Defending Hash Tables from Algorithmic Complexity Attacks with Resource Burning

Trisha Chakraborty[a], Jared Saia[b], Maxwell Young[c]

[a] *Mississippi State University, Department of Computer Science and Engineering, Mississippi State, 39762, MS, USA, tc2006@msstate.edu*
[b] *University of New Mexico, Department of Computer Science, Albuquerque, 87131, NM, USA, saia@cs.unm.edu*
[c] *Mississippi State University, Department of Computer Science and Engineering, Mississippi State, 39762, MS, USA, myoung@cse.msstate.edu*

Abstract

We consider the problem of defending a hash table against a Byzantine attacker that is trying to degrade the performance of query, insertion and deletion operations. Our defense makes use of resource burning (RB)—the verifiable expenditure of network resources—where the issuer of a request incurs some RB cost. Our algorithm, Depth Charge, charges RB costs for operations based on the depth of the appropriate object in the list that the object hashes to in the table. By appropriately setting the RB costs, our algorithm mitigates the impact of an attacker on the hash table's performance. In particular, in the presence of a significant attack, our algorithm incurs a cost which is asymptotically less that the attacker's cost.

Keywords: algorithmic complexity attack, hash table, resource burning

1. Introduction

While hash tables are a popular data structure, their performance can be significantly degraded if the objects to be stored are chosen adversarially (e.g., Crosby and Wallach (2003), Bar-Yosef and Wool (2007), and Tobin and Malone (2012)). In an extreme case, all objects can be hashed to the same

A preliminary version of this work appeared in the proceedings of the *25th International Conference on Distributed Computing and Networking (ICDCN)* (see Chakraborty et al. (2024)).

index of the table. Under the common collision-resolution method of chaining, this attack effectively transforms the hash table into a linked list, which leads to a worst-case query time that is linear in the number of objects; this is an example of an ***algorithmic complexity attack (ACA)*** (see Bar-Yosef and Wool (2007), Crosby and Wallach (2003), Cai et al. (2009), Sun et al. (2011), and Khan and Traore (2005)). Many data structures are vulnerable to ACAs, and designing a defense is challenging, since malicious inputs need not be large, or arrive at a high rate, in order to degrade performance. In other words, *ACAs are often less costly to launch than they are to defend against.*

In hash tables, a common defensive measure is to keep the hash function secret (known only to the server) and use stronger (cryptographic) hash functions that are difficult to invert. However, side-channel attacks may allow an adversary to learn the hash function (see Olekšák and Miškovský (2022)), and in distributed settings where the hash table may be stored on multiple machines, a single compromised machine may reveal the secret. Similarly, stronger hash functions offer insufficient protection, as an adversary can find objects that hash to the same index through trial and error. These vulnerabilities are discussed in Section 1.5, which highlight a fundamental shortcoming of prior defenses: they do not counteract the cost advantage enjoyed by the attacker.

In this work, we design and analyze a new defense for hash tables that employs ***resource burning (RB)***—the verifiable expenditure of a network resource—to reverse this cost asymmetry. Specifically, any user wishing to access the hash table must pay an RB cost. By setting the amount of RB appropriately, our defense guarantees that the cost to legitimate users grows slowly as a function of an attacker's cost for launching an ACA. In practice, attackers must often pay for the resources needed to launch attacks, such as renting compromised machines (see Franklin et al. (2007)). Therefore, the asymptotic advantage given by our approach ultimately translates into a financial edge for the defenders.

1.1. Motivating Our Setting

We consider the challenging setting where both (1) the number of indices in the hash table; and (2) the hash function are fixed. Our "fixed" setting is particularly relevant for many applications in distributed computing. For example, in the client-server setting, changing the hash table size or the hash function can result in down time that negatively impacts quality of service for the clients. Thus, system administrators often analyze workload data to set the hash table size appropriately; a discussion of this is provided in IBM (2023). In peer-to-peer systems such as distributed hash tables (DHTs) (e.g., Wang and Kangasharju (2013), Falkner et al. (2007), and Stoica et al. (2001)), memory and disk space is bound by the number of participating machines, and thus resizing is not possible.

Our approach allows for flexibility in setting the appropriate table size. Specifically, we parameterize our results by $\boldsymbol{\ell_M}$, which is the maximum number of legitimate objects (or "good objects", as defined below) that are hashed to the same index. Intuitively, this parameter is small when the table is appropriately sized, and we discuss this further in Section 1.3.

1.2. Model

In our setting, there are clients, an adversary, and a server. Note that all clients are "good"; we do not refer to "bad" clients, since the adversary incarnates them. Our server may represent multiple real-world servers. We now describe the key aspects of our model.

Hash Table. The server holds a hash table that services insertions, queries, and deletions of objects by request from the clients and the adversary. An insertion by a client is a ***good insertion***, and the corresponding object is a ***good object***, which is placed at an index selected uniformly at random (u.a.r.).[1] Otherwise, the insertion is a ***bad insertion*** and the corresponding object is a ***bad object***; in this case, the adversary selects the index where the object is inserted. Good insertions cannot be distinguished from bad insertions, and good objects cannot be distinguished from bad objects.

A collision occurs when two or more objects are inserted at the same index of the hash table, and this is resolved via the popular method of chaining. That is, the objects involved in the collision form a ***list***, where the head of the list (***HoL***) is located in the index of the hash table, with subsequent objects added to the tail of the list (***ToL***) in the order that they are inserted. The ***length*** of a list at index i is the number of objects stored at index i. The ***depth*** of an object is the position measured from the head of its list;

[1] We view this as performing a hash function evaluation on an identifier for the object. The first evaluation returns an index in the hash table selected uniformly at random, while subsequent evaluations of this identifier always map to the same index (i.e., our hash function obeys the random oracle assumption by Koblitz and Menezes (2015)).

the minimum depth is 1. If a list exists at the index of insertion, then an object inserted at that index is added to the ToL.[2]

In addition to insertion, the hash table also handles query and delete requests. A query (deletion) from a client is said to be a ***good query*** (***good deletion***); otherwise, it is a ***bad query*** (***bad deletion***). Good queries (deletions) cannot be distinguished from bad queries (deletions). Clients only issue queries for good objects, while the adversary may issue queries for any object; this captures a pessimistic setting where the objects inserted by the adversary are not useful and only serve to degrade the performance of the algorithm. Clients may delete good objects, and the adversary may delete bad objects but not good objects.

Resource Burning. Upon receiving a request to insert or query an object the server may issue a ***resource-burning (RB) challenge*** to the requester (i.e., a client or the adversary). The requester must return a solution to the RB challenge before the corresponding request is satisfied. To specify the RB cost x, for any positive integer x, we will refer to an ***x-hard*** RB challenge.

The mechanisms for issuing and verifying RB challenges can be protected from attack themselves, given their narrow functionality (see Waters et al. (2004)). Furthermore, significant work has gone into addressing the many practical details of designing and deploying RB challenges, such as handling device heterogeneity, pre-computation attacks, and the reuse of old solutions (see Ali et al. (2020), Li et al. (2012), and Walfish et al. (2010)).

Performance Metrics. We use two metrics for gauging performance: (1) the ***RB cost*** of solving RB challenges and (2) the ***latency*** for servicing requests. Regarding (1), the ***algorithm's RB cost*** is the sum of the hardness values for all RB challenges solved by the clients; likewise, the ***adversary's RB cost*** is the sum of the hardness values for RB challenges it solves.

Regarding (2), if the request is an insertion, then the latency equals 1, since we assume that each list maintains a pointer to the ToL. If the request is a query and the object exists in that list, then the latency equals the object's depth in that list; otherwise, the latency equals the list length at the index where the object would have been stored if it existed in the table. We pessimistically assume that only the algorithm, and not the adversary, incurs a latency cost.

[2] A design where objects are inserted at the HoL are also vulnerable to ACAs and would result in essentially the same analysis.

Communication. Requests received by the server are ordered in time, and we pessimistically assume this ordering is set by the adversary. Point-to-point communication can occur between the clients and the server, and between the adversary and the server. This communication occurs instantaneously; however, in Section 4.1 we discuss handling communication delay.

Adversary. We consider a ***Byzantine adversary*** that is not constrained to obey protocol and is not computationally bounded. The adversary has full knowledge of the hash table's configuration, as well as the state of the clients and server. The adversary can instantaneously create as many bad objects as it likes that hash to any targeted index of the hash table. In other words, the number of bad objects and the indices into which they are inserted is chosen by the adversary.

In contrast, the adversary has no control over where good objects are inserted; each good object is inserted into an index chosen u.a.r. from the set of all indices. While the adversary does not control where good objects are inserted, it may control which good objects are queried (i.e., generate or schedule queries for good objects). We consider both the case where (1) the good queries are generated by the adversary (Section 3.2); and (2) the good queries are distributed u.a.r. over all indices (Section 3.3).

1.3. Main Results

For requests, we let $\boldsymbol{\mathcal{I}}$ be the number of good insertions; let $\boldsymbol{\mathcal{Q}}$ and $\boldsymbol{\mathcal{D}}$ be the number of good queries and good deletions for objects that exist in the hash table.

In discussing the hash table, we denote the number of indices in the hash table by $\boldsymbol{t}$; the maximum number of good objects in any index by $\boldsymbol{\ell_M}$. The number of good objects in index i is $\boldsymbol{\ell_i}$, and the average number of good objects, $(1/t)\sum_{i=1}^{t}\ell_i$, is denoted by $\boldsymbol{\ell}_{\text{ave}}$.

Throughput, we use $\boldsymbol{\mathcal{B}}$ to denote the total RB cost incurred by the adversary. We state our main result below regarding our defense algorithm, Depth Charge.

Theorem 1. Depth Charge *guarantees the following properties:*

1. *(Single Requests) Any single insertion has RB cost $O(\sqrt{\mathcal{B}} + \ell_M)$ and latency $O(1)$. Any single query or deletion has an RB cost and latency that are each $O(\sqrt{\mathcal{B}} + \ell_M)$.*

2. *(Amortized Requests) When $\mathcal{I}$, $\mathcal{Q}$, and $\mathcal{D}$ are set by an Byzantine adversary, the total RB cost and the total latency are each $O((\mathcal{I}+\mathcal{Q}+\mathcal{D}+\sqrt{(\mathcal{I}+\mathcal{Q}+\mathcal{D})\mathcal{B}})\ell_M^2)$.*

3. *(Randomly Queried Indices) Consider Q queries where the corresponding objects belong to indices of the hash table chosen u.a.r.. For $Q \geq \ell_M^2\mathcal{B}$, the average cost per query is $O(\ell_{ave})$ in expectation.*

Discussion. As mentioned earlier, our results are parameterized by ℓ_M. For $\mathcal{I}$ insertions into a table of size t, $\ell_M = O(\lceil \mathcal{I}/t \rceil \log t)$ with high probability in t (***w.h.p.***).[3] Notably, for $\mathcal{I} = O(t)$, it is well known that $\ell_M = O(\log t / \log\log t)$ (see Oliveira (2021), Kesselheim (2016), and Raab and Steger (1998)). This case is pertinent, since many applications limit the amount by which their hash table can grow (see Crosby and Wallach (2003)), and the number of size increases may be very limited (e.g. see Czubak and Szymanek (2017)).[4] From a theory perspective, such limited growth increases the table size by a constant factor, and using the the largest size aligns with our model.

To provide context for Theorem 1, it is helpful to compare Depth Charge to a standard hash table with chaining. In the latter, the adversary may create a list that has size linear in the number of bad objects for "free". This attack leads to poor latency if good objects reside at the ToL. By comparison, Property 1 bounds improves (roughly) quadratically by bounding the longest list length to be $O(\sqrt{\mathcal{B}}+\ell_M)$; clearly, this holds for any query, even for objects that do not exist in the table. Another implication of Property 1 is that when there is little-to-no attack (i.e., when $\mathcal{B} \approx 0$), the RB cost and latency are each roughly $O(\ell_M)$, which should be small (i.e., logarithmic in t) for an appropriately sized table, as discussed above.

Regarding Property 2, for multiple requests scheduled by a Byzantine adversary, Depth Charge retains an asymptotic advantage when under significant attack. Conversely, when the attack is not large relative to $\mathcal{I}+\mathcal{Q}+\mathcal{D}$, Depth Charge has RB cost and latency proportional to this number of requests and ℓ_M^2. In contrast, in a standard hash table, the adversary can amplify its attack by forcing multiple requests involving a linear-sized chain.

Finally, Property 3 provides bounds on the expected performance under

[3] With probability at least $1-t^{-d}$ for some constant $d \geq 1$.

[4] For example, the table in the Cisco router examined in Czubak and Szymanek (2017) has an initial size of 1024, and can increase to sizes 2048, 4096, and 8192.

a sequence of queries that map to indices selected u.a.r. Specifically, the expected cost for Q such good queries is $O(Q\ell_{\text{ave}} + \ell_M\sqrt{Q\mathcal{B}})$. Thus, if that expectation holds, then the average cost per query is $O(\ell_{\text{ave}})$ when Q is large relative to $\mathcal{B}$ and ℓ_M. When $\ell_{\text{ave}} = O(1)$, this implies that the average cost per query is $O(1)$ in expectation. Interestingly, this is comparable to the expected $O(1)$ latency per query in a standard hash table.

1.4. Technical Overview

At a high level, our analysis relies on upper bounding Depth Charge's cost and lower bounding the adversary's cost. Below we sketch how to do this first for insertions, and then for queries and deletions.

Insertion Costs. We define a ***targeted*** index to be an index where there is at least one bad object and at least one good object. Then we lower bound the adversary's cost as a function of the number of bad objects inserted into targeted indices (Lemma 1).

Next, we upper bound the total cost of good insertions as a function of the number of objects in targeted indices, noting that this cost is maximized when the bad objects are distributed as uniformly as possible across such indices (Lemma 3). We pessimistically assume that good insertions come after bad insertions, since this minimizes the insertion cost to the adversary, while maximizing the insertion cost to the algorithm. Additionally, we assume that there are ℓ_M good insertions in every targeted index.

Why Use Move-to-Front? An analysis of the longest list (Lemma 5) shows that the worst-case latency per query is $O(\sqrt{\mathcal{B}}+\ell_M)$. While this significantly improves over the linear latency—for example, where the adversary inserts all objects in a single list—that can arise in undefended hash tables, there is still room for improvement. To see why, consider that even if the adversary ceases its attack, good objects will remain near the tails of their respective lists, leading to persistently poor query latency. By moving queried objects to the head of their respective lists, we can improve their latency in subsequent queries.

The classic move-to-front (MTF) heuristic proposed by Bentley and McGeoch (1985), Hester and Hirschberg (1985), and Rivest (1976) is known to improve performance in chained hash tables when they are not under attack (see Zobel et al. (2001), Askitis and Zobel (2011), and Song et al. (2017)). Our motivation for using MTF in our adversarial setting is that a substantial improvement may be attained over multiple queries, since good objects can "skip the line" in long lists that contain mostly bad objects.

However, the adversary can cause trouble for MTF in the following manner. When the adversary queries a bad object, it is moved to the front of the list. This increments the depth of a number of good objects as large as the depth of the bad object prior to being moved to the front; this increases the query latency of these good objects. We can discourage this bad behavior by charging for a query, but how much should we charge? Intuitively, a reasonable charge would be the depth of the queried object.

Analysis of Charging by Depth for Queries. To see why this is the correct charging scheme, consider a list composed of bad objects, except for a single good object o at the HoL. In order to increase the depth of o by d, the adversary must pay for d bad queries. Observe that each bad query must be for a bad object with larger depth than o; otherwise, querying the bad object does not increase o's depth. Under our charging scheme, the adversary pays at least $\sum_{j=1}^{d}(j+1) = \Theta(d^2)$. Then, when Depth Charge next queries o, it will pay an RB cost of $\Theta(d)$, and the query requires $\Theta(d)$ latency. Thus, Depth Charge obtains a quadratic advantage, similar to what is achieved for our bound on insertion costs.

This charging scheme motivates the name Depth Charge and it guarantees that the adversary must spend continually in order to keep good objects at large depth in the list.

The Amortized Analysis. A major technical challenge of our paper is to formalize the above intuition in the general case—with multiple lists, each with potentially multiple good objects. This analysis is challenging, since both bad and good queries can increase the depth of multiple good objects in a list. Over all lists, we need to track the depth of all good objects over a sequence of requests. We highlight that must account not only for queries—although they are what increases the depth of an object—but also insertions and deletions. Fortunately, insertions do not increase depth of other objects, given that objects are added to the ToL, so our bound on insertion cost (discussed above) can be used. As for deletions, we can treat them as queries, since they are no worse in terms of increasing depth.

One main analytic tool used is amortized analysis; in particular, the accounting method (see pg. 453 by Cormen et al. (2022)). Each good object is given a (conceptual) wallet into which Depth Charge makes deposits for each request that increases the depth of that object. The payments ensure a key invariant: the depth of a good object is never more than the number of dollars in its wallet. Therefore, an object's wallet always contains enough dollars to cover the cost of its next query. Over a sequence of requests, the total number of dollars deposited into all wallets is an upper bound on both Depth Charge's RB cost and latency.

How can we relate the number of dollars deposited into wallets to the adversary's cost? This is addressed formally in Lemma 6; however, to gain insight, let us extend our example to $q_i \geq 1$ good queries in a single list at index i. Prior to each good query, there are d_r bad queries that increase the depth of at most ℓ_M good objects by d_r, for $r = 1, ..., q_i$. The resulting number of dollars that Depth Charge places into the wallets of the corresponding ℓ_M good objects is $\mathcal{A}_i \leq \ell_M \sum_{r=1}^{q_i} d_r$, while the adversary's cost is $\mathcal{B}_i \geq \sum_{r=1}^{q_i} d_r^2 = \Omega((1/q_i)(\sum_{r=1}^{q_i} d_r)^2)$ by Jensen's inequality for concave functions. Therefore, the number of dollars deposited into wallets for objects in the list at index i is $\mathcal{A}_i = O(\ell_M \sqrt{q_i \mathcal{B}_i})$. To this, we add Depth Charge's cost for insertions, denoted by $\mathcal{A}_i^{\text{ins}}$, to get an upper bound on all requests involving this list.

Finally, in Lemma 7 and Corollary 3, we sum up the costs to Depth Charge over all lists. Our previous bound on the insertion costs handles the sum of the $\mathcal{A}_i^{\text{ins}}$ terms. To simplify $O(\sum_i \ell_M \sqrt{q_i \mathcal{B}_i})$, we apply the Cauchy-Schwarz inequality to get an upper bound of $O(\ell_M \sqrt{\mathcal{Q}\mathcal{B}})$, where $\sum_i q_i = \mathcal{Q}$ is the total number of queries and $\sum \mathcal{B}_i \leq \mathcal{B}$, where $\mathcal{B}$ is *total* adversarial cost. Together, these bounds yield the expression in Property 2.

Randomly Queried Indices. Our result for randomly queried indices does not follow directly from Property 2. Instead, our argument (Lemma 8) leverages the bound for a single list (Lemma 6) in order to express the total cost from the randomly queried indices as a function of $E[Q_i]$ and $E[\sqrt{Q_i}]$. The latter is the more complicated term, which is handled by the application of Jensen's inequality for the expectation of concave functions, which shows that $E[\sqrt{Q_i}] \leq \sqrt{E[Q_i]}$. Using the fact that $E[Q_i] = Q/t$, and summing the terms over all lists, yields the expression in Property 3.

1.5. Related Work

In this section, we summarize work on RB-based defenses for a variety of attacks. Next, we discuss results from the literature on ACAs, with a focus on prior results for hash tables. We highlight that a preliminary version of this work appeared in the proceedings of the *25th International Conference on Distributed Computing and Networking* (Chakraborty et al. (2024)).

1.5.1. Defenses using Resource Burning

RB is a well-established tool for securing distributed systems; for example, this is discussed by Gupta et al. (2020). Many approaches based on RB consist of two primary components: a *prover* and a *verifier*; for example see Dwork and Naor (1992), Waters et al. (2004), and Aura et al. (2000). The prover offers verifiable evidence of completed work, while the verifier is responsible for confirming that the evidence is valid. For instance, when a client requests a service, the server responds by issuing an RB challenge. The client provides an RB solution, and if this solution is verified, the service is granted. The difficulty of the challenge can be adjusted, where a higher difficulty requires using more units of a chosen resource.

There is a substantial body of research spanning several decades that utilizes RB to address general security issues; we refer the interested reader to the surveys by Ali et al. (2020) and Gupta et al. (2020). RB has been applied in various domains such as in wireless networks by Gilbert and Zheng (2013), in peer-to-peer systems by Li et al. (2012) and Borisov (2006), and in blockchains (see the survey by Lin and Liao (2017)). Specifically, RB techniques have contributed to auditing metered websites (Franklin and Malkhi (1997)), making a digital data preservation protocol resistant to malicious peers (Neudecker (2017) and Nakamoto (2008)), limiting the incoming flow of service requests (Waters et al. (2004)), and combating electronic spam mails (Dwork and Naor (1993)).

The choice of resource burned in an RB system is an implementation detail and should be approached by finding a balance between security, fair participation, and integration feasibility in different network settings. Given this, our algorithm is deliberately agnostic about the resource burned, such as computational power (e.g., Wang and Reiter (2003)), bandwidth (e.g., Walfish et al. (2006)), computer memory (eg., Abadi et al. (2005), Dwork et al. (2003), and Dziembowski et al. (2015)), and human effort (e.g., Von Ahn et al. (2003) and Oikonomou and Mirkovic (2009)).

1.5.2. Prior Defenses for ACAs.

Many other common data structures and algorithms are vulnerable to ACAs, such as linked lists (see Atre et al. (2022)), quicksort (see Khan and Traore (2005) and McIlroy (1999)), cardinality sketches (see Reviriego and Ting (2020)), pattern matchers (see Kirrage et al. (2013) and Namjoshi and Narlikar (2010)), cuckoo filters (see Reviriego and Larrabeiti (2020)), and bloom filters (see Reviriego and Rottenstreich (2020)). As a result, AC attacks can impact common applications: networked applications (see Chang et al. (2009) and Atre et al. (2022)), firewalls (see Czubak and Szymanek (2017)), PDF compressors (see Hauke and Renardy (2019)), web services (see Altmeier et al. (2016)), and intrusion detection systems (see Crosby and Wallach (2003)).

In the context of hash tables, the prior literature on defending against ACAs falls into the three general categories discussed below.

Choice of Hash Functions. Crosby and Wallach (2003) showed the first ACA on hash tables, which caused a server to drop over 70% of queries. The authors proposed two techniques to mitigate ACAs: (a) adding a secret value as a parameter to the hash function, and (b) using universal hash functions (UHFs). The usage of UHFs can minimize the number of collisions, but UHFs can add computational overhead on the server side. Furthermore, Bar-Yosef and Wool (2007) demonstrated an ACA against hash tables despite the use of a secret value; the authors suggest that the secret-key length should be increased (beyond 32 bits) or be changed frequently. In a similar vein, Aumasson and Bernstein (2012) proposed SipHash which uses a secret key (known only to the server), which is used as input to the hash function. Unfortunately, a secret key may be compromised via side-channel attacks (see Olekšák and Miškovský (2022)) or, in distributed settings, by an adversary who controls one or more of the servers. Finally, perfect hashing is a technique that guarantees no collisions (see Lu et al. (2006), Cercone (1988), and Majewski et al. (1996)). However, constructing perfect hash functions is time consuming and requires knowing the set of objects to be hashed, which is not always available.

Application-Specific Defenses. Many defenses against ACAs are application-specific. For example, PHP limits the number of GET and POST HTTP requests so that the adversary cannot request to store many bad objects in a hash table (see Heimes (2013)). Another approach is the use of caching to store pre-computed results of expensive hash table lookups (see Zhang and Sanchez (2019), Metreveli et al. (2012), and Bender et al. (2012)).

Switching to Deterministic Data Structures. Another method for defending against ACAs is to adopt deterministic data structures with strong worst-case performance guarantees. For example, a deterministic skip list Munro et al. (1992) performs each insertion and each query with a worst-case bound that is logarithmic in the number of objects. However, this is inferior to the performance of a hash table, which have constant expected time per query in the absence of attack. If attacks are likely to occur over a minority of the system lifetime, then using a deterministic data structure is costly.

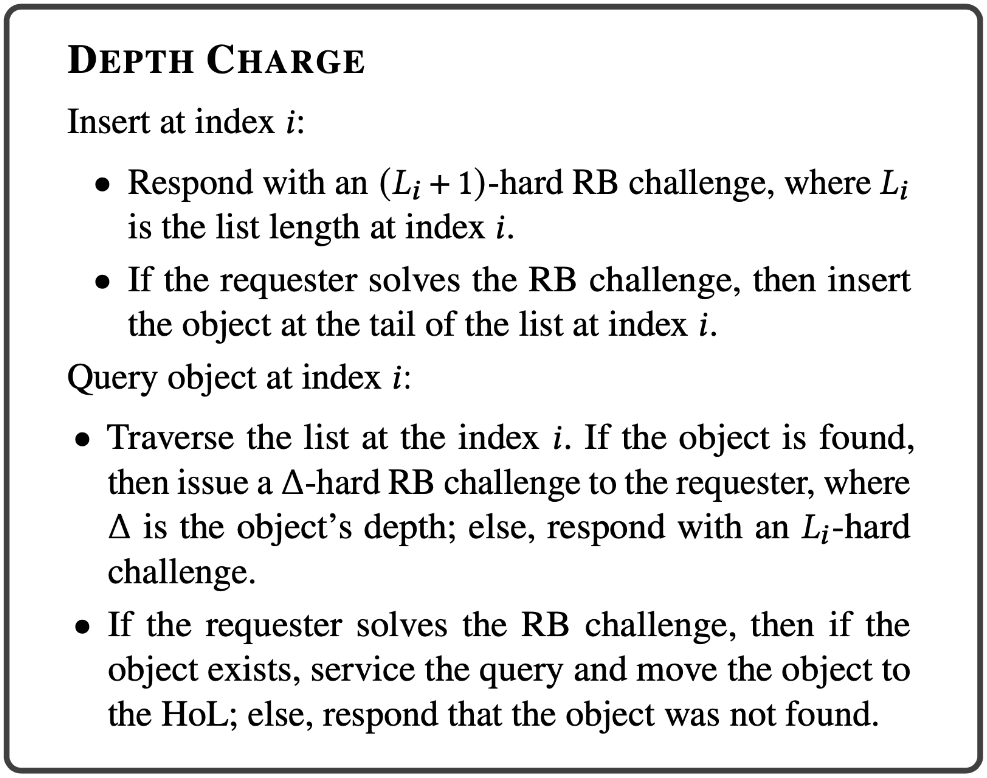
More generally, deterministic data structures incur theoretical and/or practical costs exceeding that of their randomized equivalents; for example, this shortcoming is acknowledged by Czubak and Szymanek (2017) in regards to B-trees, AVL-trees, and red-black trees, which offer worst-case logarithmic guarantees. Maintaining both a deterministic and randomized data structure might provide the advantages of both options, but such redundancy is likely to be expensive. In contrast, our algorithm's costs adapts to the degree of attack—in particular, growing slowly in the amount spent by the adversary—which allows for low cost when the attack is absent/small, and giving a favorable relative cost when the attack is large.

1.5.3. Resource Competitiveness

Algorithms that parameterize the algorithm's cost by the adversary's cost are called ***resource competitive***. Prior resource-competitive algorithms have been utilized for a number of network security problems, where the aim is to impose a higher cost on the adversary for launching attacks relative to the defender's cost. For example, Bender et al. (2016), Gilbert and Young (2012), Gilbert et al. (2014), King et al. (2011), and Chen and Zheng (2020) introduce resource-competitive algorithms to combat malicious interference on broadcast channels. In permissionless systems, Gupta et al. (2023) and Gupta et al. (2018) propose defenses against the Sybil attack, while Augustine et al. (2019) presents a Byzantine agreement protocol. The approach has been applied to mixing networks, where Zamani et al. (2017) optimize bridge assignment in the Oinion Router network (see Dingledine et al. (2004). A recent application has been to denial-of-service (DoS) attacks, where Chakraborty et al. (2022) propose a resource-competitive algorithm for mitigating DoS in the client-server setting.

1.5.4. Compatibility with Prior Defenses

We emphasize that our results may be used in conjunction with many prior solutions. For example, DEPTH CHARGE can be used alongside methods that use stronger hash functions and secret keys, and also within application-specific defenses—these approaches are not mutually exclusive. Given that there is no single approach that can completely protect against ACAs, having multiple complementary tools for defense can be useful, and we view our approach as adding to a defensive "toolkit".



DEPTH CHARGE

Insert at index i:

- Respond with an $(L_i + 1)$-hard RB challenge, where L_i is the list length at index i.
- If the requester solves the RB challenge, then insert the object at the tail of the list at index i.

Query object at index i:

- Traverse the list at the index i. If the object is found, then issue a Δ-hard RB challenge to the requester, where Δ is the object's depth; else, respond with an L_i-hard challenge.
- If the requester solves the RB challenge, then if the object exists, service the query and move the object to the HoL; else, respond that the object was not found.

Figure 1: Pseudocode for DEPTH CHARGE.

2. Our Algorithm

The pseudocode for our algorithm, DEPTH CHARGE, is presented in Figure 1 and is assumed to be executed by the server; for clarity, as discussed below, we omit deletions and the specifics of message exchanges omitted.

Insertions. Upon receiving an insertion request, the server responds with an RB challenge whose hardness equals L_i+1, where L_i is list length at index i. If the server receives a valid solution to this challenge, then the object is inserted at the ToL; a pointer is assumed to be kept to the ToL in order to give $O(1)$ latency per insertion.

Queries. Upon receiving a query request for an object, the server calculates the index i where the object should be stored and traverses that list starting from the HoL. If the object is found, then the server responds with a Δ-hard RB challenge, where Δ is the object's depth. Otherwise, the server discovers that the object does not exist by traversing the entire list and then issues an L_i-hard RB challenge. In the latter case, imposing a cost mitigates spurious requests by the adversary for non-existent objects, while the cost for such requests from clients can be viewed as the price for a membership test.

If a valid solution is received and the object exists in the table, then the server services the query and also performs a move-to-front operation by repositioning the queried object to the head of its corresponding list. Otherwise, the queried object does not exist in the table, and responds that the object was not found.

Deletions. A deletion is performed almost identically to a query. However, in the case where the object is located, the object is deleted rather than being moved to the HoL. For ease of presentation, we omit deletions from the pseudocode in Figure 1.

2.1. Ensuring Payment by the Adversary

In Depth Charge, when a request is received, the server sets the hardness of the RB challenge to be issued to the requester (and this challenge must be solved prior to the request being serviced). For clarity of presentation, in our pseudocode we omit the details of how the server sets this hardness and instead discuss them here.

For each index of the table, the server maintains state on the length of the corresponding chain. When handling an insertion request, the server computes the challenge hardness by simply using this chain-length value. Thus, the algorithm incurs no latency cost for issuing the RB challenge.

But what about query and deletion operations? The server must find the corresponding object and learn its depth prior to creating the RB challenge. What happens if the adversary issues such a request, but then abandons the request without solving the RB challenge? This would impose a latency cost on the algorithm at no cost to the adversary.

To prevent this, the server responds to any initial query or deletion request with an RB challenge of hardness 1. Upon receiving a valid solution, the server only traverses the list to a depth of 1 (i.e., the object at HoL). If the object is not found there, the server then issues a second challenge of hardness of 2 and, if a valid solution is received, the server checks up to depth 2 in the corresponding list. This process continues, with the server increasing the challenge hardness by a factor of 2 until the object is located or the end of the list is reached.

If this process completes, then it guarantees that the adversary incurs an RB cost that is to within a constant factor of the requested object's depth.

Otherwise, if the adversary abandons the request partway through when, say, the challenge hardness is 2^i, then we may treat this a bad request for an object at depth $\Theta(2^i)$. In this way, this process imposes a cost on the adversary that aligns with that prescribed by our algorithm up to a constant factor; thus, for the purposes of our analysis, we ignore this aspect, since it does not alter our asymptotic results. Finally, we note that this process requires only $O(\log L)$ challenges and messages per request, where L is the list length at the corresponding table index.

3. Analysis

Our analysis of Depth Charge is presented in three pieces. First, in Section 3.1, we analyze the RB cost and latency for insertions; this is a stepping stone to proving bounds on sequences of requests. Second, in Section 3.2, we provide a bound on the longest list length (Lemma 5), which is used to establish Property 1. We then use an amortized analysis for a sequence of queries, which is combined with our bound on insertions to prove Property 2 (in Corollary 3). Third, in Section 3.3, we bound the expected RB cost and latency for a sequence of queries that occur in indices selected u.a.r., which allows us to establish Property 3 (in Lemma 8).

3.1. Insertion Cost

In this section, we analyze Depth Charge's RB cost over all $\mathcal{I}$ good insertions. Define an ***targeted index*** to be any index that contains at least one bad object and at least one good object. In the current table, let $\boldsymbol{s}$ be the number of targeted indices.

Unless specified otherwise, our analysis in this section pessimistically assumes that all targeted indices contain ℓ_M good objects; this can only increase the cost to Depth Charge.

Lemma 1. *Suppose the adversary inserts b bad balls in the s targeted indices. Then,* Depth Charge*'s RB cost for insertions into the targeted indices is at most $\frac{s\ell_M^2}{2} + b\ell_M$.*

Proof. Let x_i be the number of bad objects placed by the adversary into the i-th targeted index, where $i = 1, ..., s$. Fix any particular index i, Depth Charge's cost for this index is at most:

$$\sum_{k=1}^{\ell_M}(x_i + k) < \frac{\ell_M^2}{2} + \ell_M x_i.$$

Using the above bound, Depth Charge's insertion cost over all targeted indices is at most:

$$\begin{aligned}\sum_{i=1}^{s}\left(\frac{\ell_M^2}{2}+\ell_M x_i\right) &\leq s\frac{\ell_M^2}{2}+\ell_M\sum_{i=1}^{s}x_i\\ &= \frac{s\ell_M^2}{2}+b\ell_M\end{aligned}$$

where the second line follows from noting that $\sum_{i=1}^{s} x_i = b$. □

Lemma 2. *Suppose that the adversary inserts* $b \geq 1$ *objects in* $s \geq 1$ *targeted indices. Then,* $\mathcal{B} \geq \frac{b^2}{8s}$.

Proof. Assume that the adversary's bad objects are all added before any good objects are added to the table; this only reduces the adversary's cost. Furthermore, observe that the adversary's cost from the placement of bad objects in targeted indices is minimized when these b objects are spread as evenly as possible over the s indices. To see this, we describe two cases.

Case 1: $\boldsymbol{b \pmod{s} = 0}$. Consider any two indices, each with $x = b/s$ bad objects. In this case, the adversary's cost is $2\sum_{i=1}^{x} i$. In contrast, if we move p bad object, where $p \in [1, x]$ from one of these indices to the other, the adversary's cost is $\sum_{j=1}^{x-p} j + \sum_{k=1}^{x+p} k$. Note that the first cost minus the second cost is:

$$\begin{aligned}&2\sum_{i=1}^{x} i - \left(\sum_{j=1}^{x-p} j + \sum_{k=1}^{x+p} k\right)\\ &= ((x-p+1)+...+x) - ((x+1)+...+(x+p))\\ &< 0.\end{aligned}$$

Therefore, deviating from the case where all indices have the same number of bad objects will increase the adversary's cost.

Case 2: $\boldsymbol{b \pmod{s} \neq 0}$. In this case, consider that the inserted bad objects are spread as evenly as possible. Then, we will show that deviating from this arrangement can only increase the adversary's cost. Under this spreading of bad objects, there will be indices with $x = b/s$ bad objects and at least one index with at most $x+1$ bad objects; thus, there can be at most a difference of 1 bad object between any two indices. Consider any a "small" index and a "large" index with x and $x+1$ bad objects, respectively. In this case, the adversary's cost is $\sum_{h=1}^{x} h + \sum_{i=1}^{x+1} i$.

Moving p bad objects from the small index to the large index means that the adversary's cost is now $\sum_{j=1}^{x-p} j + \sum_{k=1}^{(x+1)+p} k$. Note that the first cost minus the second cost is:

$$\begin{aligned}
&\sum_{h=1}^{x} h + \sum_{i=1}^{x+1} i - \left(\sum_{j=1}^{(x+1)+p} j + \sum_{k=1}^{x-p} k \right) \\
&= ((x-p+1) + ... + (x+1)) \\
&\quad - ((x+1) + ... + (x+1+p)) \\
&< 0.
\end{aligned}$$

Again, deviating from the case where all indices have the same number of bad objects will increase the adversary's cost.

Given this case analysis, the adversary's cost over the s targeted indices is at least:

$$\begin{aligned}
s \sum_{i=1}^{\lfloor b/s \rfloor} i &\geq s \int_{0}^{\lfloor b/s \rfloor} i \; di \\
&= \left(\frac{s}{2}\right) (\lfloor b/s \rfloor)^2 \\
&\geq \left(\frac{s}{2}\right) (\max\{1, (b/s) - 1\})^2 \\
&\geq \left(\frac{s}{2}\right) \left(\frac{b}{2s}\right)^2 \\
&= \frac{b^2}{8s}
\end{aligned}$$

where the first line follows since i is a monotonically increasing function, and the third line holds since $b \geq s$ by definition of targeted indices and $\lfloor x \rfloor \geq x - 1$. The fourth line follows by noting that, if $\max\{1, (b/s) - 1\} = 1$, then $b/s \leq 2$ and so $b/(2s) \leq 1$, which justifies the inequality. Else, if $b/s - 1 > 1$, which implies $b/(2s) > 1$ iff $(b/s) - (b/2s) > 1$ iff $(b/s) - 1 > b/(2s)$, which again justifies the inequality (although, it is strict in this case). □

Lemma 3. *The RB cost to* DEPTH CHARGE *for insertions into targeted indices is* $O\left(\ell_M^2 \sqrt{s\mathcal{B}}\right)$.

Proof. By Lemma 2, $\mathcal{B} \geq \frac{b^2}{8s}$ for placing b objects into targeted indices. Solving for b yields $b \leq \sqrt{8s\mathcal{B}}$. Next, we use Lemma 1, which shows that the RB cost to Depth Charge due to the targeted indices is at most $\frac{s\ell_M^2}{2} + \ell_M b$. Thus, Depth Charge 's cost for good objects in targeted indices is at most:

$$\begin{aligned}
&= \frac{s\ell_M^2}{2} + b\ell_M \\
&\leq \frac{s\ell_M^2}{2} + \ell_M\sqrt{8s\mathcal{B}} \\
&= O\left(\ell_M\sqrt{s\mathcal{B}} + \ell_M^2\sqrt{s\mathcal{B}}\right)
\end{aligned}$$

where the second step holds by substituting the upper bound on b, and the third step holds since $s \leq b$. □

Define a ***good index*** to be an index containing only good objects. Having analyzed the cost to targeted indices, we now analyze the additional cost to Depth Charge due to good indices.

Lemma 4. *With high probability, the RB cost to* Depth Charge *for good insertions into good indices is $O(\mathcal{I}\ell_M^2)$.*

Proof. There are at most $\mathcal{I}$ good indices, each with ℓ_M good objects. The resource burning cost to Depth Charge for at most t such indices is at most:

$$\mathcal{I}\left(\sum_{i=1}^{\ell_M} i\right) = O\left(\mathcal{I}\ell_M^2\right)$$

which completes the argument. □

We can now bound the total RB cost to Depth Charge over the $\mathcal{I}$ insertions.

Corollary 2. *The total RB cost to* Depth Charge *for good insertions is:*

$$O\Big(\left(\sqrt{\mathcal{I}\mathcal{B}} + \mathcal{I}\right)\ell_M^2\Big).$$

Proof. This follows directly by adding up Depth Charge's cost incurred by all targeted indices and good indices as derived in Lemmas 3 and 4, respectively, and noting that $\mathcal{I} \geq s$. □

3.2. Single and Amortized Requests

We start by obtaining an upper bound on the longest list that can be created by the adversary, which in turn, provides an upper bound on the RB cost and latency for any single query. Note that this bound holds regardless of whether the corresponding object exists in the hash table, which establishes Property 1 in Theorem 1.

Lemma 5. *The maximum number of bad objects in any list is $O(\sqrt{\mathcal{B}})$ and, with high probability, the RB cost and latency for any single query is $O\left(\sqrt{\mathcal{B}} + \ell_M\right)$.*

Proof. The cost to the adversary is minimized if its bad objects are inserted in a list ahead of any good objects; thus, the cost for b bad objects is at least $\sum_{j=1}^{b} j$. Given that the adversary spends $\mathcal{B}$, the maximum number of bad objects b that can be placed in an index satisfies the following equation:

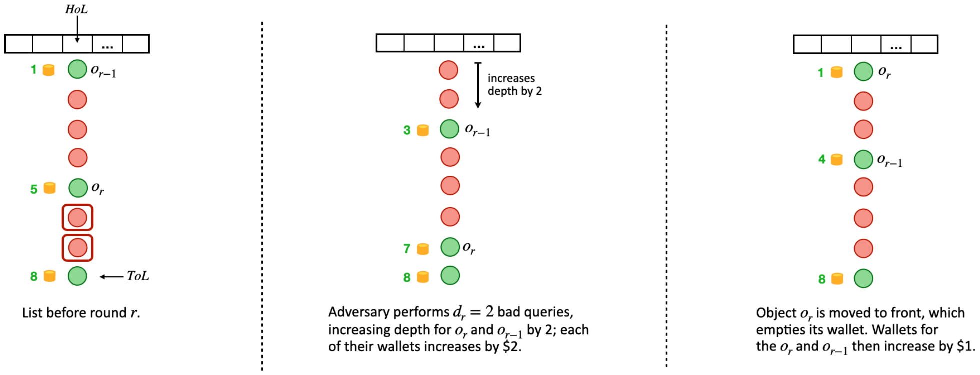
$$\begin{aligned} \mathcal{B} &\geq \sum_{j=1}^{b} j \\ &> b^2/2 \end{aligned}$$

and solving for b yields $b < \sqrt{2\mathcal{B}}$. Noting that there are at most ℓ_M good objects in this list establishes the maximum number of bad objects in the list. Finally, since the RB cost and latency for a query are both equal to the depth of the associated object, the claim follows. □

Next, we examine the cost to our algorithm under a sequence of $\boldsymbol{\mathcal{Q}}$ good queries, whose corresponding objects exist in the table. We analyze the MTF heuristic to show that the adversary must continually incur an RB cost in order cause bad latency for $\mathcal{Q}$.

Setup and Argument Overview. We first focus on a single list and the subsequence of q good queries that involve this list: we denote these queries by $Q_1, Q_2, \ldots, Q_q$ for the queried (good) objects $o_1, o_2, \ldots, o_q$. We can later aggregate the costs to Depth Charge over all lists to arrive at our final claim.

A complication arises due to the changing position of good objects over time. For example, once a good object o_r is queried under Q_r, for $1 \leq r \leq q$, we must keep track of o_r, so that we can charge Depth Charge the correct amount if it is queried again later; simply assuming the object has an RB cost



List before round r.

Adversary performs $d_r = 2$ bad queries, increasing depth for o_r and o_{r-1} by 2; each of their wallets increases by \$2.

Object o_r is moved to front, which empties its wallet. Wallets for the o_r and o_{r-1} then increase by \$1.

Figure 2: An illustration of the query analysis for some intermediate round r. Green and red balls represent good and bad objects, respectively. The amount of money in a wallet is depicted by the number of coins.

and latency equal to the list length would result in poor bounds. Additionally, queries prior to Q_r do not only increase the depth of o_r, but also every other object in the list (except for the object at the ToL), thus increasing their query cost and latency. This increase in depth is illustrated in Figure 2.

As discussed in Section 1.4, we use amortized analysis to handle such complications. Specifically, we use the accounting method, where we track Depth Charge's cost by letting each good object have a conceptual "wallet"; in this section, we speak of cost in terms of generic dollars. When queried, the good object pays for this query with dollars from its wallet. The total amount of money placed in the wallets of all good objects provides an upper bound on Depth Charge's RB cost and latency for queries.

Initially, each wallet holds a dollar amount equal to the *insertion* cost in its list. For the purposes of our accounting-method analysis, this means that when o_r is originally inserted, Depth Charge conceptually pays $L+1$ dollars for the insertion and another $L+1$ dollars as a down-payment towards its next query, where L is the list length immediately prior to the insertion of o_r. Thus, the RB cost for the first query of o_r is at least partially paid for, since o_r's wallet holds dollars equal to its depth when inserted. These extra $L + 1$ dollars are charged to the insertion of o_r; this is captured by Corollary 2.

Defining Rounds. To analyze attacks on the use of MTF in a single list, we consider a sequence of ***rounds*** also indexed by r, for $r = 1, ..., q$. Round r starts with the adversary selecting an integer value $d_r \geq 0$ and moving d_r bad objects to the HoL via d_r queries, which the adversary pays for.[5] For each good object in this list, Depth Charge places a dollar amount into each wallet equal to the increase in the depth of the corresponding good object, which is upper bounded by d_r. Then, query Q_r is executed, which brings the queried good object o_r to the HoL and reduces o_r's wallet to zero. Next, we insert an additional 1 dollar into each of the wallets of all good objects in the list whose depth increased by 1 by bringing o_r to the HoL, and also place an additional 1 dollar into o_r's wallet. After these actions are completed, round r ends.

Figure 2 illustrates the query analysis for some intermediate round r. In Figure 2 (left), the hash table's state at the end of previous round $r-1$, where object o_{r-1} resides at the HoL and the good objects hold \$1, \$5, and \$8 in their respective wallets. In Figure 2 (center), the adversary chooses $d_r = 2$ and so executes 2 bad queries, which increases the depth of the first two good objects by 2 (and does not impact the good object already at the ToL). In Figure 2 (right), a query for object o_r is executed, which empties its wallet corresponding to the cost of 7 for this query. This results in a depth of 1 for o_r, while increasing the depth of the second good object from the HoL by 1; therefore, Depth Charge adds \$1 to each of their wallets (but not to the wallet of the good object at the ToL). Thus, round r ends with each good object holding an amount of money at least equal to its current depth.

Payments by Depth Charge at the end of each round allow us to maintain the following invariant in our amortized analysis: *At the end of each round, for every good object, the amount of money in the good object's wallet is at least equal to its depth.* We leverage this invariant in our analysis of Depth Charge's RB cost and latency.

Finally, over all rounds, the adversary may schedule good and bad insertions arbitrarily. These insertions do *not* increase the depth of any good object in a list, since objects are inserted at the ToL. Consequently, Depth Charge's RB cost and latency for good insertions in any list can be accounted for separately in our analysis.

Our next lemma considers the subsequence of queries in a single list of

[5] The adversary can never increase the depth of a good object to more than the its corresponding list length; however, the adversary can perform as many bad queries as it wishes, i.e. it can set d_r to any non-negative value.

the hash table. We note that deletions are no worse than any queries, since deletions can only decrease the depth of a good object. Thus, for ease of presentation, our analysis only argues about insertions and queries, even though the statement of our final result will include deletions.

Lemma 6. *Consider any fixed list at index i in the hash table and suppose this list is involved in q_i good queries whose corresponding objects exist in the table. Let $\mathcal{A}_i^{ins}$ be* Depth Charge*'s total RB cost to insert the good objects in list i. Let $\mathcal{B}_i$ be the cost to the adversary for bad queries and bad insertions in list i. For all of the q_i queries, the total RB cost and total latency for* Depth Charge *is at most:*

$$\mathcal{A}_i^{ins} + \ell_i \left(q_i + \sqrt{2 q_i \mathcal{B}_i} \right).$$

Proof. Our aim is to guarantee that, prior to round $r \geq 1$, each good object has a number of dollars in its wallet equal to its depth. Given this, we then argue that the number of dollars in each object's wallet can pay for the cost of querying the object; notably, this cost can be either RB cost or latency, since they are both equal to the object's depth.

Round 1. We first prove that, for each good object, the depth is at most the number of dollars in the corresponding object's wallet. Initially, sometime prior to Q_1, o_1 is be inserted (since, by assumption, it exists in the table when Q_1 is executed). The wallet of o_1 contains a number of dollars equal to its depth when o_i is inserted. This is done by having o_i pay $2(L+1)$ when inserted, where L is the list length immediately prior to the o_1's insertion. The first $L+1$ dollars pay for the insertion, while the ***extra*** $L+1$ dollars are held in the o_1's wallet to help pay for the cost when it is next queried.

Any increase in depth experienced by good objects due to d_1 bad queries in round 1 results in a matching number of dollars added to each wallet. Thus, when Q_1 is executed, o_1's wallet has sufficient funds to pay for the latency of the query. Object o_1 moves to the HoL, and every good object whose depth increased by 1, along with o_1, has 1 dollar added to its corresponding wallet. These deposits to the wallets ensures that the invariant holds at the end of round 1.

Round $\geq$ 2. At the end of round $r-1$, for $r \geq 2$, each good object in the list holds a number of dollars at least equal to its depth. Thus, in round r, o_r has sufficient funds in its wallet to pay for the RB cost of Q_r. The adversary's d_r bad queries increase the depth of each good object by at most an additional

d_r, and Q_r results in all other good objects increasing their depth by at most 1. Since Depth Charge puts dollars in each good objects' wallets equal to the corresponding increase in depth due to bad queries, the invariant holds at the end of round r.

Total Cost. Depth Charge pays the following. First, the cost for all good insertions is $\mathcal{A}_i^{\text{ins}}$. Second, the algorithm pays for all the increases in depth over all rounds for all ℓ_i good objects in this list, which amounts to at most $\ell_i \sum_{r=1}^{q_i}(d_r+1)$ dollars.

In contrast, the total RB cost to the adversary is at least:

$$\begin{aligned} \mathcal{B} &\geq \sum_{r=1}^{q_i}\sum_{j=1}^{d_r} j \\ &\geq \frac{1}{2}\sum_{r=1}^{q_i} d_r^2 \\ &\geq \frac{1}{2q_i}\left(\sum_{r=1}^{q_i} d_r\right)^2 \end{aligned} \tag{1}$$

where the last line follows from Jensen's inequality for convex functions. By substituting into the algorithm's cost, we have that Depth Charge pays at most:

$$\begin{aligned} \mathcal{A}_i^{\text{ins}} + \ell_i \sum_{r=1}^{q_i}(d_r+1) &= \mathcal{A}_i^{\text{ins}} + \ell_i q_i + \ell_i \sum_{r=1}^{q_i} d_r \\ &= \mathcal{A}_i^{\text{ins}} + \ell_i q_i + \ell_i\sqrt{2q_i\mathcal{B}_i} \end{aligned}$$

where the second line follows by solving for $\sum_{r=1}^{q_i} d_r \leq \sqrt{2q_i\mathcal{B}_i}$ in Equation 1. Since $\mathcal{A}_i^{\text{ins}}$ is measured in RB cost, this concludes the bound on RB cost.

To derive total latency, recall that the RB cost for an insertion equals the depth of the object being inserted. In other words, $\mathcal{A}_i^{\text{ins}}$ equals the sum of the depths of the good objects when they are inserted. Thus, the extra dollars can also be viewed as being stored in o_i's wallet to help pay the latency when the object is next queried. This leads to the same bound on latency. □

We can now account for the algorithm's total RB cost and total latency for all good queries $\mathcal{Q}$.

Lemma 7. *The total RB cost and the total latency of* Depth Charge *due to the $\mathcal{Q}$ queries is:*

$$O\left(\left(\mathcal{I} + \mathcal{Q} + \sqrt{(\mathcal{I} + \mathcal{Q})\mathcal{B}}\right)\ell_M^2\right).$$

Proof. Let S denote the indices of the hash table where at least one good query takes place. For $i \in S$, q_i is the number of good queries that occur in this list; $\mathcal{A}_i^{\text{ins}}$ is algorithm's RB cost to insert the good objects in list i; and $\mathcal{B}_i$ be the amount spent by the adversary on bad queries in this list.

By Lemma 6, over all good queries, the total RB cost and the total latency are each at most:

$$\begin{aligned}
&\sum_{i\in S}\left(\mathcal{A}_i^{\text{ins}} + \ell_i\left(q_i + \sqrt{2q_i\mathcal{B}_i}\right)\right)\\
&\le \sum_{i\in S}\left(\mathcal{A}_i^{\text{ins}} + \ell_M\left(q_i + \sqrt{2q_i\mathcal{B}_i}\right)\right)\\
&\le \sum_{i\in S}\mathcal{A}_i^{\text{ins}} + \ell_M\sum_{i\in S} q_i + \ell_M\sqrt{2\left(\sum_{i\in S}\mathcal{B}_i\right)\left(\sum_{i\in S} q_i\right)}\\
&\le \left(\sum_{i\in S}\mathcal{A}_i^{\text{ins}}\right) + \ell_M\left(\mathcal{Q} + \sqrt{2\mathcal{B}\mathcal{Q}}\right)\\
&= O\left(\mathcal{I} + \sqrt{\mathcal{I}\mathcal{B}}\right)\ell_M^2 + \left(\left(\mathcal{Q} + \sqrt{\mathcal{Q}\mathcal{B}}\right)\ell_M\right)
\end{aligned}$$

where the first line follows since $\ell_i \le \ell_M$. The second line is obtained via the Cauchy–Schwarz inequality $\left(\sum_{i\in S}\sqrt{\mathcal{B}_i}\sqrt{q_i} \le \sqrt{\sum_i \mathcal{B}_i \sum_i q_i}\,\right)$. The third line is derived from noting $\sum_{i\in S} q_i = \mathcal{Q}$ and $\sum_{i\in S}\mathcal{B}_i \le \mathcal{B}$. The fourth line follows from Corollary 2, which states that $\sum_{i\in S}\mathcal{A}_i^{\text{ins}} = O((\mathcal{I} + \sqrt{\mathcal{I}\mathcal{B}})\ell_M^2)$.

We can rewrite the last line of our above bound on the total RB cost and total latency as:

$$O\left(\left(\mathcal{I} + \mathcal{Q} + \sqrt{\mathcal{B}}(\sqrt{\mathcal{I}} + \sqrt{\mathcal{Q}})\right)\ell_M^2\right)$$

By Jensen's inequality for concave functions:

$$\begin{aligned}
\sqrt{\mathcal{I}} + \sqrt{\mathcal{Q}} &\le \sqrt{2(\mathcal{I} + \mathcal{Q})}\\
&= O(\sqrt{\mathcal{I} + \mathcal{Q}})
\end{aligned}$$

from which the claimed result follows. □

We are now ready to bound the total RB cost and the total latency for all good insertions, along with all good queries and good deletions whose corresponding objects are in the hash table. Corollary 3 establishes the expression in Property 2 of Theorem 1.

Corollary 3. *Each of the total RB cost and the total latency for* DEPTH CHARGE *is:*

$$O\left(\left(\mathcal{I}+\mathcal{Q}+\mathcal{D}+\sqrt{(\mathcal{I}+\mathcal{Q}+\mathcal{D})\mathcal{B}}\right)\ell_M^2\right).$$

Proof. Adding the cost from all good insertions in the table, given by Corollary 2, alters the asymptotic cost given in Lemma 7. Then, since (as discussed earlier) deletions are no more costly than queries, we may replace $\mathcal{Q}$ with $\mathcal{Q}+\mathcal{D}$ to obtain the result. □

3.3. Randomly Queried Indices

Note that Corollary 3 addresses a challenging setting: deriving worst case bounds where a significant attack may be underway and the good requests are scheduled by the adversary. We conclude this section on a more optimistic note in regards to Q queries that occur in randomly chosen indices. Recall (from Section 1.3) that ℓ_i is the maximum number of good objects that are ever in bin i and that $\ell_{\text{ave}} = (1/t)\sum_{i=1}^{t}\ell_i$. We show that when Q is large relative to $\mathcal{B}$ and ℓ_M, the average query cost is $O(\ell_{\text{ave}})$ in expectation.

This result implies that, if $\ell_{\text{ave}} = O(1)$ and the adversary does not launch a significant attack, then we should expect per-query performance that matches that of standard hash tables in benign settings. The following result establishes Property 3 of Theorem 1.

Lemma 8. *Consider Q queries where the corresponding objects belong to indices of the hash table chosen u.a.r.. For $Q \geq \ell_M^2\mathcal{B}$, the average cost per query is $O(\ell_{\text{ave}})$ in expectation.*

Proof. By Lemma 6, the RB cost and latency for the i-th index are each at most:

$$\ell_i\left(Q_i+\sqrt{2Q_i\mathcal{B}_i}\right)$$

where ℓ_i, Q_i, and $\mathcal{B}_i$ are the number of good objects, number of good queries, and adversarial cost in the i-th index. Given that each query occurs in an index selected uniformly at random, in expectation over Q_i the RB cost and latency are each at most:

$$\ell_i E[Q_i] + \ell_i \sqrt{2\mathcal{B}_i} E[\sqrt{Q_i}] \leq \ell_i (Q/t) + \ell_i \sqrt{2\mathcal{B}_i} \sqrt{Q/t}$$

where the second step holds since $E[Q_i] = Q/t$, and by applying Jensen's inequality for concave functions (i.e., $E[\varphi(X)] \leq \varphi(E[X])$, where φ is a concave function and X is a random variable). Summing the above over all bins, we can bound the total RB cost and latency for queries to be at most:

$$\begin{aligned} &\sum_{i=1}^{t} \ell_i \left(Q/t + \sqrt{2\mathcal{B}_i}\sqrt{Q/t} \right) \\ &= O(Q\,\ell_{\text{ave}}) + O\left(\sqrt{2Q/t} \sum_{i=1}^{t} \ell_i \sqrt{\mathcal{B}_i} \right) \end{aligned}$$

since $\ell_{\text{ave}} = (1/t)\sum_{i=1}^{t} \ell_i$. Noting that $\ell_i \leq \ell_M$, this becomes:

$$O(Q\,\ell_{\text{ave}}) + O\left(\left(\sqrt{2Q/t} \right) t \ell_M \sqrt{\mathcal{B}/t} \right).$$

Simplifying terms yields:

$$O\left(Q\,\ell_{\text{ave}} + \ell_M \sqrt{Q\mathcal{B}} \right).$$

For $Q \geq \ell_M^2 \mathcal{B}$, if this expectation holds, then the average cost per query is $O(\ell_{\text{ave}})$, as claimed. □

4. Discussion and Future Work

We have designed and analyzed an RB-based defense against ACAs on hash tables, where the cost of our defense grows slowly with the cost of the adversary. To the best of our knowledge, our defense is the first to leverage RB for defending against ACA attacks. In this section, we discuss aspects of our defense, including those that touch on practical issues, as well as potential future work.

4.1. Delay from Communication and RB

In our model (recall Section 1.2), communication occurs instantaneously, while in practice there will be communication delay, either involving the time required to transmit data between the client and server, or due to generating RB solutions.

What happens if there is such communication delay? For example, consider the following scenario where such delay exists. Upon an initial request, the server sends a message to the client, specifying the hardness of an RB-challenge, say x. The client solves this challenge, returns its solution to the server, but the new hardness for the request has now been increased to $x+1$.

Fortunately, the x amount of RB already performed by the client is not wasted. RB work required to solve RB challenges is cumulative: the effort required to solve x 1-hard challenges is equivalent to the effort required to solve a single x-hard challenge. Consequently, if a client burns x units during the initial request, they only need to burn one additional unit to solve a $x+1$-hard challenge; thus, no RB is wasted. Given this observation, the impact of delay is equivalent to the case where the adversary inserts x bad objects into the corresponding table index ahead of this client who then must solve an $x + 1$ hard challenge, and our analysis accounts for this.

While this observation preserves our cost analysis, the number of messages can increase. Specifically, in the worst case, the client must participate in $O(x)$ message exchanges with the server instead of $O(1)$. To mitigate this, we can have the server issue a puzzle of hardness $2^{\lceil \log_2(x) \rceil}$ instead of x. For example, instead of the hardness values $1, 2, 3, 4, 5, 6, 7, 8, ...$, the server would use $1, 2, 2, 4, 4, 4, 4, 8, ...$ This preserves our cost analysis, to within a factor of 2, and has the benefit that the hardness value changes much less often. In particular, if the current hardness is 2^i for $i \geq 0$, then 2^i (valid) solutions will be accepted before the hardness value increases. Consequently, a client need only participate in $O(\log H)$ message exchanges with the server before obtaining service, where H is the maximum hardness value for this operation.

Finally, we note that the delay from generating RB solutions could increase the latency of hash table operations. However, this delay is small when there is no attack. Under significant attack, the delay will grow, but our algorithm will maintain the availability of the hash table, which would otherwise be greatly reduced in the absence of a defense.

4.2. Challenges Become Too Hard

When it comes to denial-of-service attacks, *there is no silver bullet* against a sufficiently powerful adversary. In contrast to prior results, our defense provably increases the cost on the attacker, giving legitimate clients an asymptotic advantage. Our defense is thus tailored to ACAs, given that ACA attackers often use carefully chosen inputs (recall Section 1), rather than a brute force attack that uses significant resource expenditure.

That said, our defense has limitations. Specifically, suppose the hardest RB challenge any client can solve is m. Imagine an adversary that can insert m objects into an index ahead of time, thus preventing any client from performing operations on this index. If this occurs in a small number of indices, then the hash table is still useful to many clients, as most operations will still be available.

However, the situation can be taken to an extreme where the adversary uses this tactic for most or all of the table indices, which would cost the adversary $\Omega(tm^2)$, where t represents the hash table size. Against such a powerful adversary, new ideas seem to be required. To speculate on a possible solution, imagine that for an insertion operation, the hardness of a challenge depends on the rate of operations, rather than on the chain length at the corresponding index. Then, the adversary would have to continually attack this index in order to prevent clients from executing operations at this location in the table; otherwise, the hardness would decrease over time. This may be a promising avenue for future work.

4.3. Lowering RB Costs in the Absence of Attack

In our algorithm, good clients incurs a non-zero cost, even in the absence of an attack. To elaborate, even when $\mathcal{B} = 0$, any client incurs an insertion cost of $O(\ell_M^2)$, where ℓ_M is the maximum number of good objects possible in an index. Similarly, for a query or deletion operation, a client incurs $O(\ell_M)$ despite $\mathcal{B} = 0$. While $\ell_M = O(\log n / \log\log n)$ (see Oliveira (2021), Kesselheim (2016), and Raab and Steger (1998)), it is worth considering whether these costs can be reduced when there is no attack.

A potential solution is to alter the charging scheme. Specifically, we can initiate RB challenges only after a threshold number of objects are inserted into an index; say ℓ_M. In the absence of an attack, w.h.p., the algorithm will incur zero cost for insertion of $O(\ell_M)$ objects in any index. Of course, during an attack, the adversary may insert ℓ_M objects into an index at no cost. However, this does not greatly advantage the adversary, and we expect that our asymptotic results remain valid. Similarly, query and deletion operations would only incur an RB charge only if the corresponding object's depth surpasses the threshold beyond ℓ_M. To the best of our knowledge, prior resource-competitive algorithms do not achieve zero cost in the absence of attack, which makes this an interesting approach to explore.

4.4. Additional Extensions

There are many directions for future work. First, our current approach addresses fixed-size hash tables, which captures settings where there is no need to resize the hash table, or it is not desirable to do so. However, can we extend our approach to the case when legitimate system load is unpredictable *and* there exist adequate server-side resources to resize many times?

Second, can we extend our approach to other data structures that employ hash functions, such as Bloom filters? Similarly, decentralized data structures other than those based on hash tables—such as skip graphs (see Aspnes and Shah (2003))—might also benefit from a RB-based defense, offering greater resilience for large-scale distributed systems.

Third, our upper bound on single requests may be pessimistic. Specifically, our upper bound applies to any index, but consider if only a single index is targeted by an adversary. In this case, our upper bound is loose for the other indices. It may be worthwhile to purseu an analysis that captures this aspect.

Finally, machine learning (ML) has become an important tool for improving the performance of algorithms (see Mitzenmacher and Vassilvitskii (2021) and Chakraborty et al. (2022)). In our setting, it would be interesting to determine if ML predictions about whether a request is good or bad can be leveraged to improve performance.

Acknowledgements. This work is supported by NSF awards CNS-2210299, CNS-2210300, and CCF-2144410.

References

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

Ali, I.M., Caprolu, M., Pietro, R.D., 2020. Foundations, properties, and security applications of puzzles: A survey. ACM Computing Survey 53, 1–38.

Altmeier, C., Mainka, C., Somorovsky, J., Schwenk, J., 2016. AdIDoS–adaptive and intelligent fully-automatic detection of denial-of-service weaknesses in web services, in: Proceedings of the 10th International Workshop on Data Privacy Management, and Security Assurance, pp. 65–80.

Askitis, N., Zobel, J., 2011. Redesigning the string hash table, burst trie, and BST to exploit cache. Journal of Experimental Algorithmics (JEA) 15, 1–1.

Aspnes, J., Shah, G., 2003. Skip graphs, in: Proceedings of the 14^{th} Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 384–393.

Atre, N., Sadok, H., Chiang, E., Wang, W., Sherry, J., 2022. Surgeprotector: Mitigating temporal algorithmic complexity attacks using adversarial scheduling, in: Proceedings of the ACM SIGCOMM 2022 Conference, pp. 723–738.

Augustine, J., King, V., Molla, A.R., Pandurangan, G., Saia, J., 2019. Scalable and secure computation among strangers: Resource-competitive byzantine protocols. arXiv preprint arXiv:1907.10308 .

Aumasson, J.P., Bernstein, D.J., 2012. SipHash: a fast short-input PRF, in: Proceedings of the 13th International Conference on Cryptology in India (INDOCRYPT), Springer. pp. 489–508.

Aura, T., Nikander, P., Leiwo, J., 2000. Dos resistant authentication with client puzzles, in: Revised Papers from the 8th International Workshop on Security Protocols, Springer-Verlag, Berlin, Heidelberg. p. 170–177.

Bar-Yosef, N., Wool, A., 2007. Remote algorithmic complexity attacks against randomized hash tables, in: Proceedings of the International Conference on E-Business and Telecommunications (ICETE), pp. 162–174.

Bender, M.A., Farach-Colton, M., Johnson, R., Kraner, R., Kuszmaul, B.C., Medjedovic, D., Montes, P., Shetty, P., Spillane, R.P., Zadok, E., 2012. Don't thrash: how to cache your hash on flash. Proc. VLDB Endow. , 1627–1637.

Bender, M.A., Fineman, J.T., Gilbert, S., Young, M., 2016. How to Scale Exponential Backoff: Constant Throughput, Polylog Access Attempts, and Robustness, in: Proceedings of the 27^{th} Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 636–654.

Bentley, J.L., McGeoch, C.C., 1985. Amortized analyses of self-organizing sequential search heuristics. Communications of the ACM 28, 404–411.

Borisov, N., 2006. Computational puzzles as Sybil defenses, in: Proceedings of the 6^{th} IEEE International Conference on Peer-to-Peer Computing (P2P), pp. 171–176.

Cai, X., Gui, Y., Johnson, R., 2009. Exploiting UNIX file-system races via algorithmic complexity attacks, in: 2009 30th IEEE Symposium on Security and Privacy, IEEE. pp. 27–41.

Cercone, N., 1988. Finding and applying perfect hash functions. Applied Mathematics Letters 1, 25–28.

Chakraborty, T., Islam, A., King, V., Rayborn, D., Saia, J., Young, M., 2022. Bankrupting DoS attackers. arXiv preprint arXiv:2205.08287 .

Chakraborty, T., Saia, J., Young, M., 2024. Defending hash tables from subterfuge with depth charge, in: Proceedings of the 25th International Conference on Distributed Computing and Networking, pp. 134–143.

Chang, R., Jiang, G., Ivancic, F., Sankaranarayanan, S., Shmatikov, V., 2009. Inputs of coma: Static detection of denial-of-service vulnerabilities, in: Proceedings of the 22nd IEEE Computer Security Foundations Symposium, pp. 186–199. doi:10.1109/CSF.2009.13.

Chen, H., Zheng, C., 2020. Broadcasting competitively against adaptive adversary in multi-channel radio networks, in: 24th International Conference on Principles of Distributed Systems, OPODIS, pp. 22:1–22:16.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2022. Introduction to algorithms. 4th ed., MIT press.

Crosby, S.A., Wallach, D.S., 2003. Denial of service via algorithmic complexity attacks, in: 12th USENIX Security Symposium (USENIX Security 03).

Czubak, A., Szymanek, M., 2017. Algorithmic complexity vulnerability analysis of a stateful firewall, in: Proceedings of 37th International Conference on Information Systems Architecture and Technology (ISAT), pp. 77–97.

Dingledine, R., Mathewson, N., Syverson, P.F., et al., 2004. Tor: The second-generation onion router., in: USENIX security symposium, pp. 303–320.

Dwork, C., Goldberg, A., Naor, M., 2003. On memory-bound functions for fighting spam, in: Proceedings of the Annual International Cryptology Conference, Springer. pp. 426–444.

Dwork, C., Naor, M., 1992. Pricing via processing or combatting junk mail, Springer-Verlag, Berlin, Heidelberg. p. 139–147.

Dwork, C., Naor, M., 1993. Pricing via processing or combatting junk mail, in: Proceedings of the 12^{th} Annual International Cryptology Conference on Advances in Cryptology, pp. 139–147.

Dziembowski, S., Faust, S., Kolmogorov, V., Pietrzak, K., 2015. Proofs of space, in: Annual Cryptology Conference, Springer. pp. 585–605.

Falkner, J., Piatek, M., John, J.P., Krishnamurthy, A., Anderson, T., 2007. Profiling a million user DHT, in: Proceedings of the 7^{th} ACM SIGCOMM Conference on Internet Measurement, pp. 129–134.

Franklin, J., Paxson, V., Perrig, A., Savage, S., 2007. An inquiry into the nature and causes of the wealth of internet miscreants, in: Proceedings of the 14^{th} ACM Conference on Computer and Communications Security, pp. 375–388.

Franklin, M.K., Malkhi, D., 1997. Auditable metering with lightweight security, in: Proceedings of the First International Conference on Financial Cryptography, Springer-Verlag, Berlin, Heidelberg. p. 151–160.

Gilbert, S., King, V., Pettie, S., Porat, E., Saia, J., Young, M., 2014. (Near) optimal resource-competitive broadcast with jamming, in: Proceedings of the 26^{th} ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 257–266.

Gilbert, S., Young, M., 2012. Making Evildoers Pay: Resource-Competitive Broadcast in Sensor Networks, in: Proceedings of the 31^{th} Symposium on Principles of Distributed Computing (PODC), pp. 145–154.

Gilbert, S., Zheng, C., 2013. Sybilcast: Broadcast on the open airwaves, in: Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 130–139.

Gupta, D., Saia, J., Young, M., 2018. Proof of work without all the work, in: Proceedings of the 19th International Conference on Distributed Computing and Networking (ICDCN).

Gupta, D., Saia, J., Young, M., 2020. Resource burning for permissionless systems, in: Proceedings of the International Colloquium on Structural Information and Communication Complexity, Springer. pp. 19–44.

Gupta, D., Saia, J., Young, M., 2023. Bankrupting sybil despite churn. Journal of Computer and System Sciences 135, 89–124.

Hauke, N., Renardy, D., 2019. Denial of service with a fistful of packets: Exploiting algorithmic complexity vulnerabilities. Black Hat USA .

Heimes, C., 2013. Alternative counter measures against hash collision DoS. https://peps.python.org/pep-0456/#alternative-counter-measures-against-hash-collision-dos.

Hester, J.H., Hirschberg, D.S., 1985. Self-organizing linear search. ACM Computing Surveys 17, 295–311.

IBM, 2023. Considerations for sizing hash tables. `https://www.ibm.com/docs/en/iirfz/11.3.0?topic=analysis-considerations-sizing-hash-tables`.

Kesselheim, T., 2016. Load balancing and chernoff bounds. www.mpi-inf.mpg.de/fileadmin/inf/d1/teaching/summer16/random/loadbalancing.pdf.

Khan, S., Traore, I., 2005. A prevention model for algorithmic complexity attacks, in: Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Springer. pp. 160–173.

King, V., Saia, J., Young, M., 2011. Conflict on a Communication Channel, in: Proceedings of the 30th Symposium on Principles of Distributed Computing (PODC), pp. 277–286.

Kirrage, J., Rathnayake, A., Thielecke, H., 2013. Static analysis for regular expression denial-of-service attacks, in: Proceedings of the 7th International Conference on Network and System Security (NSS), pp. 135–148.

Koblitz, N., Menezes, A.J., 2015. The Random Oracle Model: A Twenty-Year Retrospective. Designs, Codes and Cryptography 77, 587–610.

Li, F., Mittal, P., Caesar, M., Borisov, N., 2012. SybilControl: Practical Sybil defense with computational puzzles, in: Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing, pp. 67–78.

Lin, I.C., Liao, T.C., 2017. A survey of blockchain security issues and challenges. International Journal of Network Security 19, 653–659.

Lu, Y., Prabhakar, B., Bonomi, F., 2006. Perfect hashing for network applications, in: 2006 IEEE International Symposium on Information Theory, IEEE. pp. 2774–2778.

Majewski, B.S., Wormald, N.C., Havas, G., Czech, Z.J., 1996. A family of perfect hashing methods. The Computer Journal 39, 547–554.

McIlroy, M.D., 1999. A killer adversary for quicksort. Software: Practice and Experience 29, 341–344.

Metreveli, Z., Zeldovich, N., Kaashoek, M.F., 2012. CPHash: A cache-partitioned hash table. ACM SIGPLAN Notices 47, 319–320.

Mitzenmacher, M., Vassilvitskii, S., 2021. Algorithms with Predictions. In *Beyond the Worst-Case Analysis of Algorithms*. T. Roughgarden, Ed. Cambridge University Press. doi:10.1017/9781108637435.

Munro, J.I., Papadakis, T., Sedgewick, R., 1992. Deterministic skip lists, in: Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 367–375.

Nakamoto, S., 2008. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf.

Namjoshi, K., Narlikar, G., 2010. Robust and fast pattern matching for intrusion detection, in: Proceedings of the IEEE International Conference on Computer Communications (INFOCOM), IEEE. pp. 1–9.

Neudecker, T., 2017. Bitcoin cash (BCH) Sybil nodes on the Bitcoin peer-to-peer network. http://dsn.tm.kit.edu/publications/files/332/bch_sybil.pdf.

Oikonomou, G., Mirkovic, J., 2009. Modeling human behavior for defense against flash-crowd attacks, in: Proceedings of the IEEE International Conference on Communications, pp. 1–6.

Olekšák, M., Miškovský, V., 2022. Correlation power analysis of SipHash, in: Proceedings of the 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), pp. 84–87. doi:10.1109/DDECS54261.2022.9770139.

Oliveira, R., 2021. Lecture 4: Balls & bins. cs.uwaterloo.ca/∼r5olivei/courses/2021-spring-cs466/lecture04.pdf.

Raab, M., Steger, A., 1998. "Balls into bins"—A simple and tight analysis, in: International Workshop on Randomization and Approximation Techniques in Computer Science, Springer. pp. 159–170.

Reviriego, P., Larrabeiti, D., 2020. Denial of service attack on cuckoo filter based networking systems. IEEE Communications Letters 24, 1428–1432. doi:10.1109/LCOMM.2020.2983405.

Reviriego, P., Rottenstreich, O., 2020. Pollution attacks on counting bloom filters for black box adversaries, in: Proceedings of the 16th International Conference on Network and Service Management (CNSM), pp. 1–7. doi:10.23919/CNSM50824.2020.9269076.

Reviriego, P., Ting, D., 2020. Security of hyperloglog (HLL) cardinality estimation: Vulnerabilities and protection. IEEE Communications Letters 24, 976–980. doi:10.1109/LCOMM.2020.2972895.

Rivest, R., 1976. On self-organizing sequential search heuristics. Communications of the ACM 19, 63–67.

Song, T., Yang, Y., Crowley, P., 2017. RwHash: Rewritable hash table for fast network processing with dynamic membership updates, in: Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), IEEE. pp. 142–152.

Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H., 2001. Chord: A scalable peer-to-peer lookup service for internet applications, in: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), pp. 149–160.

Sun, X., Cheng, L., Zhang, Y., 2011. A covert timing channel via algorithmic complexity attacks: Design and analysis, in: 2011 IEEE International Conference on Communications (ICC), IEEE. pp. 1–5.

Tobin, R.J., Malone, D., 2012. Hash pile ups: Using collisions to identify unknown hash functions, in: Proceedings of the 7th International Conference on Risks and Security of Internet and Systems (CRiSIS), pp. 1–6.

Von Ahn, L., Blum, M., Hopper, N.J., Langford, J., 2003. CAPTCHA: Using hard AI problems for security, in: Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques, Springer. pp. 294–311.

Walfish, M., Vutukuru, M., Balakrishnan, H., Karger, D., Shenker, S., 2006. DDoS defense by offense, in: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), pp. 303–314.

Walfish, M., Vutukuru, M., Balakrishnan, H., Karger, D., Shenker, S., 2010. DDoS defense by offense. ACM Transactions on Computer Systems (TOCS) 28, 3.

Wang, L., Kangasharju, J., 2013. Measuring large-scale distributed systems: Case of BitTorrent Mainline DHT, in: IEEE 13th International Conference on Peer-to-Peer Computing (P2P), pp. 1–10.

Wang, X., Reiter, M.K., 2003. Defending against denial-of-service attacks with puzzle auctions, in: Proceedings of the 2003 IEEE Symposium on Security and Privacy, pp. 78–92.

Waters, B., Juels, A., Halderman, A., Felten, E., 2004. New client puzzle outsourcing techniques for DoS resistance, in: Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS), pp. 246–256.

Zamani, M., Saia, J., Crandall, J., 2017. TorBricks: Blocking-Resistant Tor Bridge Distribution, in: International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Springer. pp. 426–440.

Zhang, G., Sanchez, D., 2019. Leveraging caches to accelerate hash tables and memoization, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 440–452.

Zobel, J., Heinz, S., Williams, H.E., 2001. In-memory hash tables for accumulating text vocabularies. Information Processing Letters 80, 271–277.