Defending Hash Tables from Subterfuge with Depth Charge*

Trisha Chakraborty[1], Jared Saia[2], and Maxwell Young[3]

[1,3]Department of Computer Science and Engineering, Mississippi State University, MS, USA
tc2006@msstate.edu, myoung@cse.msstate.edu

[2]Department of Computer Science, University of New Mexico, NM, USA
saia@cs.unm.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.

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 [19, 8, 61]. In an extreme case, all objects can be hashed to the same 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)*** [8, 19, 14, 59, 35, 9]. 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 [48], 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 1.4, but a fundamental shortcoming of prior defenses is that 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

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

ACA. In practice, attackers must often pay for the resources needed to launch attacks, such as renting compromised machines (see [26]). Therefore, the asymptotic advantage given by our approach ultimately translates into a financial edge for the defenders.

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.[1] 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 [33]. In peer-to-peer systems such as distributed hash tables (DHTs) (e.g. [65, 25, 58, 57]), 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 ℓ_M, which is the maximum number of (legitimate) objects that are hashed to the same index. Intuitively, this parameter is small when the table is appropriately sized (see Section 1.2 for discussion).

1.1 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 said to be a ***good object***, which is placed at an index selected uniformly at random (u.a.r.).[2] 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; similarly, 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; 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.[3]

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. We note that, in practice, the situation may be even more restrictive: a client might only be allowed to delete those objects it inserts. This can be accomplished by having the client share a secret with the server in order to verify ownership of the object.

[1] Our approach could be combined with heuristics that dynamically update the number of indices or the hash function.

[2] We can view this as performing a hash function evaluation on the object or, more commonly, an identifier/key value/name of the object. The output is an index in the hash table whose first evaluation is an index of selected uniformly at random, while subsequent evaluations of this object always map to the same index (i.e., our hash function obeys the random oracle assumption [37]).

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

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 [67]. 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 [2, 38, 64]).

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.

To quantify (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.

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 (see Section 3.2). 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 uniformly at random 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., generating/scheduling 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).

1.2 Main Results

For requests, we let $\mathcal{I}$ be the number of good insertions; let $\mathcal{Q}$ and $\mathcal{D}$ 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 t; the maximum number of good objects in any index by ℓ_M. The number of good objects in index i is ℓ_i, and the average number of good objects, $(1/t)\sum_{i=1}^{t}\ell_i$, is denoted by ℓ_{ave}.

Throughput, we use $\mathcal{B}$ 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 a 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.***).[4] Notably, for $\mathcal{I} = O(t)$, it is well known that $\ell_M = O(\log t / \log \log t)$ [49, 34, 51]. This case is pertinent, since many applications limit the amount by which their hash table can grow (see [19]), and the number of size increases may be very limited (e.g. see [20]).[5] 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.

RB is a well-established tool for securing distributed systems [28]. We note that the choice of resource to burn likely depends on the specifics of the application. Given this, our algorithm is deliberately agnostic about the resource burned, such as computational power [66], bandwidth [63], computer memory [1, 22, 24], and human effort [62, 47].

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 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.

1.3 Technical Overview

At a high level, our analysis relies on upper bounding the algorithm'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 , and that there are ℓ_M good insertions in every targeted index.

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

[5] For example, the table in the Cisco router examined in [20] has an initial size of 1024, and can increase to sizes 2048, 4096, and 8192.

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 places 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 [11, 32, 55] is known to improve performance in chained hash tables when they are not under attack [69, 4, 56]. 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 the algorithm next queries o, it will pay an RB cost of $\Theta(d)$, and the query requires $\Theta(d)$ latency. Thus, the algorithm obtains a quadratic advantage, similar to what is achieved for our bound on insertion costs.

This charging scheme motivates the name DEPTH CHARGE[6] 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 [18]. Each good object is given a (conceptual) wallet into which the algorithm 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 the algorithm'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 the algorithm 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 $\mathcal{B}_i \geq$

[6] At the risk of ruining a pun via explanation, a depth charge also is a defense against subs/subterfuge.

$\sum_{r=1}^{q_i} d_r^2 = \Omega((1/q_i)(\sum_{r=1}^{q_i} d_r)^2)$ 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 the algorithm'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 3.2, we sum up the costs to the algorithm 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.

1.4 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.

Defenses using Resource Burning. The use of RB as a tool for solving security problems spans several decades (e.g. see the surveys [2, 28]). RB-based defenses arise in many contexts, such as spam mitigation [22, 23], wireless networks [27], peer-to-peer systems [38, 13], blockchains [39], the Sybil attack [29], and denial-of-service attacks [64, 50, 16].

Prior Defenses for ACAs. Many other common data structures and algorithms are vulnerable to ACAs, such as linked lists [6], quicksort [35, 42], cardinality sketches [54], pattern matchers [36, 46, 12], cuckoo filters [52], and bloom filters [53]. As a result, AC attacks can impact common applications: networked applications [17, 6], firewalls [20], web services [3], PDF compressors [30], TCP reassembler [21, 60], and intrusion detection systems [19].

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

(1) Choice of Hash Functions. Crosby et al. [19] 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, Yosef et al. [8] 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, SipHash [7] 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 [48] 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 [40, 15, 41]). However, constructing perfect hash functions is time consuming and requires knowing the set of objects to be hashed, which is not always available.

(2) 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 [31]. Another approach is the use of caching to store pre-computed results of expensive hash table lookups [68, 43, 10].

(3) 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 [45] 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 [20] 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.

Compatibility with Prior Defenses. Our algorithmic 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.

2 Our Algorithm

The pseudocode for our algorithm (with deletions omitted), DEPTH CHARGE, is presented in Figure 1 and is assumed to be executed by the server. Below, are the requests supported.

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, which is assumed to require constant latency. 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.

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.

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 3.2). 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 the algorithm'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 the algorithm.

Lemma 1. *Suppose the adversary places b bad balls in the s targeted indices. Then, the algorithm's RB cost for insertions into the attack indices is at most $s\ell_M^2 + b\ell_M$.*

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, the algorithm'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, the algorithm's insertion cost over all targeted indices is at most:

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

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

Lemma 2. *Suppose that the adversary places $b \geq 1$ 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 $b \pmod s = 0$. 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 $b \pmod s \neq 0$. In this case, there will be indices with x objects and at least one index with $x+1$ bad objects; thus, there can be at most a difference of 1 bad object between any two indices. Consider any "small" index and any "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:

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

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:

$$s\sum_{i=1}^{\lfloor b/s \rfloor} i \geq s\int_{0}^{\lfloor b/s \rfloor} i\, di$$

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

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 the algorithm 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 the algorithm due to the targeted indices is at most $s\ell_M^2 + \ell_M b$. Thus, the algorithm's cost for good objects in targeted indices is at most:

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

where the second step holds by substituting the upper bound on b. Noting that $\mathcal{B} \geq s$ yields the claim. □

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 the algorithm due to good indices.

Lemma 4. *With high probability, the RB cost to the algorithm 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 the algorithm for at most t such indices is at most:

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

which completes the argument. □

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

Corollary 3.1. *The total RB cost to the algorithm 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 the algorithm'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 the algorithm 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 the algorithm the correct amount if it is queried again later; simply assuming the object has an RB cost 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 1.3, we use amortized analysis to handle such complications. Specifically, we use the accounting method, where we track the algorithm'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 the algorithm'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, the algorithm 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 3.1.

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.[7] For each good object in this list, the algorithm 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.

Payments by the algorithm 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 the algorithm'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, the algorithm'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 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 the algorithm'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 the algorithm 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

[7] 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.

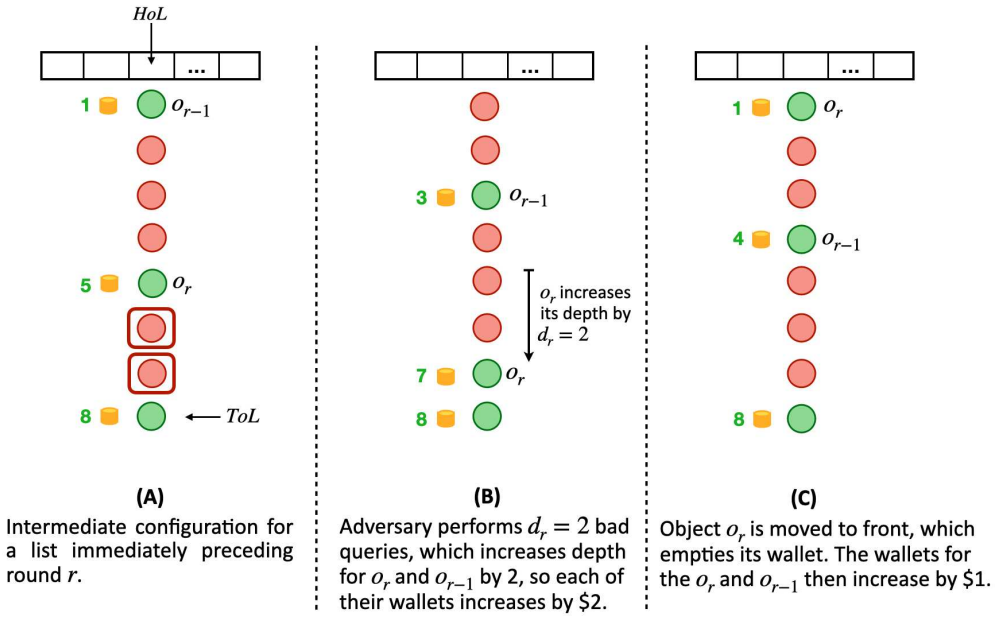

Figure 2: An illustration of the query analysis for some intermediate round r. Green and red balls represent good and bad objects, respectively. The amount of money in a wallet is depicted by the number of coins. (A) This is the hash table's state at the end of previous round $r-1$, where object o_{r-1} resides at the HoL and the good objects hold \$1, \$5, and \$8 in their respective wallets. (B) 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). (C) Query Q_r 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, the algorithm 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.

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 the algorithm 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. The algorithm 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 the algorithm pays at most:

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

where the second line follows by solving for $\sum_{r=1}^{q_i} d_r \leq \sqrt{2 q_i \mathcal{B}_i}$ 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 the algorithm 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 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:

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

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

where the first line follows since $\ell_i \le \ell_M$. The second line is obtained via the Cauchy–Schwarz inequality $\left(\sum_{i\in S}\sqrt{B_i}\sqrt{q_i} \le \sqrt{\sum_i B_i \sum_i q_i}\,\right)$. The third line is derived from noting $\sum_{i\in S} q_i = \mathcal{Q}$ and $\sum_{i\in S} B_i \le \mathcal{B}$. The fourth line follows from Corollary 3.1, which states that $\sum_{i\in S} \mathcal{A}_i^{\text{ins}} = O(\mathcal{I} + \sqrt{\mathcal{I}\mathcal{B}})$. □

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 3.2 establishes the expression in Property 2 of Theorem 1.

Corollary 3.2. *Each of the total RB cost and the total latency for the algorithm 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 3.1, 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 3.2 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 1.2) 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 $Q \ge \ell_M^2\mathcal{B}$, the average cost per query is $O(\ell_{ave})$ 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:

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

where the second step holds since $E[Q_i] = Q/t$, and by applying Jensen's inequality for concave functions (i.e., $E[\varphi(X)] \le \varphi(E[X])$, 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:

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

where the first line follows since $\ell_{\text{ave}} = (1/t)\sum_{i=1}^{t} \ell_i$, the second line follows from again noting that $\ell_i \leq \ell_M$, and the third line follows from simplifying terms. 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 Conclusion and Future Work

In this paper, we have designed and analyzed an RB-based defense against algorithmic complexity attacks on hash tables. The cost of our defense grows slowly with the cost of the adversary. For example, when there is little-to-no attack, the defense has low RB cost and latency; conversely, when the attack is significant, our defense imposes an asymptotically-higher RB cost on the adversary.

To the best of our knowledge, our defense is the first to leverage RB for defending against ACA attacks, and there are several promising questions for future work. First, our current work 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 [5]—might also benefit from such a defense.

Third, can we derive asymptotically matching lower bounds for the algorithm's performance? Is there a fundamental trade off between these RB cost and latency?

Fourth, we

Finally, machine learning (ML) has become an important tool for improving the performance of algorithms [44, 16]. 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 the bounds on the cost ratio or latency.

Acknowledgements. We are grateful to the anonymous reviewers for their comments, which allowed us to improve our manuscript.

References

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

[2] Isra Mohamed Ali, Maurantonio Caprolu, and Roberto Di Pietro. Foundations, properties, and security applications of puzzles: A survey. *ACM Computing Survey*, 53(4):1–38, 2020.

[3] Christian Altmeier, Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. 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*, pages 65–80, 2016.

[4] Nikolas Askitis and Justin Zobel. Redesigning the string hash table, burst trie, and BST to exploit cache. *Journal of Experimental Algorithmics (JEA)*, 15:1–1, 2011.

[5] James Aspnes and Gauri Shah. Skip graphs. In *Proceedings of the* 14^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 384–393, 2003.

[6] Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, and Justine Sherry. Surgeprotector: Mitigating temporal algorithmic complexity attacks using adversarial scheduling. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 723–738, 2022.

[7] Jean-Philippe Aumasson and Daniel J Bernstein. SipHash: a fast short-input PRF. In *Proceedings of the 13th International Conference on Cryptology in India (INDOCRYPT)*, pages 489–508. Springer, 2012.

[8] Noa Bar-Yosef and Avishai Wool. Remote algorithmic complexity attacks against randomized hash tables. In *Proceedings of the International Conference on E-Business and Telecommunications (ICETE)*, pages 162–174, 2007.

[9] Udi Ben-Porat, Anat Bremler-Barr, and Hanoch Levy. Computer and network performance: graduating from the "age of innocence". *Computer Networks*, 66:68–81, 2014.

[10] Michael A Bender, Martin Farach-Colton, Rob Johnson, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. Don't thrash: how to cache your hash on flash. In *Proceedings of the 3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*, 2011.

[11] Jon L Bentley and Catherine C McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Communications of the ACM*, 28(4):404–411, 1985.

[12] Martin Berglund, Frank Drewes, and Brink Van Der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. *arXiv preprint arXiv:1405.5599*, 2014.

[13] Nikita Borisov. Computational puzzles as Sybil defenses. In *Proceedings of the* 6^{th} *IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 171–176, 2006.

[14] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting UNIX file-system races via algorithmic complexity attacks. In *2009 30th IEEE Symposium on Security and Privacy*, pages 27–41. IEEE, 2009.

[15] Nick Cercone. Finding and applying perfect hash functions. *Applied Mathematics Letters*, 1(1):25–28, 1988.

[16] Trisha Chakraborty, Abir Islam, Valerie King, Daniel Rayborn, Jared Saia, and Maxwell Young. Bankrupting DoS attackers. *arXiv preprint arXiv:2205.08287*, 2022.

[17] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, pages 186–199, 2009.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[19] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.

[20] Adam Czubak and Marcin Szymanek. Algorithmic complexity vulnerability analysis of a stateful firewall. In *Proceedings of 37th International Conference on Information Systems Architecture and Technology (ISAT)*, pages 77–97, 2017.

[21] Sarang Dharmapurikar and Vern Paxson. Robust TCP stream reassembly in the presence of adversaries. In *Proceedings of the 14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association.

[22] Cynthia Dwork, Andrew Goldberg, and Moni Naor. On memory-bound functions for fighting spam. In *Proceedings of the Annual International Cryptology Conference*, pages 426–444. Springer, 2003.

[23] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Proceedings of the* 12^{th} *Annual International Cryptology Conference on Advances in Cryptology*, pages 139–147, 1993.

[24] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Annual Cryptology Conference*, pages 585–605. Springer, 2015.

[25] Jarret Falkner, Michael Piatek, John P. John, Arvind Krishnamurthy, and Thomas Anderson. Profiling a million user DHT. In *Proceedings of the* 7^{th} *ACM SIGCOMM Conference on Internet Measurement*, pages 129–134, 2007.

[26] Jason Franklin, Vern Paxson, Adrian Perrig, and Stefan Savage. 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*, pages 375–388, 2007.

[27] Seth Gilbert and Chaodong Zheng. Sybilcast: Broadcast on the open airwaves. In *Proceedings of the* 25^{th} *Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 130–139, 2013.

[28] Diksha Gupta, Jared Saia, and Maxwell Young. Resource burning for permissionless systems. In *Proceedings of the International Colloquium on Structural Information and Communication Complexity*, pages 19–44. Springer, 2020.

[29] Diksha Gupta, Jared Saia, and Maxwell Young. Bankrupting Sybil despite churn. In *Proceedings of the 41st IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2021.

[30] Nathan Hauke and David Renardy. Denial of service with a fistful of packets: Exploiting algorithmic complexity vulnerabilities. *Black Hat USA*, 2019.

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

[32] James H. Hester and Daniel S. Hirschberg. Self-organizing linear search. *ACM Computing Surveys*, 17(3):295–311, 1985.

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

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

[35] Suraiya Khan and Issa Traore. A prevention model for algorithmic complexity attacks. In *Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 160–173. Springer, 2005.

[36] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static analysis for regular expression denial-of-service attacks. In *Proceedings of the 7th International Conference on Network and System Security (NSS)*, pages 135–148, 2013.

[37] Neal Koblitz and Alfred J Menezes. The Random Oracle Model: A Twenty-Year Retrospective. *Designs, Codes and Cryptography*, 77(2-3):587–610, 2015.

[38] Frank Li, Prateek Mittal, Matthew Caesar, and Nikita Borisov. SybilControl: Practical Sybil defense with computational puzzles. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, pages 67–78, 2012.

[39] Iuon-Chang Lin and Tzu-Chun Liao. A survey of blockchain security issues and challenges. *International Journal of Network Security*, 19(5):653–659, 2017.

[40] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Perfect hashing for network applications. In *2006 IEEE International Symposium on Information Theory*, pages 2774–2778. IEEE, 2006.

[41] Bohdan S Majewski, Nicholas C Wormald, George Havas, and Zbigniew J Czech. A family of perfect hashing methods. *The Computer Journal*, 39(6):547–554, 1996.

[42] M. Douglas McIlroy. A killer adversary for quicksort. *Software: Practice and Experience*, 29(4):341–344, 1999.

[43] Zviad Metreveli, Nickolai Zeldovich, and M Frans Kaashoek. CPHash: A cache-partitioned hash table. *ACM SIGPLAN Notices*, 47(8):319–320, 2012.

[44] Michael Mitzenmacher and Sergei Vassilvitskii. *Algorithms with Predictions. In Beyond the Worst-Case Analysis of Algorithms*. T. Roughgarden, Ed. Cambridge University Press, 2021.

[45] J Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 367–375, 1992.

[46] Kedar Namjoshi and Girija Narlikar. Robust and fast pattern matching for intrusion detection. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–9. IEEE, 2010.

[47] Georgios Oikonomou and Jelena Mirkovic. Modeling human behavior for defense against flash-crowd attacks. In *Proceedings of the IEEE International Conference on Communications*, pages 1–6, 2009.

[48] Matúš Olekšák and Vojtěch Miškovský. Correlation power analysis of SipHash. In *Proceedings of the 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 84–87, 2022.

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

[50] Bryan Parno, Dan Wendlandt, Elaine Shi, Adrian Perrig, Bruce Maggs, and Yih-Chun Hu. Portcullis: Protecting connection setup from denial-of-capability attacks. *ACM SIGCOMM Computer Communication Review*, 37(4):289–300, 2007.

[51] Martin Raab and Angelika Steger. "Balls into bins"—A simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.

[52] Pedro Reviriego and David Larrabeiti. Denial of service attack on cuckoo filter based networking systems. *IEEE Communications Letters*, 24(7):1428–1432, 2020.

[53] Pedro Reviriego and Ori Rottenstreich. Pollution attacks on counting bloom filters for black box adversaries. In *Proceedings of the 16th International Conference on Network and Service Management (CNSM)*, pages 1–7, 2020.

[54] Pedro Reviriego and Daniel Ting. Security of hyperloglog (HLL) cardinality estimation: Vulnerabilities and protection. *IEEE Communications Letters*, 24(5):976–980, 2020.

[55] Ronald Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19(2):63–67, 1976.

[56] Tian Song, Yating Yang, and Patrick Crowley. 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)*, pages 142–152. IEEE, 2017.

[57] Tyron Stading, Petros Maniatis, and Mary Baker. Peer-to-peer caching schemes to address flash crowds. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 203–213, 2002.

[58] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 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)*, pages 149–160, 2001.

[59] Xiaoshan Sun, Liang Cheng, and Yang Zhang. A covert timing channel via algorithmic complexity attacks: Design and analysis. In *2011 IEEE International Conference on Communications (ICC)*, pages 1–5. IEEE, 2011.

[60] Juha-Matti Tilli. CVE-2018-5390: Linux Kernel TCP Reassembly Algorithm Lets Remote Users Consume Excessive CPU Resources on the Target System. https://ubuntu.com/security/cve-2018-5390, 2018.

[61] R. Joshua Tobin and David Malone. 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)*, pages 1–6, 2012.

[62] Luis Von Ahn, Manuel Blum, Nicholas J Hopper, and John Langford. CAPTCHA: Using hard AI problems for security. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*, pages 294–311. Springer, 2003.

[63] Michael Walfish, Mythili Vutukuru, Hari Balakrishnan, David Karger, and Scott Shenker. DDoS defense by offense. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 303–314, 2006.

[64] Michael Walfish, Mythili Vutukuru, Hari Balakrishnan, David Karger, and Scott Shenker. DDoS defense by offense. *ACM Transactions on Computer Systems (TOCS)*, 28(1):3, 2010.

[65] L. Wang and J. Kangasharju. Measuring large-scale distributed systems: Case of BitTorrent Mainline DHT. In *IEEE 13th International Conference on Peer-to-Peer Computing (P2P)*, pages 1–10, 2013.

[66] XiaoFeng Wang and Michael K. Reiter. Defending against denial-of-service attacks with puzzle auctions. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 78–92, 2003.

[67] Brent Waters, Ari Juels, Alex Halderman, and Edward Felten. New client puzzle outsourcing techniques for DoS resistance. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 246–256, 2004.

[68] Guowei Zhang and Daniel Sanchez. Leveraging caches to accelerate hash tables and memoization. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pages 440–452, 2019.

[69] Justin Zobel, Steffen Heinz, and Hugh E Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277, 2001.