1}{2^k}\left(\sum_{i=1}^{\omega} q_{i-1}^{(j)}\right) \le \frac{q_{i-1}}{2^k} \le \frac{q_i - n/\alpha}{2^k}.$$

This concludes the proof of the claim. □

We now substitute Inequality (6) into Equation (5) to give:

$$p_{\text{win}} \le \frac{q_i - n/\alpha}{2^k} + \frac{1}{2^k} \le \frac{q_i}{2^k}.$$

This is a contradiction to our assumption (2) on $\mathcal{A}$ and proves the theorem. □

9.3 Putting It All Together

The last step of the security analysis is to relate the statement of Theorem 33 back to the design of Construction 3.

Theorem 34. *Let k, M, and n be as in Theorem 33. In particular, M is an $(\alpha, \alpha/\omega, \rho)$-inflexible matrix in $\mathbb{F}_{2^k}^{n\times n}$, for some integer $1 \le \omega < \alpha$ such that $\alpha \cdot \omega < n$.*

Let $\mathcal{A}$ be an algorithm that uses at most $\frac{nk}{2\rho}$ bits of storage space and that makes at most q random oracle queries during its execution. Then, provided that $q < \omega^r n - \alpha$, and $q < \alpha(2^{\frac{\alpha k}{2\rho}} - 1)$, the probability that $\mathcal{A}$ correctly computes output of the r-round n-block linear Balloon function is at most $\frac{q+1}{2^k}$.

Proof. Let C denote the event that $\mathcal{A}$ makes n correct distinct level-r queries during its execution. We can write:

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

Conditioned on $\neg C$, the adversary's probability of success is at most 2^{-k}. This is because the output of the linear Balloon function consists of the n outputs of the correct level-r queries summed modulo two. If the adversary makes fewer than n correct distinct level-r queries, it has at best a 2^{-k} chance of guessing the correct value. So we have:

$$\Pr[\mathcal{A} \text{ succeeds}] \leq \Pr[C] + 2^{-k}. \tag{7}$$

We can now use Theorem 33 to bound $\Pr[C]$. To apply the theorem, divide the sequence of $\mathcal{A}$'s random oracle queries into α distinct subsequences: $\mathcal{Q}_1, \dots, \mathcal{Q}_\alpha$. We choose the subdivisions such that for $i \in \{1, \dots, \alpha\}$, (a) $\mathcal{Q}_i$ contains fewer than $\omega^r n/\alpha$ queries and (b) $\mathcal{Q}_i$ contains fewer than $2^{\frac{\alpha k}{2\rho}}$ queries. Such a subdivision is possible since the size of the entire sequence of queries $\mathcal{Q}$ is bounded by the hypothesis of the theorem.

We can apply Theorem 33 to each of the subsequences $\mathcal{Q}_1, \dots, \mathcal{Q}_\alpha$. For each subsequence $\mathcal{Q}_i$, Theorem 33 asserts that the probability that $\mathcal{Q}_i$ contains at least n/α correct distinct level-r queries is bounded by $|\mathcal{Q}_i|/2^k$. By the Union Bound, the probability that there exists some $i \in \{1, \dots, \alpha\}$ such that $\mathcal{Q}_i$ contains more than n/α distinct correct level-r queries is at most

$$\frac{|\mathcal{Q}_1| + \dots + |\mathcal{Q}_\alpha|}{2^k} = \frac{q}{2^k}.$$

Thus, $\Pr[C] \leq \frac{q}{2^k}$. Substituting this expression for $\Pr[C]$ into Equation (7) gives:

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

□

To complete the analysis, we show how to apply Theorem 34 to obtain the space-hardness result of Table 1. The following theorem demonstrates that if an adversary tries to compute the linear Balloon function with fewer than $N/128$ blocks of storage space, then the adversary will need to make at least (roughly) $\min\{2^{r-1}N, 2^{\frac{k}{8}+3}\}$ queries to compute the function with better than miniscule probability.

Theorem 35. *Let the linear Balloon function be instantiated with an N-block main memory buffer, where each block is k bits in length. Let $n = N/2$ and let $n \geq 1024$. Let M be an $n \times n$ matrix over $\mathbb{F}_{2^k}^{n \times n}$ sampled from the distribution in which $w = 10$ ones are placed independently and uniformly at random into each row.*

Let $\mathcal{A}$ be an algorithm that uses at most $\frac{N}{128}$ bits of storage space and that makes at most q random oracle queries during its execution. Then, provided that $q < 2^{r-1}N - 8$, and $q < 8(2^{\frac{k}{8}} - 1)$, the probability that $\mathcal{A}$ correctly computes output of the r-round N-block linear Balloon function is at most $\frac{q+1}{2^k} + 2^{-128}$.

Here, N is the total amount of space required for the main memory buffer. Since the linear construction actually uses two buffers—a source and destination buffer—the size of each of these buffers is only $n = N/2$ blocks. Thus the matrix M used in the construction is only an $n \times n$ matrix.

We prove a straightforward claim before proving Theorem 35.

Claim 36. *The matrix M is a $(4, 2, 32)$-inflexible matrix with probability at least $1 - 2^{-128}$.*

Proof. First use Theorem 27 with $w = 10$ and $n \geq 1024$ to show that an $n \times n$ matrix over $\mathbb{F}_2$ with 10 ones thrown independently and uniformly at random into each row is a $(4, 2, 32)$-inflexible matrix with probability at least $1 - 2^{-128}$. Then apply Corollary 30 to show that the corresponding matrix over $\mathbb{F}_{2^k}$ is equally inflexible with the same probability. □

Proof of Theorem 35. Let E be the event that the matrix M is a $(4, 2, 32)$-inflexible matrix. We can write:

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

By Claim 36, we know that $\Pr[E] \geq 1 - 2^{-128}$, so:

$$\Pr[\mathcal{A} \text{ succeeds}] \leq \Pr[\mathcal{A} \text{ succeeds}|E] + 2^{-128}. \tag{8}$$

Conditioned on M being an inflexible matrix, we can apply Theorem 27 to bound the probability that $\mathcal{A}$ succeeds at computing the output of the r-round N-block linear Balloon function. We apply Theorem 27 with $\alpha = 4$, $\omega = 2$, and $\rho = 32$. Under the conditions of Theorem 35, Theorem 27 gives that:

$$\Pr[\mathcal{A} \text{ succeeds}|E] \leq \tfrac{q+1}{2^k}.$$

Substituting this inequality into (8) completes the proof. □

10 Experimental Evaluation

In the prior sections, we have argued that the Balloon functions are space-hard. In this section, we

argue that they are also practical. To compare the performance of the Balloon hash functions with the state-of-the-art methods for password hashing, we implemented the three Balloon variants and evaluated them on a number of benchmarks.

Experimental Set-up. For the initial "extract" step in all constructions, we hash the input and then use AES-256 in CTR mode to fill the buffer initially. We used the OpenSSL implementations (version 1.0.1f) of SHA-512 and AES-256 and we used the reference implementations of the other compression functions. We compiled the code for our timing results with clang version 3.4.1 using the `-O3` option. We use optimized versions of the underlying cryptographic primitives where available, but the core Balloon hash 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 at 3.40 GHz with 8 GiB of RAM for our performance benchmarks. This CPU uses uses the most recent Intel microarchitecture (Skylake). Since our operating system does not yet support accessing the performance counters on Skylake processors, we recorded the L1 and L2 cache miss rates (right side, Figure 13) on an older machine with an Intel Xeon E5620 CPU at 2.40 GHz (Westmere) with 48 GiB of RAM. Both CPUs support the AES-NI instructions.

We set the parameters of the schemes so as to achieve the bounds listed in Table 1 and we average all of our measurements over 32 trials.

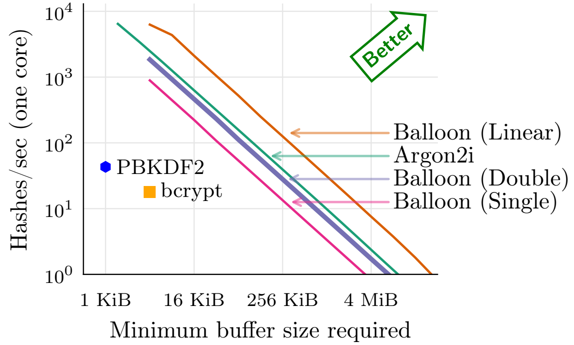


Figure 11: Throughput of the Balloon schemes and standard password hashing schemes, as the minimum required buffer size varies. Argon2i and Balloon functions make three passes over the memory ($r = 3$).

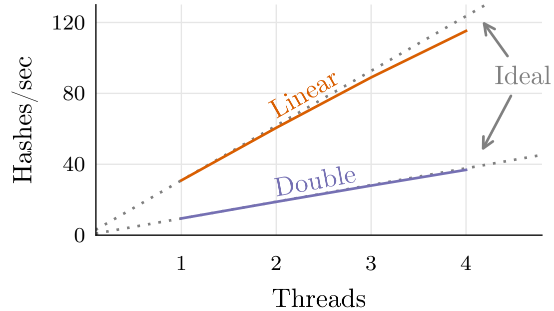


Figure 12: Hashing speeds for the linear Balloon construction (1 MiB buffer, three mixing rounds) as the number of threads varies.

10.1 Authentication Throughput

The goal of a space-hard password hash function is to *use as much working space as possible quickly as possible* over the course of its computation. To evaluate the effectiveness of the Balloon hashes 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 11 shows the minimum buffer size required to compute each hash function with high probability with no computational slowdown, for a variety of password hashing functions. We configure the Balloon and Argon2i functions to use SHA-3 (Keccak-1600, rate = 1344) as their underlying cryptographic compression function. 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 Section 5.3. We prefer Keccak-1600 because it has a large internal state size (1600 bits) and thus allows us to use a large block size to maximize data locality.

The charted results for Argon2i incorporate the fact that an adversary can compute three-pass Argon2i in a factor of $e \approx 2.72$ less working space than the defender must allocate for the computation (see Appendix A). For comparison, we also plot the space usage of two non-space-hard password hashing functions, bcrypt [81] (with cost = 12) and PBKDF2-SHA512 [49] (with 10^5 iterations).

If we assume that an authentication server must perform 100 hashes per second per core, Figure 11 shows that it would be possible to use our linear construction with a 256 KiB buffer, running the construction for three rounds. At the same authentication rate, Argon2i requires the attacker to use a

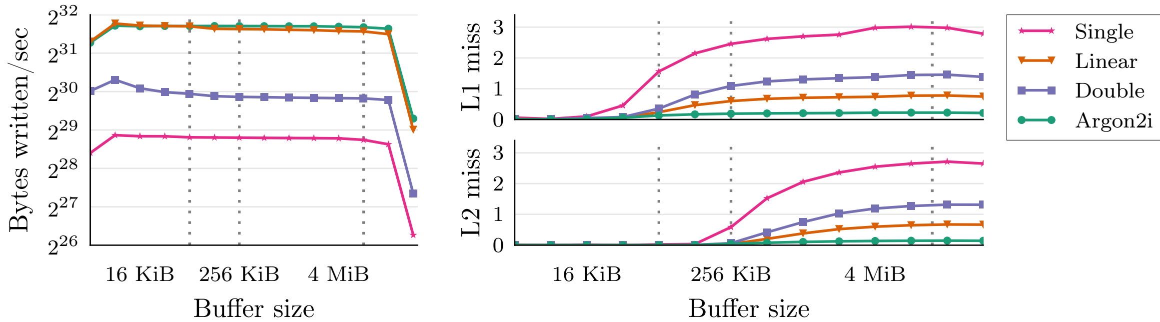


Figure 13: Hashing throughput (left) and the normalized number of cache misses (right) for the three Balloon variants. The dashed vertical lines indicate the sizes of the L1, L2, and L3 cache.

smaller buffer (roughly 100 KiB), which is on par with the performance of the double-buffer Balloon construction. By exploiting all four compute cores at once, our linear construction can handle a 1 MiB buffer at roughly 115 hashes per second (Figure 12). When using all cores, the linear construction can fill each core's L2 cache (256 KiB) simultaneously at 489 hashes per second.

10.2 Balloon Mixing Variants

The left-hand chart in Figure 13 shows the rate at which the Balloon functions can fill memory, as the size of the buffer varies. Our linear construction is comparable with Argon2i construction in terms of memory write throughput. As expected, the computational overhead of invoking the compression function $10N$ or $20N$ times per mixing round slows down the double- and single-buffer constructions relative to the linear construction. As long as the size of the entire buffer fits in L3 cache, the rate at which the functions can fill memory is roughly constant. Once the buffer spills into main memory, the performance rapidly degrades due to the latency of memory reads.

The right-hand side of Figure 13 shows the average number of L1 and L2 cache misses during each algorithm's execution, normalized by the number of bytes written during the computation. The dashed vertical lines in Figure 13 indicate the sizes of the L1, L2, and L3 caches in the machine on which we ran the experiments. The Balloon hashes make many more random reads to memory than the Argon2i hash function does (by roughly a factor of 20×), which explains why the Balloon hashes cause relatively many cache misses. That said, the fact that the Balloon hashes make many random reads to memory is critical to its space-hardness properties, so the increase in cache misses may be inevitable to some extent.

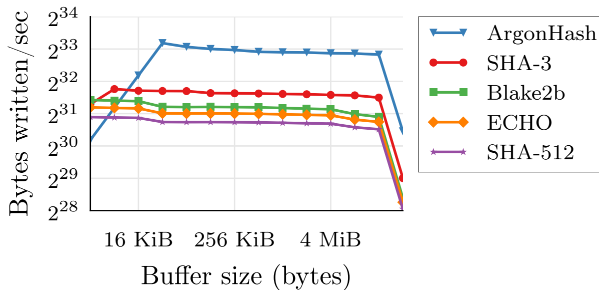


Figure 14: Throughput for the linear Balloon variant when instantiated with different compression functions.

10.3 Compression Function

Finally, Figure 14 shows the result of instantiating the Balloon linear construction with five different compression functions: SHA-3 [14], Blake2b [6], SHA-512, ECHO (a SHA-3 candidate that exploits the AES-NI instructions) [11], and ArgonHash (the custom compression function defined in the Argon specification [15]). ArgonHash operates on 8192-bit blocks, SHA-3 (with rate = 1344) operates on 1344-bit blocks, and we configure the other hash functions to use 512-bit blocks. As Figure 14 demonstrates, the compression functions with larger block sizes dramatically outperform the compression functions using smaller blocks.

The reason is locality: increasing the block size decreases the number of random reads required to take

a pass over a fixed-size buffer. For example, consider a buffer of $N \times B$-bit blocks. One mixing iteration of our single-buffer construction requires making roughly $20N$ random reads to the buffer. If we increase the block size to $B' = 2B$ while holding the buffer size constant at NB bits, the number of random reads decreases to: $20\frac{NB}{B'} = 10N$. If we double the block size again, we can cut the number of random reads down by another factor of two.

Although ArgonHash is the fastest of the cryptographic compression functions, we hesitate to use it because (1) it is non-standard so has not been the subject of public cryptanalysis and (2) it is much weaker than a traditional cryptographic hash function. In particular, ArgonHash is not collision-resistant [15, Section 5.3], so modeling it as a random oracle seems particularly problematic. That said, it is possible to instantiate the Balloon hash functions with any cryptographic compression function, so users who have confidence in the ArgonHash design can deploy the Balloon construction with ArgonHash.

11 Conclusions and Open Questions

In this paper, we have introduced the Balloon family of hash functions. These are the first space-hard functions that are provably space-hard, are based on standard cryptographic primitives, exhibit a password-independent memory access pattern, and are practically efficient. With novel analysis techniques, we have shown that it is possible reconcile the need for fast password-hashing functions with the desire for a formal security analysis.

This work raises a number of open questions:

Space-Hardness Under Batching. We prove that it is hard to compute a *single* instance of the Balloon functions in a small amount of working space. An important—and apparently non-trivial—next question is whether computing P instances of the Balloon functions in parallel requires at least a factor of P more space or a factor of P more time than computing a single instance.

Space-Hardness Over Time. We prove that computing the Balloon functions quickly requires a large amount of storage space *at some point* during the computation. A stronger, and more desirable, property to prove would be that the Balloon functions use a large amount of storage space *at most points* during the computation.

Space-Hardness with Parallelism. Our proofs use a sequential model of computation, in which each query to the random oracle happens in sequence. To model an attacker who uses multiple hashing engines running in parallel, it would be ideal to execute show space-hardness properties in the *parallel random-oracle model*. With an involved argument, Alwen and Serbinenko are able to prove this sort of "space-hardness with parallelism" property for a certain type of function [4]. Their analysis also addresses the issues of batching-resistance and space-hardness over time discussed in the prior two paragraphs. Unfortunately, the function that they define is efficient in an asymptotic sense but is not practical. Is it possible to apply their analysis techniques to our (or other practically efficient) functions?

Analysis of Argon. Are there better small-space strategies for computing Argon2i than the ones of Appendix A? Or can we use pebbling arguments to prove a time-space lower bound for Argon?

Inflexible Matrices. Are there constructions of inflexible matrices with a linear-time product that have better constants than the ones given in Theorem 27? It might also be possible to replace the sparse matrix of the linear construction with a linear-sized circuit that implements an inflexible [linear] transformation. Do such circuits yield linear transformations with better efficiency or inflexibility properties?

As these open problems demonstrate, the study of practical space-hard functions provides a forum for the fruitful interplay of theory and practice. With the design and implementation of the Balloon functions, we have endeavored to build space-hard password-hashing functions that are both practically and provably good.

Acknowledgements

We would like to thank Josh Benaloh, Bryan Parno, Yan Michalevsky, Sergey Yekhanin, and Greg Zaverucha for comments on early versions of this work. Greg Valiant offered helpful advice on how to prove the matrix properties needed for the linear construction and Ali Mashtizadeh gave useful tips on the experimental set-up. Joe Bonneau, Greg Hill, David Mazières, Keith Winstein gave suggestions that improved the writing.

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.

References

[1] Martín Abadi, Mike Burrows, Mark Manasse, and Ted Wobber. Moderately hard, memory-bound functions. *ACM Transactions on Internet Technology*, 5(2):299–327, 2005.

[2] Martin R. Albrecht, Benedikt Driessen, Elif Bilge Kavun, Gregor Leander, Christof Paar, and Tolga Yalçın. Block ciphers–focus on the linear layer (feat. PRIDE). In *CRYPTO*, pages 57–76. Springer, 2014.

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

[4] Joël Alwen and Vladimir Serbinenko. High parallel complexity graphs and memory-hard functions. In *STOC*, pages 595–603, 2015.

[5] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of space: When space is of the essence. In *Security and Cryptography for Networks*, pages 538–557. Springer, 2014.

[6] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In *Applied Cryptography and Network Security*, pages 119–135. Springer, 2013.

[7] Adam Back. Hashcash–a denial of service counter-measure. http://www.cypherspace.org/hashcash/, May 1997. Accessed 9 November 2015.

[8] Mihir Bellare, Alexandra Boldyreva, and Adriana Palacio. An uninstantiable random-oracle-model scheme for a hybrid-encryption problem. In *EUROCRYPT 2004*, pages 171–188. Springer, 2004.

[9] Mihir Bellare, Thomas Ristenpart, and Stefano Tessaro. Multi-instance security and its application to password-based cryptography. In *CRYPTO*, pages 312–329, 2012.

[10] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, pages 62–73. ACM, 1993.

[11] Ryad Benadjila, Olivier Billet, Henri Gilbert, Gilles Macario-Rat, Thomas Peyrin, Matt Robshaw, and Yannick Seurin. SHA-3 proposal: ECHO. Submission to NIST (updated), 2009.

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

[13] Daniel J Bernstein. The Salsa20 family of stream ciphers. In *New stream cipher designs*, pages 84–97. Springer, 2008.

[14] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family. *Submission to NIST (Round 2)*, 2009.

[15] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2 design document (version 1.2.1), October 2015.

[16] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Fast and tradeoff-resilient memory-hard functions for cryptocurrencies and password hashing. Cryptology ePrint Archive, Report 2015/430, 2015. http://eprint.iacr.org/.

[17] Alex Biryukov and Dmitry Khovratovich. Tradeoff cryptanalysis of memory-hard functions. Cryptology ePrint Archive, Report 2015/227, 2015. http://eprint.iacr.org/.

[18] Matt Bishop and Daniel V Klein. Improving system security via proactive password checking. *Computers & Security*, 14(3):233–249, 1995.

[19] Johannes Blömer, Malik Kalfane, Marek Karpinski, Richard Karp, Michael Luby, and David Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, ICSI, August 1995.

[20] Joseph Bonneau. *Guessing human-chosen secrets*. PhD thesis, University of Cambridge, 2012.

[21] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. May 2015.

[22] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *CHES 2006*, pages 201–215. Springer, 2006.

[23] Xavier Boyen. Halting password puzzles. In *USENIX Security*, 2007.

[24] Ran Canetti, Oded Goldreich, and Shai Halevi. The Random Oracle Methodology, revisited. *Journal of the ACM*, 51(4):557–594, 2004.

[25] Ran Canetti, Shai Halevi, and Michael Steiner. Mitigating dictionary attacks on password-protected local storage. In *CRYPTO 2006*, pages 160–179. Springer, 2006.

[26] Siu Man Chan. Just a pebble game. In *IEEE Conference on Computational Complexity*, pages 133–143. IEEE, 2013.

[27] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-damgård revisited: How to construct a hash function. In *CRYPTO*, pages 430–448, 2005.

[28] 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.

[29] Ivan Bjerre Damgård. A design principle for hash functions. In *CRYPTO*, pages 416–427, 1989.

[30] Solar Designer. scrypt time-memory tradeoff, July 2011.

[31] Giovanni Di Crescenzo, Richard Lipton, and Shabsi Walfish. Perfectly secure password protocols in the bounded retrieval model. In *Theory of Cryptography*, pages 225–244. Springer, 2006.

[32] Jacob Donnelly. Bitmain announces launch of next-generation Antminer S7 Bitcoin miner. https://bitcoinmagazine.com/articles/bitmain-announces-launch-next-generation-antminer-s7-bitcoin-miner-1440958100, August 2015. Accessed 9 November 2015.

[33] Markus Dürmuth. Useful password hashing: how to waste computing cycles with style. In *New Security Paradigms Workshop*, pages 31–40. ACM, 2013.

[34] Cynthia Dwork, Andrew Goldberg, and Moni Naor. On memory-bound functions for fighting spam. In *CRYPTO*, pages 426–444. Springer, 2003.

[35] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *CRYPTO 1992*, pages 139–147. Springer, 1993.

[36] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In *CRYPTO*, pages 37–54, 2005.

[37] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *CRYPTO*, 2015.

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

[39] Serge Egelman, Andreas Sotirakopoulos, Ildar Muslukhov, Konstantin Beznosov, and Cormac Herley. Does my password go up to eleven?: the impact of password meters on password selection. In *CHI*, pages 2379–2388. ACM, 2013.

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

[41] David C. Feldmeier and Philip R. Karn. UNIX password security–ten years later. In *CRYPTO 1989*, pages 44–63. Springer, 1990.

[42] Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password-scrambling framework. Cryptology ePrint Archive, Report 2013/525, 2013. http://eprint.iacr.org.

[43] I. Gohberg and V. Olshevsky. Complexity of multiplication with vectors for structured matrices. *Linear Algebra and Its Applications*, 202:163–192, 1994.

[44] Oded Goldreich. Candidate one-way functions based on expander graphs. In *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*, pages 76–87. Springer, 2011.

[45] Shafi Goldwasser and Yael Tauman Kalai. On the (in)security of the Fiat-Shamir paradigm. In *FOCS*, pages 102–113. IEEE, 2003.

[46] Martin E. Hellman. A cryptanalytic time-memory trade-off. *Information Theory, Transactions on*, 26(4):401–406, 1980.

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

[48] John Hopcroft, Wolfgang Paul, and Leslie Valiant. On time versus space. *Journal of the ACM (JACM)*, 24(2):332–337, 1977.

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

[50] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner. Secure applications of low-entropy keys. In *Information Security*, pages 121–134. Springer, 1998.

[51] Jeremy Kirk. Internet address overseer ICANN resets passwords after website breach. http://www.pcworld.com/article/2960592/security/icann-resets-passwords-after-website-breach.html, August 2015. Accessed 9 November 2015.

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

[53] Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. Of passwords and people: measuring the effect of password-composition policies. In *CHI*, pages 2595–2604. ACM, 2011.

[54] Laura Krantz. Harvard says data breach occurred in June. http://www.bostonglobe.com/metro/2015/07/01/harvard-announces-data-breach/pqzk9IPWLMiCKBl3IijMUJ/story.html, July 2015. Accessed 9 November 2015.

[55] Jérôme Lacan and Jérôme Fimes. Systematic MDS erasure codes based on Vandermonde matrices. *IEEE Communications Letters*, 8(9):570–572, 2004.

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

[57] Philip Leong and Chris Tham. UNIX password encryption considered insecure. In *USENIX Winter*, pages 269–280, 1991.

[58] Sergio Demian Lerner. Strict memory hard hashing functions. https://bitslog.files.wordpress.com/2013/12/memohash-v0-3.pdf, January 2014. Accessed 9 November 2015.

[59] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, 1997.

[60] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The Theory of Error-Correcting Codes*. North-Holland Publishing Company, 1981.

[61] Katja Malvoni, Solar Designer, and Josip Knezovic. Are your passwords safe: Energy-efficient bcrypt cracking with low-cost parallel hardware. In *USENIX Workshop on Offensive Technologies*, 2014.

[62] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC press, 1996.

[63] Ralph C. Merkle. One way hash functions and DES. In *CRYPTO*, pages 428–446, 1989.

[64] Robert Morris and Ken Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, 1979.

[65] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *CRYPTO 2002*, pages 111–126. Springer, 2002.

[66] Jakob Nordström. New wine into old wineskins: A survey of some pebbling classics with supplemental results. http://www.csc.kth.se/~jakobn/research/PebblingSurveyTMP.pdf, March 2015. Accessed 9 November 2015.

[67] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *CRYPTO*, pages 617–630. Springer, 2003.

[68] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA 2006*, pages 1–20. Springer, 2006.

[69] Sunoo Park, Krzysztof Pietrzak, Joël Alwen, Georg Fuchsbauer, and Peter Gazi. Spacecoin: a cryptocurrency based on proofs of space. Technical report, Cryptology ePrint Archive, Report 2015/528, 2015.

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

[71] Wolfgang J. Paul and Robert Endre Tarjan. Time-space trade-offs in a pebble game. *Acta Informatica*, 10(2):111–115, 1978.

[72] Wolfgang J. Paul, Robert Endre Tarjan, and James R. Celoni. Space bounds for a game on graphs. *Mathematical Systems Theory*, 10(1):239–251, 1976.

[73] Colin Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCan*, May 2009.

[74] Alexander Peslyak. yescrypt. https://password-hashing.net/submissions/specs/yescrypt-v2.pdf, October 2015. Accessed 13 November 2015.

[75] Andrea Peterson. 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/, October 2015. Accessed 9 November 2015.

[76] Mark S. Pinsker. On the complexity of a concentrator. In *7th International Teletraffic Conference*, 1973.

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

[78] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *Journal of the ACM*, 26(2):361–381, 1979.

[79] Thomas Pornin. The Makwa password hashing function. http://www.bolet.org/makwa/, April 2015. Accessed 13 November 2015.

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

[81] Niels Provos and David Mazières. A future-adaptable password scheme. In *USENIX Annual Technical Conference*, pages 81–91, 1999.

[82] Arnold Reinhold. HEKS: A family of key stretching algorithms (draft g). http://world.std.com/~reinhold/HEKSproposal.html, July 2001. Accessed 13 November 2015.

[83] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, pages 199–212. ACM, 2009.

[84] John E. Savage. *Models of computation: Exploring the Power of Computing*. Addison-Wesley, 1998.

[85] Stuart Schechter, Cormac Herley, and Michael Mitzenmacher. Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks. In *HotSec*, pages 1–8. USENIX Association, 2010.

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

[87] Mohammad Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. A remark on matrix rigidity. *Information Processing Letters*, 64(6):283–285, 1997.

[88] Adam Smith and Ye Zhang. Near-linear time, leakage-resilient key evolution schemes from expander graphs. IACR Cryptology ePrint Archive, Report 2013/864, 2013.

[89] Martijn Sprengers and Lejla Batina. Speeding up GPU-based password cracking. In *SHARCS Workshop*, 2012.

[90] Rudy Takala. UVA site back online after chinese hack. http://www.washingtonexaminer.com/uva-site-back-online-after-chinese-hack/article/2570383, August 2015. Accessed 9 November 2015.

[91] Martin Tompa. Time-space tradeoffs for computing functions, using connectivity properties of their circuits. In *STOC*, pages 196–204. ACM, 1978.

[92] Abigail Tracy. 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/, October 2015. Accessed 9 November 2015.

[93] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[94] John Tromp. Cuckoo Cycle: a memory-hard proof-of-work system. *IACR Cryptology ePrint Archive*, 2014:59, 2014.

[95] Leslie G Valiant. On non-linear lower bounds in computational complexity. In *STOC*, pages 45–53. ACM, 1975.

[96] Leslie G. Valiant. Graph-theoretic arguments in low-level complexity. In Jozef Gruska, editor, *Mathematical Foundations of Computer Science*, volume 53 of *Lecture Notes in Computer Science*, pages 162–176. Springer Berlin Heidelberg, 1977.

[97] Steven J. Vaughan-Nichols. Password site LastPass warns of data breach. http://www.zdnet.com/article/lastpass-password-security-site-hacked/, June 2015. Accessed 9 November 2015.

[98] David Wagner and Ian Goldberg. Proofs of security for the Unix password hashing algorithm. In *ASIACRYPT 2000*, pages 560–572. Springer, 2000.

[99] Yaacov Yesha. Time-space tradeoffs for matrix multiplication and the discrete Fourier transform on any general sequential random-access computer. *J. Computer and System Sciences*, 29(2):183–197, 1984.

A Analysis of Argon2

In this section, we demonstrate that, after a linear-time pre-computation step, it is possible to compute the single-pass variant of the Argon2i password hashing function [15], the function that won a recent password hashing competition [70], using between a quarter and a fifth of the desired space with no computational penalty.

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.

Our findings apparently contrast with those of the Argon design documents, which seem to claim that computing N-block single-pass Argon2i with $N/4$ space incurs a $7.3\times$ computational penalty [15, Table

2]. Increasing the number of passes over the memory ameliorates this weakness but does not eliminate it: we show that computing the multiple-pass variant of Argon2i requires only roughly $N/e < N/2.71$ blocks of storage in expectation.

We analyze a idealized version the Argon2i algorithm, which is slightly simpler than that proposed in the Argon2 v1.2.1 specification [15]. 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 15 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.

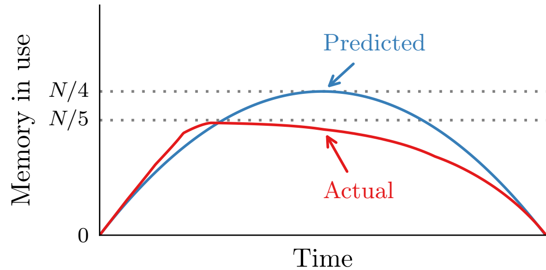


Figure 15: Space used by our algorithm for computing single-pass Argon2i during a single hash computation.

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 $i \in \{2, \ldots, N-1\}$, the block at index i is derived from two blocks: the block at index $(i-1)$ and a block selected pseudorandomly from the set of blocks generated so far. If we denote the contents of block i as x_i, then Argon2i operates as follows:

$$x_0 = H(\texttt{passwd},\ \texttt{salt} \parallel 0)$$
$$x_1 = H(\texttt{passwd},\ \texttt{salt} \parallel 1)$$
$$x_i = H(x_{i-1}, x_{r_i}) \quad \text{where } r_i \in \{0, \ldots, i-1\}$$

Here, H is a cryptographic compression function mapping two blocks into one block. The random index r_i is sampled from a non-uniform distribution over $S_i = \{0, \ldots, i-1\}$ 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 $(x_0, \ldots, x_{N-1})$ in sequence and outputs bytes derived from the last block x_{N-1}. 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 (e.g., the default is 3). 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-1}. 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.

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.

Small-Space Strategy. 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 $\{t_0, \ldots, t_{(pN-1)}\}$ in a read-only array. The total size of this table on a 64-bit machine is $8pN$ bytes.[3]

[3] 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.

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).

- **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.

Analysis of One-Pass Argon2i. We now analyze the space consumption of our algorithm. We are interested in the algorithm's expected space usage at time step t—call this function $S(t)$.[4] At each step of the 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* [...] *and* is not used at time N. Let $U_{i,t}$ denote the event that block i is *unused* at time t. Then:

$$\Pr\left[A_{i,t}\right] = \Pr\left[\bigcap_{t'=t+1}^{N} U_{i,t'}\right] = \prod_{t'=t+1}^{N} \Pr[U_{i,t'}] \qquad (9)$$

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 9, we get:

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

[4] As described in the prior section, the contents of block i are derived from the contents of block $i-1$ and a block chosen at random from the set $r_i \xleftarrow{R} \{1, \ldots, i-1\}$. 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. 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.

Let $B_{i,t}$ denote the event that block i in the storage buffer 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 block in the buffer can be accessed at time $t+1$ but by time $t+2$, the first block will have been deleted from the buffer. Similarly, the second block in the buffer at time t can be accessed at time $t+1$ or $t+2$, but not $t+3$ (since by then it will have been deleted from the buffer). Similarly, 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 Proof of Theorem 27

We prove Theorem 27 in a sequence of lemmata.

Lemma 37. *Let $V \subseteq \mathbb{F}^n$ be a linear space of dimension d. For $w \leq n$, there are at most $\binom{d+w-1}{w} \cdot (|\mathbb{F}|-1)$ vectors of exact weight w in V.*

Proof. Let $S_1 \subseteq V$ be a set of distinct vectors of exact weight w. To prove the lemma, we put an upper bound on the size of S_1 by constructing a sequence of sets: $S_1, S_2, \ldots, S_v$.

To construct S_{i+1} from S_i, arrange the vectors in S_i in an $(|S_i| \times n)$ matrix M_i. Let c_i be the number of non-zero columns in this matrix. There are $w|S_i|$ non-zero cells in M_i and therefore there must exist a non-zero column in M_i with at most $|S_i|(w/c_i)$ non-zero cells. Let k_i be the index of that column. Remove all the vectors from S_i that are non-zero in component k_i and let S_{i+1} be the remaining vectors.

First, we know that

$$|S_{i+1}| \geq |S_i| - |S_i| \cdot \frac{w}{c_i} = |S_i| \cdot \frac{c_i - w}{c_i} \ . \qquad (10)$$

Second, we know that $|S_{i+1}| < |S_i|$, since we have removed at least one vector from S_i. Finally, we know that $\mathrm{rank}(S_{i+1}) < \mathrm{rank}(S_i)$, since S_i contains at least one vector that is outside of the span of the set S_{i+1} (i.e., a vector that is non-zero in coordinate k_i).

Since the rank of each set we construct is strictly less than the rank of the previous set, after constructing at most $d-1$ such sets, we reach a set S_v (for some $v \leq d$) such that the vectors of S_v span a space of dimension one. Then $|S_v| \leq |\mathbb{F}| - 1$. Moreover, from Equation (10) we know that

$$|S_1| \leq \frac{c_1}{c_1 - w} \cdot |S_2| \leq \cdots \leq \frac{c_1}{c_1 - w} \cdots \frac{c_{v-1}}{c_{v-1} - w} \cdot |S_v|$$
$$\leq \frac{c_1}{c_1 - w} \cdots \frac{c_{v-1}}{c_{v-1} - w} \cdot (|\mathbb{F}| - 1) \qquad (11)$$

where $c_1 > c_2 > \cdots > c_{v-1} > w$. The quantity in (11) is maximized when

$$c_{v-1} = w+1, \quad c_{v-2} = w+2, \quad \ldots, \quad c_1 = w+v-1$$

and therefore

$$|S_1| \leq \binom{w+v-1}{v-1} \cdot (|\mathbb{F}|-1) = \binom{w+v-1}{w} \cdot (|\mathbb{F}|-1)$$

as required. □

As an immediate corollary, we obtain a bound on the number of vectors of weight w in an affine space.

Corollary 38. *Let $V \subseteq \mathbb{F}^n$ be a linear space of dimension d. For a vector $\mathbf{v} \in V$, the affine space $V + \mathbf{v}$ contains at most $\binom{d+w}{w}(|F|-1)$ vectors of exact weight w.*

Proof. Let $S \subseteq V + \mathbf{v}$ be a set of vectors of exact weight w. Then $S - \mathbf{v} \subseteq V$ and therefore $(S - \mathbf{v}) \cup \{\mathbf{v}\}$ spans a space W of dimension at most $d+1$. Since S is contained in W, and W has dimension at most $d+1$, it follows by Lemma 37 that $|S| \leq \binom{d+w}{w}(|F|-1)$ as required. □

For the next lemma it is convenient to introduce the following notation: For a set $S \subseteq \{1, \ldots, n\}$ and a vector $\mathbf{v} \in \mathbb{F}^n$, let $\mathbf{v}_S$ denote the vector $\mathbf{v}$ after removing all the columns whose index is not in S, so that $\mathbf{v}_S \in \mathbb{F}^{|S|}$. For example, $(1,0,1,0,1)_{\{1,3,4\}} = (1,1,0)$.

Recall that $\mathcal{P}_{n,w}$ denotes the distribution that samples a vector $v \in \mathbb{F}_2^n$ by choosing w random indices, with replacement, in $\{1, \ldots, n\}$ and setting these components of v to 1 (v is zero elsewhere).

Lemma 39. *Let n, w, ρ be as in Theorem 27. Let $S \subseteq \{1, \ldots, n\}$ be a set of size $n/2$. Let $\mathbf{v} \in \mathbb{F}_2^n$ be a vector sampled from $\mathcal{P}_{n,w}$ and let $V \subseteq \mathbb{F}_2^{n/2}$ be some linear space of dimension at most n/ρ. Then*

$$\Pr\left[\mathbf{v}_S \in V\right] < \left(\frac{1}{2} + \frac{1.1}{\rho}\right)^w$$

Proof. Let $\mathbf{z} \in \mathbb{F}_2^n$ be some fixed vector such that $\mathbf{z}_S$ has weight $k \leq w$. The probability that $\mathbf{v}_S$ is equal to $\mathbf{z}_S$ is at most

$$\frac{k!}{n^k}\binom{w}{k} \cdot \left(\frac{1}{2} + \frac{k}{n}\right)^{w-k} \qquad (12)$$

To see why, we choose k of the w non-zero positions in $\mathbf{v}_S$ to match the non-zero positions in $\mathbf{z}_S$ and this

matching can be done $k!$ ways. The remaining $w-k$ non-zero positions of v_S must either fall outside the set S or on one of the k non-zero positions in $\mathbf{z}_S$.

Let $r = n/\rho$. By Lemma 37 the number of vectors of weight k in V is bounded by

$$\binom{r+k-1}{k} \le \frac{(r+k)^k}{k!} \le \frac{(r+w)^k}{k!} \qquad (13)$$

Combining (12) and (13) we obtain:

$$\begin{aligned}\Pr\left[\mathbf{v}_S \in V\right] &\le \sum_{k=0}^{w} \binom{w}{k}\left(\frac{r+w}{n}\right)^k \cdot \left(\frac{1}{2}+\frac{w}{n}\right)^{w-k} \\ &= \left(\frac{1}{2}+\frac{r+2w}{n}\right)^w\end{aligned}$$

where the last equality is an application of the binomial theorem. Since $r = n/\rho$ and by the assumption on n imposed in the statement of the theorem ($n > 20w\rho$) we have:

$$(r+2w)/n < 1.1/\rho,$$

so that

$$\Pr\left[\mathbf{v}_S \in V\right] < \left(\frac{1}{2}+\frac{1.1}{\rho}\right)^w$$

as required. □

Lemma 40. *With the notation of Theorem 27, let A be an $(n/4)\times n$ matrix obtained by selecting some $n/4$ rows of M. Then A is $(1,2,\rho)$ inflexible with probability at least $1-2^{-0.82n}$.*

Proof. Fix some set of $n/2$ columns of A and call the resulting $n/4 \times n/2$ matrix $A_{1/2}$. We first argue that $\mathrm{rank}(A_{1/2}) > n/\rho$ with overwhelming probability.

For convenience, let $m = n/4$ and $r = n/\rho$. For all $i = 1,\dots,m$, let I_i be the indicator variable that is one if the ith row of $A_{1/2}$ is outside of the span of the top $i-1$ rows of $A_{1/2}$, and is zero otherwise. We show that

$$\begin{aligned}\Pr\left[\mathrm{rank}(A_{1/2}) > r\right] &= \Pr\left[\sum_{i=1}^{m} I_i > r\right] \\ &> 1-2^{-1.82n} \qquad (14)\end{aligned}$$

or equivalently, that

$$\Pr\left[\sum_{i=1}^{m} I_i \le r\right] < 2^{-1.82n}. \qquad (15)$$

Once we prove that (14) holds, we can apply the union bound over all $\binom{n}{n/2}$ choices for $A_{1/2}$ to prove the lemma:

$$\begin{aligned}\Pr\left[\begin{array}{c} A \text{ is } (1,2,\rho) \\ \text{inflexible}\end{array}\right] &> 1-\binom{n}{n/2}\cdot 2^{-1.82n} \\ &> 1-2^n\cdot 2^{-1.82n} = 1-2^{-0.82n}.\end{aligned}$$

It remains to prove (15). Let $A_{1/2}^{(i)}$ be the $i \times n/2$ matrix obtained from the top i rows of $A_{1/2}$. By Lemma 39 we have that

$$\Pr\left[I_{i+1} = 0 \;\middle|\; \mathrm{rank}(A_{1/2}^{(i)}) \le n/\rho\right] < \left(\frac{1}{2}+\frac{1.1}{\rho}\right)^w$$

To complete the proof of (15) first observe that the event $I_1 + \ldots + I_m \le r$ is the same as the event that *some* set of $m-r$ indicator variables is 0. We first bound the probability that a specific set of $m-r$ indicator variables is 0. Without loss of generality we can focus on the last $m-r$ variables $I_r,\dots,I_m$ since the bottom rows of $A_{1/2}$ are the ones most likely to be linearly dependent on the previous rows. Since the first $r-1$ rows of $A_{1/2}$ have rank at most $r-1$ we have:

$$\begin{aligned}\Pr\left[\bigcap_{i=r}^{m}(I_i = 0)\right] &= \Pr[I_r = 0]\cdot\Pr[I_{r+1} = 0 \mid (I_r = 0)]\cdots \\ &\cdots\Pr[I_m = 0 \mid (I_r = 0)\wedge\cdots\wedge(I_{m-1} = 0)] \\ &< \left(\frac{1}{2}+\frac{1.1}{\rho}\right)^{w(m-r)} = \left(\frac{1}{2}+\frac{1.1}{\rho}\right)^{w(1/4-1/\rho)n}\end{aligned}$$

We now prove (15) by a union bound over all sets of $m-r$ indicator variables:

$$\Pr\left[\sum_{i=1}^{m} I_i \le r\right] < \binom{m}{m-r}\cdot\left(\frac{1}{2}+\frac{1.1}{\rho}\right)^{w(1/4-1/\rho)n}$$

Using the inequality $\binom{a}{b} < (\frac{a\cdot e}{b})^b$ we obtain

$$\binom{m}{m-r} = \binom{m}{r} = \binom{n/4}{n/\rho} < \frac{e\cdot\rho}{4}^{n/\rho} < \rho^{n/\rho}$$

Therefore,

$$\begin{aligned}\Pr\left[\sum_{i=1}^{m} I_i \le r\right] &< \left[\rho^{1/\rho}\left(\frac{1}{2}+\frac{1.1}{\rho}\right)^{w(1/4-1/\rho)}\right]^n \\ &< 2^{-1.82n}\end{aligned}$$

where the last inequality follows from the requirements on w and ρ in the statement of Theorem 27. This proves (15) and completes the proof of Lemma 40. □

Theorem 27 now follows directly from Lemma 40.

Proof of Theorem 27. The proof is by a union bound over the n-choose-$n/4$ possible choices for the $n/4$ rows of M. We know that

$$\binom{n}{n/4} < 2^{H_b(1/4)n} < 2^{0.812n},$$

where $H_b(p) = -p \log_2 p - (1-p)\log_2(1-p)$ is the binary entropy function. We have applied the fact that $H_b(1/4) < 0.812$.

Using Lemma 40, the matrix M in Theorem 27 is $(4, 2, \rho)$ inflexible with probability at least

$$\begin{aligned} 1 - \binom{n}{n/4} \cdot 2^{-0.82n} &> 1 - 2^{0.812n} \times 2^{-0.82n} \\ &= 1 - 2^{-0.008n} > 1 - 1.0055^{-n} \end{aligned}$$

as required. □