Algorithmic Complexity Attacks on Dynamic Learned Indexes

Rui Yang
University of Virginia
qgh4hm@virginia.edu

Evgenios M. Kornaropoulos
George Mason University
evgenios@gmu.edu

Yue Cheng
University of Virginia
mrz7dp@virginia.edu

ABSTRACT

Learned Index Structures (LIS) view a sorted index as a model that learns the data distribution, takes a data element key as input, and outputs the predicted position of the key. The original LIS can only handle lookup operations with no support for updates, rendering it impractical to use for typical workloads. To address this limitation, recent studies have focused on designing efficient dynamic learned indexes. ALEX, as the first and one of the representative dynamic learned index structures, enables dynamism by incorporating a series of design choices, including adaptive key space partitioning, dynamic model retraining, and sophisticated engineering and policies that prioritize read/write performance. While these design choices offer improved average-case performance, the emphasis on flexibility and performance increases the attack surface by allowing adversarial behaviors that maximize ALEX's memory space and time complexity in worst-case scenarios.

In this work, we present the first systematic investigation of algorithmic complexity attacks (ACAs) targeting the worst-case scenarios of ALEX. We introduce new ACAs that fall into two categories, space ACAs and time ACAs, which target the memory space and time complexity, respectively. First, our space ACA on data nodes exploits ALEX's gapped array layout and uses Multiple-Choice Knapsack (MCK) to generate an optimal adversarial insertion plan for maximizing the memory consumption at the data node level. Second, our space ACA on internal nodes exploits ALEX's catastrophic cost mitigation mechanism, causing an out-of-memory (OOM) error with only a few hundred adversarial insertions. Third, our time ACA generates pathological insertions to increase the disparity between the actual key distribution and the linear models of data nodes, deteriorating the runtime performance by up to $1,641\times$ compared to ALEX operating under legitimate workloads.

PVLDB Reference Format:
Rui Yang, Evgenios M. Kornaropoulos, and Yue Cheng. Algorithmic Complexity Attacks on Dynamic Learned Indexes. PVLDB, 17(4): 780 - 793, 2023.
doi:10.14778/3636218.3636232

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ds2-lab/aca-dlis.

Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636232

1 INTRODUCTION

Index structures are essential to database systems for efficient data storage and retrieval. Traditional database index structures precisely organize each and every data item to balance tradeoffs in query performance and memory efficiency. Recent work by Kraska, Beutel, Chi, Dean, and Polyzotis [35] proposes to replace the traditional database index with a hierarchy of machine learning (ML) models, which is called *Learned Index Structure* (LIS). The insight is that a sorted index can be viewed as a model, which learns the data distribution, takes a data element key as input, and outputs the location of the key stored in a storage medium (e.g., a disk). Under LIS, a query starts by consulting a root model at the top of the hierarchy in order to predict the child model to use, and performs a model traversal downwards the model hierarchy until it reaches a leaf node; the leaf node is also a model that will return the predicted position of the key from a densely packed array.

The very first LIS proposed in the literature could only handle read-only data with no support for update operations. This limitation made LIS unusable for dynamic read-write workloads. To address this gap, various dynamic index structures have been introduced, including ALEX [19], PGM [22], LIPP [61], and APEX [38], among others. In our work, we select ALEX [19] as our target for our algorithmic complexity attacks (ACAs), because not only is ALEX the *first* dynamic learned index (DLIS), but also (1) ALEX has *qualitative features* that make it a strong candidate for real deployment, and (2) there is strong *quantitative evidence* that ALEX is the *most efficient and well-rounded* DLIS that can support real workloads, see the recent benchmark studies [56, 60] that rank ALEX highly across all tested performance metrics. Similar to LIS, ALEX organizes models in a tree structure and serves requests by traversing the tree from top to bottom. Unlike LIS, ALEX allows both the internal nodes and leaf nodes (data nodes that store the data element) to dynamically grow and shrink at difference rates. To achieve efficient lookup and insertion operations, ALEX uses an array with gaps (called a *gapped array*). ALEX also introduces heuristic-guided, dynamic node expansion/splitting mechanisms, paired with model retraining, to adapt to workload changes.

The dynamism of ALEX over LIS is achieved by enabling (1) adaptive key space repartitioning, (2) dynamic model retraining, and (3) sophisticated engineering that prioritizes performance. While these design choices offer improved average-case performance, the emphasis on flexibility and performance increases the attack surface by allowing adversarial behaviors that maximize ALEX's space and time complexity in *worst-case scenarios*. These worst-case scenarios can be exploited by a type of attack method called algorithmic complexity attacks (ACAs) [14, 28]. ACAs are a class of Denial-of-Service (DoS) attacks, where an attacker uses a small amount of adversarial inputs to induce a large amount of work in the target system, pushing the system into consuming all available resources.

Table 1: A summary of targeted ALEX designs and characteristics of the proposed ACAs.

		Data-Node Space ACA	Internal-Node Space ACA	Time ACA
Exploited Design Choice	Memory Over-provisioning	✓		
	ML Prediction Error Mitigation		✓	✓
	Keyspace Partitioning		✓	
Targeted Component	Internal Node		✓	
	Data Node	✓		✓
Attack Setting	Black-box		✓	✓
	Gray-box	✓		✓
	White-box	✓		✓
Targeted Resource	Memory (space)	✓	✓	
	CPU (time)			✓

The key observation of this paper is that *ALEX's various design choices to handle dynamic workload* (including the gapped array layout, the cost models to guide node resizing and model retraining) *lack worst-case guarantees, which can be exploited by attackers.*

Our Contributions. In this paper, we perform a security analysis on ALEX's design choices and present three new ACAs. To the best of our knowledge, our work is the *first to systematically analyze algorithmic complexity vulnerabilities* of dynamic learned indexes. The three ACAs fall into two categories, which target the space and time complexity aspects of the ALEX index structure.

• The first category of attacks that we introduce concerns ACA that targets the memory space complexity dimension of ALEX. We discover a set of adversarial input cases that cause significant memory consumption increases at both the data node level and internal node level. First, our space ACA on data nodes exploits the design that ALEX uses a gapped array layout with dispersed empty space called gaps for performance to perform a memory space complexity attack. This attack models ALEX's global data node structure as a knapsack problem and uses Multiple-Choice Knapsack (MCK) to generate an optimal insertion plan for maximizing the memory consumption at the data node level. Second, our space ACA on internal nodes issues duplicate keys to trigger ALEX's catastrophic cost mitigation mechanism, where ALEX needs to recursively split a data node with the hope to bring the duplicate-key-induced cost down to below the threshold; the catastrophic outcome of this attack is that, with just a small amount of insertions for duplicate keys, ALEX repeatedly applies recursive-split operations and exponentially increases the size of the internal node's pointer array, which, in turn, exhausts all available memory resources on the host.

• The second category of attacks is time ACAs, wherein we design an attack method aimed at increasing the number of model-retrain operations. This attack generates pathological insertions that consist of consecutive keys to deviate the actual key distribution from the linear model of a set of randomly chosen (under black-box attacks) or deliberately chosen (under white-box or gray-box attacks) data nodes. The outcome of this attack is a dramatically increased runtime performance overhead caused by model retraining.

• Our ACAs expose vulnerabilities that contradict the intended benefits of space efficiency and performance, which are the original design goals of LIS [35], thereby stressing the need for new designs of dynamic learned indexes with worst-case guarantees.

Summary of Our Proposed ACAs. In this work, we propose three different ACAs and summarize their features in Table 1. For each ACA, we explain the exploited ALEX design choices, the targeted components, the attack settings, and the target resources. The details are as follows: (1) Our space ACA on data nodes is designed to target the memory space resource of the target host. It exploits the memory over-provisioning design choice of ALEX to maximize ALEX's memory consumption by expanding the capacity of data nodes. The effectiveness of this ACA depends on the key distribution of the workload and the adversarial budget size. It can be mounted in both white-box and gray-box settings. (2) Our space ACA on internal nodes also targets the memory space resource but with a different approach. This attack aims to exhaust the target host's memory resources, by causing an OOM event. It exploits two design components of ALEX: recursive splitting of a data node and key-space based partitioning that halves the keyspace regardless of how keys are distributed in it. This attack requires the target database to support duplicate keys and can be mounted in a black-box setting. (3) Our time ACA targets the host's CPU resource, to maximize the number of retrains to deteriorate the performance of the index. The effectiveness of this attack can be affected by the workload's key distribution, the read/write ratio, and the adversarial budget size. It can be mounted in all three settings.

Overview of Our Findings. We conduct an extensive set of experiments on real-world datasets. For space ACA on data nodes, we increase the memory usage by up to 31% compared to baseline ALEX with a budget size ranging from 1% to 30% of various workload sizes. In the case of space ACA on internal nodes, our internal-node ACA is capable of depleting the entire memory resources of the host machine with just a few hundred of adversarial insertions. Our time ACAs cause a throughput degradation of ALEX by up to $1,641\times$ with a small adversarial budget (ranging from 2.5% to 10% of the workload size for write-heavy workloads and 0.5% to 2% of the workload size for read-heavy workloads).

2 PRELIMINARIES

2.1 Static Learned Index Structures

Kraska et al. [35] propose to replace the traditional index structures with a hierarchy of learned models that they call *Learned Index Structures* (LIS). The insight of LIS is that the task of locating a key k within a set of linearly ordered keys can be reduced to approximating the probability that a randomly chosen key would take a value less or equal to k, i.e., $Pr(X \leq k) = Rank(k)/n$, where X is the random variable that follows the empirical distribution of the n keys and $Rank(.)$ is a function that takes a key as an input and outputs its relative position among the n keys. This probability is captured by the cumulative distribution function (CDF); therefore, the task of locating a key reduces to *learning the CDF of the sorted key set.* In a CDF plot, the X-axis represents the keys, and the Y-axis represents the rank of the keys. To approximate the entire key distribution, LIS introduces the *Recursive Model Index* (RMI), a tree-structured, multi-stage architecture where models at a higher stage direct the queries to models at a lower stage to fine-tune the precision of the predicted key location.

2.2 ALEX: Dynamic Learned Index Structure

A limitation of LIS [35] is that it only supports lookup operations on read-only data. Thus, it is not applicable to scenarios where write operations are needed. A follow-up system called *ALEX* [19] was the first *dynamic* learned index structure that extends the idea

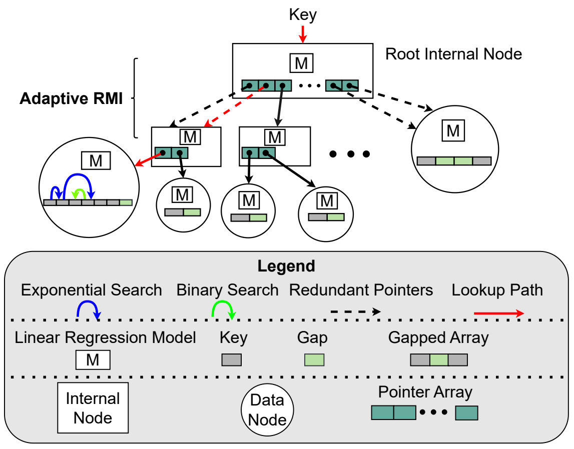


Figure 1: An illustration of ALEX.

of LIS. At ALEX's core is a dynamic, tree-based index structure that consists of, much like the commonly-used B+tree structures [50], a hierarchy of two types of nodes, namely (*i*) *internal nodes* and (*ii*) *data nodes*. Contrary to B+tree structures, each ALEX node is accompanied by a *learning model* (i.e., linear regression) that, as we explain in the following, serves different functionality depending on whether it is a data node or an internal node.

The Role of Internal Nodes. An internal node stores a linear model and a *pointer array* that points to its children nodes. On a high level, the model takes as an input a key and outputs a location of its pointer array. Each entry of the pointer array is a reference to a model from the next level of the tree. We emphasize that all the entries of the pointer array point to child nodes; in case the array has more slots than the number of child nodes, then ALEX allows multiplicities for pointers, also called *redundant pointers* (see Figure 1). The following process generates redundant pointers: Suppose that a node X is split (a mechanism that is described in detail later); if X's parent node has a single pointer in its pointer array associated with X, then the pointer array of X's parent needs to be doubled to accommodate the two new nodes from X's split. All the previous pointers are transferred to the new doubled-size array, and each pointer doubles its entries in the new array; that is if the sibling Y of X had 4 pointers pointing at it, in the new pointer array, there are 8 pointers that point to Y. The relation between the internal node and its children is the following: internal node X is responsible for handling lookups/updates on a partition $[\alpha, \beta]$ of the key space, while each of the children nodes of X is responsible for a non-overlapping (fine-grained) partition of $[\alpha, \beta]$; the role of X is to propagate a lookup/update to the right partition. At the last level of the tree, partitions are associated with a data node that hosts keys that are distributed "roughly" linearly in the associated partition (see Figure 1). In summary, working in conjunction with the data nodes, internal nodes serve two critical functions: (1) Internal nodes direct queries from top to bottom to the right data nodes; (2) Internal nodes enable requests to be progressively directed to fine-grained (and variable size) partitions of the keyspace.

The Role of Data Nodes. A data node stores a linear model and a *gapped array* to store data elements. The empty spaces of the gapped array, i.e., the "gaps", are initially distributed uniformly during the node creation. ALEX relies on these gaps to efficiently absorb new insertions and respond to lookups. The *density* is defined as the ratio between the current number of keys in the gapped array and its total capacity. Each data node maintains the fraction of gaps between a fixed *lower density limit* d_l and a fixed *upper density limit* d_h, which are (by default) set to 0.6 and 0.8, respectively.

Lookups. ALEX locates a key by performing a tree traversal from the top of the tree to the bottom. The query is directed from a parent internal node's linear model to one of its children/nodes until the query reaches a data node, also known as a leaf node in the context of a B+tree. Once the traversal reaches the correct data node, its linear model predicts the key position in the gapped array. During lookups, ALEX returns the key, if found, in the predicted position; otherwise, there is a prediction error and ALEX performs an exponential search on the gapped array to find the key.

Insertions. For insertions, ALEX directly inserts the key into the predicted position; in case an existing key already occupies the predicted position, ALEX shifts existing keys toward the direction of the closest gap to create an empty slot for the new key. ALEX guarantees accurate placement of data by using *model-based insertions* to ensure that: (1) for insertions, the input key is always inserted into the model-predicted position, and (2) for lookups, records are located close to the predicted position when possible.

Adapting to Workloads. As the number of gaps decreases in a data node, the performance of both insertions and lookups degrades. To provide a good average-case performance, ALEX dynamically expands or splits data nodes to adapt to workload changes. ALEX does not wait until a data node becomes 100% full; instead, ALEX *expands* a data node's gapped array when the used space increase beyond the chosen upper-density limit. ALEX uses linear cost models to determine if, after the updates, the trained model of a data node deviates from the true distribution of keys. The cost model uses simple heuristics to compare (1) the expected (modeled) cost computed at the data node creation time to (2) the real-time empirical cost calculated by counting the number of exponential searches (for mispredicted lookups) and memory shifts (for insertions that cause key shifts) that have occurred since data node's creation.

Handling Corner Cases. In addition, ALEX introduces heuristics to handle edge cases when a data node's cost increases beyond a threshold. Such edge cases are deemed as "catastrophic" events that significantly deteriorate ALEX's performance. In these cases, ALEX *splits* the data node affected by a catastrophic event to reduce the cost. ALEX implements two kinds of splits: *splitting sideways* and *splitting downwards*. A sideways split, which is the more common case, directly partitions a data node's key space in half and assign the first half of redundant pointers (if any) to the left child data node and the second half of redundant pointers to the right child data node. If the parent internal node runs out of redundant pointers, ALEX doubles the size of the parent node's pointer array and adjusts its linear model parameters (the intercept and the slope) accordingly. Downwards splits only happen when splitting sideways is no longer possible. That is, the pointer array of the parent internal node is 16MB (can be tuned at initialization).

2.3 Algorithmic Complexity Attacks

In ACAs [14], an attacker uses a small number of adversarial inputs to introduce a disproportional amount of work to the target system, pushing the system into using all available resources. ACAs are tailored to the algorithmic approach at hand and force a worst-case performance for the system. ACAs can be categorized into *time ACA* where an attacker aims to exhaust the CPU resources [5, 36, 47, 49], and *space ACA*, where an attacker aims to exhaust the memory or disk resources [25]. Take the hash-table attacks [14] as a time ACA example. A hash table can degrade to a linked list with carefully chosen inputs whose hash values map to the same bucket. Decompression bombs [28] exploit the capability of efficient compression algorithms to mount a space ACA by decompressing files and quickly consuming a large amount of storage space.

ML for systems is gaining traction and is becoming an increasingly powerful tool for improving modern computer systems [9, 35, 40, 41, 54, 59]. However, incorporating ML into complex computer systems inevitably expands the attack surface, leading to increasing vulnerabilities. Poisoning attacks [8, 21, 64, 65] consider adversaries that deliberately augment the training data to manipulate the results of ML models. Kornaropoulos et al. [33] were the first to demonstrate poisoning attacks (which can be thought of as a time ACA) against the original, static LIS. However, when it comes to dynamic LIS like ALEX, the impact of injecting poisoning keys is not well understood. This is primarily due to ALEX's ability to dynamically adapt its key space partitioning and retrain its models, which helps mitigate the effects of the poisoning attack.

Similar to data structures that are vulnerable to ACAs, ML systems commonly employ ML models that are trained to optimize average-case objectives. However, these ML systems often lack robust guarantees regarding worst-case complexity. As we demonstrate in this work, an attacker can exploit ALEX's worst-case complexity—node splits or expansions—by inserting deliberately chosen inputs to mount space and time ACAs.

2.4 Threat Model

Attacker's Goal. ALEX [19] offers the ability to adjust its shape to a dynamic workload adaptively. In this work, our focus is on adversaries who insert maliciously chosen keys into ALEX to trigger worst-case behaviors, ultimately using all available machine resources. The motives behind such adversarial behaviors vary depending on the potential gains and the context of the application, such as a competitor seeking to degrade performance or a denial-of-service attack. Specifically, in §3.1 and §3.2, our objective for space ACAs is to use all host's memory. In §3.3, the time ACAs aim to degrade the throughput of ALEX.

Attacker's Knowledge. In this work we study white-box, gray-box, and black-box attacks. In the white-box scenario, the attacker has full access to the inner-working of the ALEX under attack, including parameters of the linear regression models and internal structures (data nodes, internal nodes, and keys). White-box attacks [5, 10, 14–16] are a natural first step towards assessing the robustness of a target system since they *quantify the maximum damage*, i.e., act as an upper bound to any future black-box attack, which an adversary can make to a system. Previously proposed ACAs have primarily been developed in the white-box setting. For example, in [14], the adversary has full knowledge of the targeted hash table (or binary tree) in all three scenarios examined, namely ACAs against Perl's hash tables, Squid web proxy, and Bro intrusion detection. Similarly, in [15, 16], the adversary has full knowledge of the control rules of Open vSwitch (OVS) so as to mount an ACA.

Attacker's Capabilities. We assume that the attacker can perform insertions into the target ALEX instance. There are several scenarios where an attacker can insert data to a target DB. First, shared DBs are extensively used in many application scenarios (but not yet using the LIS paradigm). Companies such as Meta deploy database systems that are shared by multiple applications [11], e.g., UDB, ZippyDB, and UP2X. Another example is Google's DB infrastructure, including Bigtable [12], together with the learned index system built atop Bigtable [4], serves many concurrent users and applications. If the above systems used an LIS, malicious users could potentially mount ACAs to target shared DB services. Furthermore, the YCSB benchmark is synthesized based on the workload patterns of real-world, multi-tenant database systems [13]. Second, crowd-sourced databases [23, 32, 42, 46] allow users to contribute their data. In such a setting, an attacker can contribute to these datasets so as to poison/attack the learned index. Even if the attacker does not have access to the exact dataset, it is enough to have knowledge of the underlying distribution in order to devise an approach for generating maliciously crafted keys (e.g., the black-box setting of our space ACA on data nodes). Many real-world applications use crowd-sourced databases, e.g. OpenStreetMap [2] and Amazon's Mechanical Turk (AMT) [1].

Terminology. For space ACAs, we denote the number of adversarial insertions as a and the number of legitimate insertions as l. We test our attacks on different *adversarial budget* scenarios between 5% to 30%, i.e., different parameterization of $100 \cdot (a/(a+l))$. For time ACAs, we use a to denote the number of adversarial insertions, l to denote the number of legitimate insertions, and b to denote the lookup operations on the legitimate keys. In this case, the adversarial budget definition changes to $100 \cdot (a/(a+l+b))$. We evaluate the effect of adversarial insertions under two workloads: (1) write-heavy and (2) read-heavy. The write-heavy workload consists of 50% inserts and 50% lookups, while the read-heavy workload has a ratio of 10% inserts and 90% lookups.

3 ALGORITHMIC COMPLEXITY ATTACKS AGAINST ALEX INDEX STRUCTURE

In this section, we present three ACAs that aim to manipulate the resource utilization of the ALEX. Two of these attacks focus on memory consumption, while the third targets CPU usage. We begin by introducing a space ACA that focuses on data nodes in §3.1. Next, in §3.2, we present a space ACA that exploits the structure of *internal nodes* of ALEX. Finally, in §3.3, we present a time ACA that degrades the throughput of ALEX. Our attack exposition is organized as follows: (1) presenting the ALEX design choice that the attack relies on, (2) the attack method, and (3) evaluation.

3.1 Space ACA on Data Nodes

In this subsection, we propose our first space ACA, which targets the memory utilization of the host system through exploiting the data node over-provision logic. Our experiments show that our attack uses up to 30% more memory than ALEX [19] compared to using legitimate keys. We tested the attack on all four datasets used in the original ALEX paper [19] and reported the results.

3.1.1 Exploitable Design Choice. ALEX relies on *model-based insertion* (see "Insertions" paragraph in §2.2) to decide a memory location for an inserted key. The driving force of model-based insertion is the gapped array, a data structure that balances the efficiency/fragmentation trade-off in this dynamic setting.

Trade-offs in Gapped Array. The gapped array prioritizes the speed of insertion over memory usage. Specifically, due to the model-based insertion, the gapped array guarantees efficient insertion with an $O(logN)$ time complexity (see [19] for the analysis). However, the way of inserting (i.e., forcefully placing the key in the predicted location by shifting existing keys to make space) introduces long consecutive segments of allocated keys. As a result, in case of a lookup of a shifted key, ALEX's original prediction is wrong, which triggers a "corrective" search mechanism that locates the shifted key. To compensate for slower lookups caused by long consecutive segments, ALEX *preemptively migrates* to a larger gapped array before it reaches 100% capacity. Specifically, ALEX introduces lower and upper density limits, $d_l = 0.6$ and $d_h = 0.8$, for each gapped array. A data node is considered *full* if it reaches the upper limit d_h. Once a data node is full, ALEX performs either a node expansion or a node split, depending on the calculation of the data node's cost model (§3.2). After the above action, ALEX increases the capacity of the new data node(s) to $\frac{\#\text{ records}}{d_l}$.

On Exploiting the Over-Provisioning Logic. Notice that given the described over-provisioning approach, the empty space in a gapped can be at most 40% (which occurs when being at d_l capacity, i.e., right after a split/expansion) and at least 20% (which occurs when being at d_h capacity, i.e., before a split/expansion). We make the following observation: If the attacker constructs an insertion sequence that pushes a gapped array to $d_h = 0.8$ density, then the very next insertion to this data node will trigger a split/expansion which will initialize the new gapped with 40% empty space. We study the problem of how to craft such insertion sequences and maximize the memory allocation by exploiting the over-provisioning logic. Interestingly, since each split/expansion increases the empty space *proportionally* to the data node's relative size, we can see that some data nodes may give us more empty space than others. Thus, the attacker has to choose which data node to target first.

3.1.2 Attack Method. An attacker's goal is to maximize the memory usage of ALEX, given a limited budget of adversarially chosen keys. A greedy strategy is to process the data nodes in descending order of size, and add as many keys as needed to each processed node so as to cause a split/expansion, continuing until the budget is exhausted. However, such a greedy approach is not optimal.

Why the Greedy Approach Does Not Work? Consider the case where the largest data node requires X keys to reach d_h density and perform a split/expansion. Let Y be the number of keys corresponding to the available budget, if Y is smaller than X, then the greedy approach will "waste" the entire budget without causing a single split/expansion. This example signals that a different algorithmic approach is needed to optimaly allocate the available budget.

Towards a Knapsack Formulation. In the following, we detail how the attacker's choice of which data node to target to use the available budget better can be seen as a knapsack problem (KP). Knapsack is a fundamental problem in combinatorial optimization with numerous applications to resource allocation scenarios. In KP, there are n items, each of which with value v_i and weight w_i, and the objective is to choose a subset S of items so that their value is as high as possible while their overall weight is less than a given upper-bound W. In more technical terms, in this work, we are interested in the *0-1 Knapsack Problem* where each item i is associated with a variable $x_i \in \{0, 1\}$ which takes value 1 if the i-th object is chosen as part of S and 0 otherwise.

Towards mapping the above terminology to our space ACA, a data node can be seen as an item. Let $i \in [1, n]$ be an indexing of the data nodes. The "weight" of a data node corresponds to the number of keys k_i that are required to be inserted until the next expansion/split. The "value" of the data node corresponds to the free space (in bytes) f_i of the data node's gapped array after the next expansion/split. Finally, the "budget" is the number of keys B the attacker can insert. In this formulation, the goal is to pick a subset S of data nodes, denoted by assigning $\{x_i = 1\}_{i \in S}$ and $\{x_i = 0\}_{i \notin S}$, so as to maximize the objective function $\sum_{i \in [1,n]} x_i \cdot f_i$ while satisfying the constraint $\sum_{i \in [1,n]} x_i \cdot k_i < B$ and $x_i \in \{0, 1\}$. Notice that the solution from the above formulation tells the attacker *how many* keys need to be allocated to each data node but not *which* keys to insert. Recall that each data node is responsible for a consecutive range of the key domain. Therefore, when the attacker wants to generate k_i keys to be inserted into a data node, (s)he can simply choose uniformly at random among the corresponding range.

Extending to Multiple-Choice Knapsack. The above formulation of 0-1 KP is a good start but it is missing a large number of potential strategies for the attacker. Specifically, once the i-th data node is chosen (by assigning $x_i = 1$), the aforementioned formulation gives no option to consider expanding the same data node again. Thus, in case of an instance where optimal space ACA is given by extending/splitting the same data node multiple times, the above formulation will give a suboptimal solution.

In the following, we present how to extend our KP formulation to account for such scenarios. Multiple-Choice Knapsack (MCK) is a variant of KP where the items are subdivided into m classes. MCK has all the previously described constraints of KP plus an additional constraint that ensures that exactly one item is chosen from each class of the m classes. The main insight for extending our formulation is to consider each data node as a class of its own and generate m distinct scenarios. In each scenario of a data node, the inserted keys cause a different number of expansions/splits. Thus, we *hard-code* additional scenarios, each describing the same data node being targeted multiple times. The constraint that "only one out of m choices is allowed" guarantees that we won't choose the same data node to expand, for example, both once and twice. The last step to reach our final formulation is to introduce a scenario, the $(m + 1)$-th one, that allows a data node not to be chosen at all. To capture the MCK notation, we introduce the sub-index j that takes values from 1 to $m + 1$ and denotes which scenario is chosen for the corresponding data node. Thus, x_{ij} indicates whether the j-th scenario of the i-th data node is chosen, f_{ij} indicates the free space that the attacker generates by targeting the i-th data node to be expanded/split multiple times according to the j-th scenario, and k_{ij} indicates the number of keys that the attacker needs to insert to the i-th data node to be expanded/split multiple times according to the j-th scenario. The complete formulation is the following:

$$\begin{aligned} &\max_{x_{ij}} \sum_{i=1}^{n} \sum_{j=1}^{m+1} x_{ij} \cdot f_{ij} \\ \text{s.t. } &\sum_{i=1}^{n} \sum_{j=1}^{m+1} x_{ij} \cdot k_{ij} \leq B, \\ &\sum_{j=1}^{m+1} x_{ij} = 1, \forall i \in [1, n] \\ &x_{ij} \in \{0, 1\} \end{aligned} \tag{1}$$

White-Box & Gray-Box Extensions. The following details how the MCK formulation applies to white-box and gray-box attacks. In the white-box scenario, the attacker has access to all the data nodes of the target ALEX. Subsequently, the attacker uses an MCK solver to determine an offline plan of which data nodes to target and then proceeds to insert adversarial keys based on the generated plan. In the gray-box scenario, the attacker has only *distributional knowledge* about the target ALEX key distribution. In this case, the attacker generates synthetic data using distributional knowledge and locally constructs a substitute ALEX index. Utilizing this substitute ALEX, the attacker generates a plan using the MCK solver and inserts the resulting sequence of keys to the target ALEX.

Discussion. For simplicity, in this work, we assume a gray-box setting from which the attacker has access to the *key distribution* (modeled using a Kernel Density Estimation technique); but one can instead query the targeted ALEX instance in the black-box setting and *learn an approximation of the key distribution.* After this step, the black-box attack proceeds exactly like the gray-box. That is, it samples the approximated key distribution to build a local substitute model; it then crafts an attack on the local model and applies the adversarial payload to the target model.

3.1.3 *Evaluation.*

We implement the MCK solver using Google's OR-Tools [48]. For the substitute ALEX, we use `scikit-learn`'s Kernel Density Estimation library [3]. The `bandwidth` parameter for KDE takes values {0.5, 1, 1.5} and the `kernel` parameter is `tophat`. (Refer to [3] for detailed information on the above parameters.) We tested our data-node space ACA on an EC2 `m5.4xlarge` VM instance with 16 vCPUs and 64GB memory. We tested the white-box and gray-box attacks five times and measure the mean increase in memory consumption compared to the memory usage under a legitimate workload with the same number of keys (insertions).

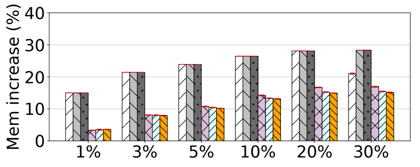
Datasets. We tested four datasets with 8-byte keys: `Longitudes` and `Longlat` use 8-byte `double`-precision floating pointer numbers as keys, while `YCSB` and `Lognormal` use 8-byte `int` as keys. The `Longitude` workload contains keys that represent the longitudes of locations drawn from the OpenStreeMaps public dataset [2]. The `Longlat` workload contains mixed keys that combine the longitude and latitude information from the OpenStreetMap dataset by applying the transformation $k = 180 * \lfloor longitude \rfloor + latitude$ to each pair of the longitude and latitude value. The `YCSB` workload [13] generates keys that represent user IDs uniformed distributed across the whole 64-bit `int` range. The `Lognormal` workload generates keys that follow the log-normal distribution with $\mu = 0$ and $\sigma = 0$.

Tested Methods. We evaluated our attack by comparing its memory consumption to the memory footprint of a normal sequence of insertions. We used different workload sizes (50M keys, 100M keys, and 150M keys). We tested the following scenarios:

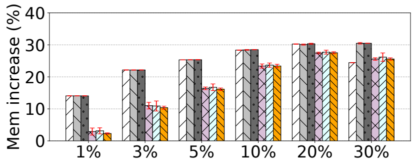
- **Baseline**: We construct an ALEX instance by inserting legitimate keys drawn from the original datasets until reaching a target workload size and measured the memory usage of ALEX. The memory usage of the baseline is not shown in Figure 2 as all bars are normalized with respect to the baseline.
- **White-box Attack**: The white-box attack follows three steps: (1) initialize an ALEX with l keys from legitimate keys of the original dataset; (2) run the MCK solver that takes the following information as input: one array that details how much budget would be needed if one were to trigger E splits/expansions (where the parameter E takes values 0, 2^0, 2^1, and 2^2; thus, the j from Eq. (3.2.1) considers four scenarios) for each of the n data nodes given the current state of the ALEX instance; the other array details the free memory increase from triggering E splits/expansions for each of the n data nodes given the current state of the ALEX instance; and an fixed budget B; given the input, the MCK solver generates the attack plan; (3) perform the sequence of insertions based on the plan generated from the previous step.
- **Gray-box Attack**: The gray-box attack follows five steps: (1) Use a Kernel Density Estimation (KDE) model to approximate the distribution of the original dataset. The KDE model takes as input the original dataset, a "bandwidth" parameter, and a "kernel" parameter; (2) initialize an ALEX with l keys from legitimate keys of the original dataset; (3) construct a substitute ALEX instance by sampling l keys from the learned KDE model; (4) run the MCK solver using the data node information from the instantiated substitute ALEX and budget B; (5) issue the sequence of insertions based on the plan generated from the previous step.

Results. Figure 2 presents normalized memory increases for gray-box and white-box attacks with respect to the baseline. We observe that, across all four datasets for the three workload sizes, both white-box and gray-box attacks result in an increase in memory consumption. Notably, under the white-box setting, the four datasets have a memory increase between 12.76% (for `YCSB`) and 15.32% (for `YCSB`) with only 1% of budget size. The memory gap between white-box attacks and gray-box attacks is particularly striking for `Longitude` and `Longlat`, while this gap is relatively smaller for `YCSB` and `Lognormal`; there is even no difference for `YCSB` 150M. (shown in Figure 2(i)). This is because `Longitude` and `Longlat` are not accurately approximated by the given number of samples to the KDE, which affects the performance of the attack. On the other hand, the other two datasets are much more linear. E.g., `YCSB` dataset is highly linear, which results in ALEX using fewer data nodes to store keys.

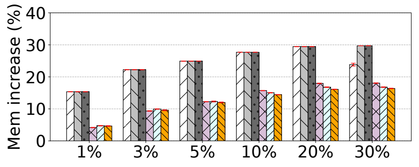
There are no obvious differences for all three expansion parameters (1, 2, and 4) for budget sizes below 30% under white-box attacks. This is because, when the budget is relatively small, one expansion is sufficient to generate an optimal solution. Whereas at least two expansions are needed for 30% in order to maximize memory increases. `white_box_2` and `white_box_4` perform better than `white_box_1` for most of the cases. However, there are rare cases where `white_box_4` performs worse than `white_box_1` and `white_box_2` as shown in Figure 2(d) and Figure 2(e). This is likely due to the computational time constraint we configured for the MCK solver implemented using the OR-Tools: we set a maximum search time of 100 seconds to strike a balance between solution optimality and time efficiency. If either the optimal solution was found within the time limit, or the configured time period elapsed, whichever condition was met first, the MCK solver returned the computed attack plan. We believe that this uncommon event appeared due to the non-linearity nature of



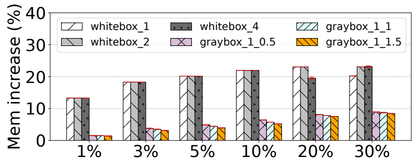
(a) Longitude: 50M keys.



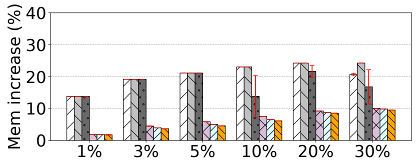
(b) Longitude: 100M keys.



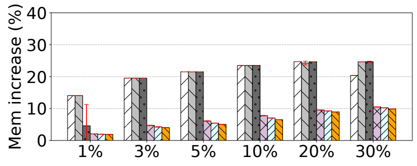
(c) Longitude: 150M keys.



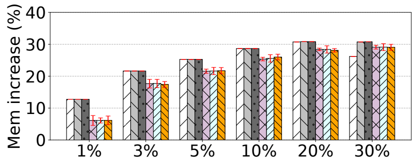
(d) Longlat: 50M keys.



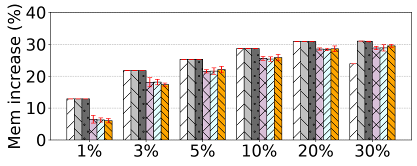
(e) Longlat: 100M keys.



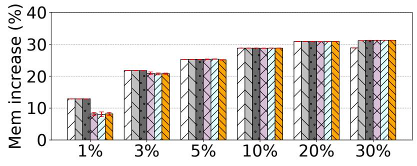
(f) Longlat: 150M keys.



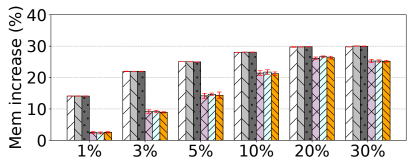
(g) YCSB: 50M keys.



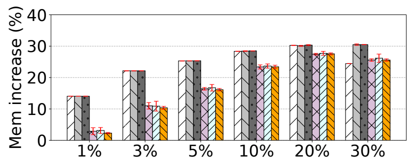
(h) YCSB: 100M keys.



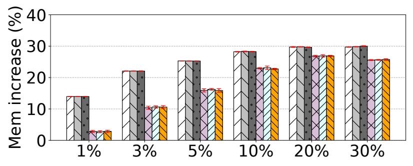
(i) YCSB: 150M keys.



(j) Lognormal: 50M keys.



(k) Lognormal: 100M keys.



(l) Lognormal: 150M keys

Figure 2: Memory increases of space ACA on data nodes. *This figure shows the normalized (with respect to the baseline) memory increase for white-box and gray-box attacks using four datasets and varying ALEX sizes of 50 million, 100 million, and 150 million. The X-axis is the percentage of attacker's budget with respect to the total keys ranging from 1% to 30%; the Y-axis is the normalized memory increase from our attacks.* `whitebox_E` *denotes a white-box attack that allows at most E expansion(s) or split(s) for any data node for the MCK optimization; for* `graybox_E_b`*, E denotes a gray-box attack that allows at most E expansion(s) or split(s) for any data node during the MCK optimization, and b is the "bandwidth" parameter for KDE. Each data point is the mean of five runs. The error bars are depicted in red.*

the `Longlat` dataset. This non-linearity resulted in the creation of a large number of data nodes, which required a longer computational time period to search for the optimal solution. The memory increases under gray-box attacks largely depend on how close the state of the substitute ALEX instance to the target ALEX instance is and the linearity of the original dataset's key distribution.

In summary, white-box attacks achieved 12.76% to 33% memory consumption increases, depending on the configuration of the expansion parameter and the attacker's budget. On the other hand, gray-box attacks are not as close to the damage made by white-box attacks because the substitute ALEX instance of a gray-box setting may deviate from the true ALEX instance.

Potential Mitigation. Instead of using extra "gaps" at data-node level to absorb inserts, one can potentially use a delta buffer to serve inserts. The granularity of the delta buffer could be index-level (e.g., PGM [22]) or node-level (e.g., FITing-Tree [24]) depending on specific goals. These buffers typically are allocated as fixed size and merged within them [22] or with existing data [24]. Once a merge process is triggered, the buffer will be emptied and reuse. Adding delta buffers is an intrusive design choice that might break certain existing properties of ALEX: while eliminating the gapped array design can potentially defend against our space ACA, one potential side effect is the extra memory copy overhead when merging buffered inserts into a node array, which will further affect the performance of ALEX.

3.2 Space ACA on Internal Nodes

In this subsection, we present our internal node ACA that exhausts the memory consumption of the host. This attack exploits multiple design choices of ALEX, which combined lead to a cascading storage increase. We show how the current design of ALEX is problematic when it comes to duplicate keys. We note that supporting duplicate keys is a necessary functionality in index structures.

3.2.1 *Exploitable Design Choice.* Recall that ALEX performs insertions by traversing the tree downwards until it identifies the data node associated with the range to which the input key belongs. Our next attack uses several mechanisms of ALEX's insertion to cause *infinite splits* using only a *single* insertion. To achieve this devastating overhead, the attacker has to insert multiple copies of

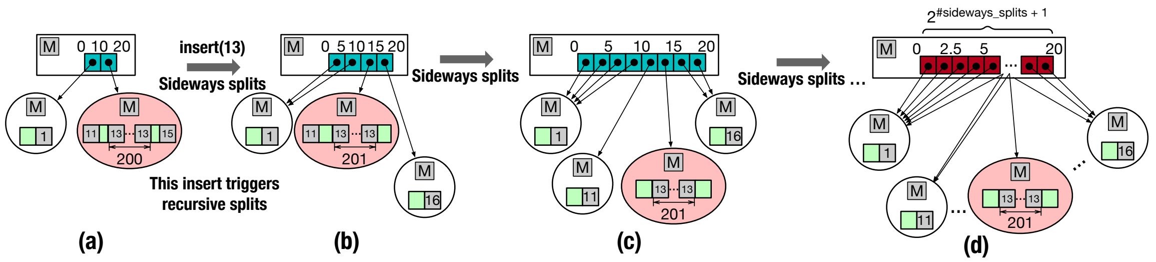


Figure 3: A Space ACA insertion of duplicate key 13 that causes recursive splits and ultimately results in an OOM error. *(a) The right data node (in light red) has accumulated 200 duplicate keys 13, exceeding the cost threshold; (b) Inserting one more key 13 triggers a sideways split; (c-d) Attempting to lower the cost, ALEX recursively (cascadingly) sideways splits; the size of the internal node's pointer array keeps growing exponentially until reaching the maximum internal node size or ultimately causing an OOM; for the former case, ALEX performs a downwards split to create a new internal node at a lower level and then continues sideways splits; this cascading process continues until OOM.*

the same key, i.e., *duplicate keys*. In the following, we unravel the series of events that cause an unbounded number of splits.

Impact of Inserting Duplicates on a Gapped Array. Suppose that a data node already stores a key κ. Let's first analyze the case where κ is inserted again. ALEX traverses the tree and lands on the same data node where the original κ resides. The linear model of the data node points to the entry in the gapped array where the "older" key κ is stored. Since this position is occupied, ALEX shifts the old κ to create a gap for the new κ. When the placement is done, the two κ entries appear consecutively in the gapped array. The above description holds when the attacker inserts yet another copy of κ, in which case, ALEX shifts both of the older κ entries to end up with three consecutive copies. The takeaway message is that after a duplicate key insertion, the following conditions hold: (1) ALEX performs at least as many shifts in the gapped array as the number of identical keys already reside in it, and (2) duplicate keys are allocated consecutively.

Impact of Duplicates on the Cost Model. As a next step, we discuss how conditions (1) and (2) from above affect the cost model of the corresponding data node. The first thing worth pointing out is that periodically, the cost model assesses its accuracy (by calculating the *cost*), and if the model deviates from the current set of keys, the cost is deemed too high, and ALEX triggers the so-called *catastrophic event*. When this occurs, the data node is forced to perform a sideways split (as opposed to an expansion or a downward split). A sideways split of the data node "redistributes" the keys of the problematic data node to two new data nodes in the hope that each new data node will have a much lower cost. Unfortunately, conditions (1) and (2) from the previous paragraph both significantly affect the cost of their data node. The reason is that the number of shifts that have occurred so far is a crucial metric of the cost model; therefore, condition (1) exacerbates the cost on that front. In the same vein, long consecutive segments of keys in the gapped array, i.e., condition (2), increase the number of steps in the exponential search, which is the other crucial metric of the cost model. Thus, the hope is that the sideways split will redistribute the keys and reduce the cost.

What Happens During a Split? When ALEX needs to perform a sideways split, it will split the data node by equally partitioning its key space into half. We emphasize here that the word split does not refer to partitioning the set of keys of the data node in two equal-sized subsets but rather partitioning the *key space* into two key ranges of equal length. This subtle but crucial detail is one of the main reasons the attack works so well. To be more precise, when a data node Z is split, ALEX generates two new data nodes, e.g., Z_L and Z_R, and each of them is responsible for half of Z's key space. As a next step, ALEX relocates the keys of Z that fall under the left key space to Z_L and the keys that fall under the right key space to Z_R. We note here that this partition of keys can be unbalanced (in fact, our attack capitalizes on that possibility). As a final step, ALEX checks the parent of Z, denoted as P_Z, to identify how many entries of P_Z's pointer array are associated with Z. If there is more than one pointer to Z (an instance of the so-called *redundant* pointers from Section 2.2) then half of them are redirected to point to Z_L and the other half to Z_R. The interesting (for our attack) case occurs when P_Z has only a single pointer to Z. In that case, P_Z is *forced to double* the size of its pointer array. Thus, each entry of P_Z's pointer array now occupies two entries in the new pointer array, which means that P_Z can now assign one pointer to Z_L and one to Z_R.

Why Splits Do Not Reduce Cost from Duplicates? The Achilles' heel of this design is their strategy of halving the key space (and redistributing the allocated keys accordingly) as opposed to halving the set of allocated keys directly. To illustrate this point, consider the case where there are hundreds of duplicate copies of κ. Then halving the key space means that all the hundreds of entries will move to one of the brand new data nodes (this is the unbalanced scenario we mentioned before). When a new data node is created, ALEX sequentially re-inserts the corresponding keys one by one to this new node. Unfortunately, this sequence of operations follows the model-based insertion which means that all hundreds of repeated κ entries will re-introduce conditions (1) & (2) in the brand new data node. The last piece of the puzzle is the following observation: if the conditions (1) and (2) in the new data node are severe enough to increase the cost and result in yet another catastrophic event, the newly introduced data node will have a high cost that will trigger another split. If ALEX reaches the above state by a single insertion, then the host will experience a recursive sequence of splits where each of them doubles its parent's pointer array. This cascading effect of an exponentially growing memory allocations will only stop when the host machine runs out of memory.

An illustration of the Attack. Suppose we have a data node responsible for the key space $[0, 20)$. Let's assume, for the sake of the illustration, that the parent node has a pointer array of length two and that the key $\kappa = 13$ has 200 copies that appear in the right data node (see Figure 3(a)). Then, when the attacker inserts key $\kappa = 13$ again, the cost that the cost model calculates increases so much that it triggers a catastrophic event and, consequently, a sideways split. As a result, the old light-red colored data node that was responsible for key space $[10, 20)$ is split to the left (red colored again) data node responsible for key space $[10, 15)$ and a right data node responsible for $[15, 20)$. Given that the parent node only had a single pointer to the old data node, the sideways split triggers a doubling of the parent's pointer array (see Figure 3(b)). The crucial part is that all 201 entries of key 13 moved to the new left data node in Figure 3(b); therefore, the cost of this new data is too high, which triggers yet another split (see Figure 3(c)). This time the key space of the data node with the 201 entries of 13 has key space $[12.5, 15)$, and due to the lack of available pointers on the parent node, another doubling of the pointer array takes place. Figure 3(d) illustrates that the pointer array (colored in dark red) grows exponentially, and all entries of 13 move to a data node with a refined key space partition. Overall, the "batch" of copies of $\kappa = 13$ move to data nodes with more and more fine-grained key space:

$$[10, 20) \rightarrow [10, 15) \rightarrow [12.5, 15) \rightarrow [12.5, 13.75) \rightarrow [12.5, 13.125) \rightarrow [12.8125, 13.125) \\ \rightarrow [12.96875, 13.125) \rightarrow [12.96875, 13.046875) \rightarrow [12.96875, 13.0078125) \rightarrow \dots$$

3.2.2 Attack Method. For the internal node attack, we consider a black-box setup where the attacker has no knowledge of the key distribution of the targeted ALEX structure. To mount the attack, the adversary generates a key that falls anywhere in the key range and inserts the duplicate key to ALEX. Typically, a few hundred duplicate insertions are sufficient o trigger OOM on the host machine. Detecting this attack can be challenging since the attacker can interleave duplicate insertions with legitimate workload traffic, leading to an eventual OOM crash of the ALEX database server.

3.2.3 Comparison with Concurrent Work. In the following, we highlight the differences between our proposed attack and that from a recently released concurrent manuscript by Schuster et al. [51] (referred to as SZEGP). The effectiveness of our approach, compared to SZEGP, is validated through our experiments in §3.2.4, where our attack achieves comparable damage with up to $300,000\times$ fewer keys. SZEGP is parameterized by N, K and works as follows: (1) being a black-box attack, SZEGP randomly samples N keys from the original dataset to roughly estimate the global key distribution; (2) SZEGP picks a random position in the key range and inserts K *consecutive* keys left to the position. If the keys are floating point, then SZEGP generates keys in 10^{-13} increments, if the keys are integers then it generates keys in 1 increments. (3) SZEGP repeats Step#2 until the budget is exhausted or an OOM error is triggered.

Comparison with SZEGP. While our internal-node space ACA and SZEGP share the common goal of doubling the size of parent nodes' pointer array, our attack uses fundamentally different insights to deliver a much more effective attack, as outlined below:

- **Identifying a Data Node**: Our attack can be applied to *any* data node, whereas SZEGP is only effective when applied to the *left-most* data node of a sub-tree. Thus, SZEGP needs to guess a key-area for the cluster and blindly (due to the black-box setting) hit a range of the key distribution with a very specific ALEX substructure. More often than not, SZEGP does not meet the above stringent requirement and as a result "wastes" valuable budget.
- **Effect on Landed Data Node**: Our attack guarantees that *all* inserted keys will end up in the same data node (due to the fact that all keys are identical), which translates to doubling the same pointer array, and thus, delivering a faster OOM. Whereas the consecutive keys inserted by SZEGP may *only partially* land to the left-most data node while the rest spill to a nearby data node.
- **ALEX's Handling of Insertions**: Our attack guarantees that *all* inserted keys will be moved to a single new data node after the split, whereas SZEGP (consecutive) inserted keys may be redistributed to both new data nodes from the split. Consequently, for SZEGP, ALEX may indeed reduce the cost by a sideways split.
- **Requirement on Adversarial Budget**: Our attack operates using a small adversarial budget; typically, a few hundred is sufficient to trigger an OOM event. Conversely, SZEGP relies on adversarial insertions falling into the leftmost data node within a sub tree; to accomplish such an allocation their attacker needs to use a significantly larger budget.

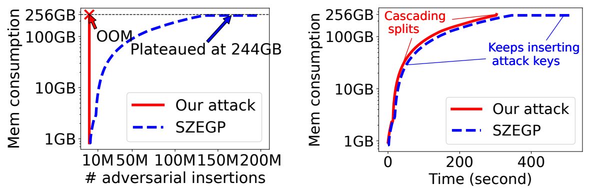
Table 2: Effectiveness of the space ACA on internal node. *In the* OOM *rows,* ✓ *denotes a triggered OOM. Otherwise, the values show the memory consumption after the specified budget is used.*

		Our attack				**Schuster et al.** (SZEGP)			
	Host memory (GB)	**64**	**128**	**256**	**384**	**64**	**128**	**256**	**384**
Longitude	**OOM**	✓	✓	✓	✓	✓	✓	244GB	244GB
	Budget	423	479	472	441	63M	116M	200M	200M
Longlat	**OOM**	✓	✓	✓	✓	✓	✓	✓	✓
	Budget	428	441	437	437	2069	2069	2069	2069
YCSB	**OOM**	✓	✓	✓	✓	✓	✓	✓	✓
	Budget	662	653	692	679	1024	1024	1024	1024
Lognormal	**OOM**	✓	✓	✓	✓	5GB	5GB	5GB	5GB
	Budget	526	589	564	551	200M	200M	200M	200M

3.2.4 Evaluation. We evaluate the internal node attacks on AWS EC2 m5 virtual machine (VM) instances with the RAM configuration ranging from 64GB to 384GB. We use the four datasets from the previous subsection: longitudes, longlat, YCSB, and lognormal.

To compare, we initialized an ALEX instance with 10 million legitimate keys from the dataset using bulk_load(), and then mounted the attack. For SZEGP, we used the same parameter configurations as the ones used in [51], where $N = 1,000$ and $K = 10,000$. We recorded the number of adversarial insertions used and the total amount of memory consumed by ALEX under both types of attacks.

The results presented in Table 2 comprehensively compare our method and SZEGP. Our attack method consistently induced an OOM event on all VM configurations tested across all four datasets, with only 423-692 adversarial insertions. On the other hand, SZEGP triggered OOM in only two out of the four datasets, i.e., Longlat and YCSB, by using $3.83\times$ and $0.55\times$ more budget than our method to exhaust 64GB of host memory. The difference in the effectiveness between the two attack methods is striking in datasets Lognormal and Longitudes, where SZEGP requires more than $10^5\times$ budget and in most cases, it doesn't trigger an OOM. In Figure 4, we take a more detailed view of the comparison between the two methods on Longtitudes. Figure 4 shows the memory consumption as a function of the number of adversarial insertions and runtime on a 256GB EC2 VM. The proposed attack brings the data node to a state where



(a) Memory consumption vs. insertions. (b) Memory consumption vs. runtime.

Figure 4: Memory usage as a function of the number of adversarial insertions for the `longitudes` dataset.

a single insertion will exhaust all available memory regardless of whether the host memory is 1GB or 1TB. Figure 4(a) illustrates this point: no matter how much the defender increases the host's memory, once the trigger insertion takes place all memory is consumed. Our space ACA attack only required 472 insertions of duplicate keys to trigger cascading splits at a memory consumption rate of 0.84 GB/sec until all the 256GB RAM is exhausted (Figure 4(b)). `SZEGP` required contiguous and cumulative adversarial insertions, ultimately reaching a memory usage plateau of around 244GB with $3 \cdot 10^5\times$ more budget. For the `Lognormal` dataset, `SZEGP` used up all the 200 million budget keys but only caused about 5GB memory increase. These results demonstrate the effectiveness of our method.

Potential Mitigation. One may argue that this attack can be mitigated by using a capacity-based partitioning mechanism that produces equally sized data nodes. This defense mechanism could potentially reduce the cost of a data node, as the duplicate keys will eventually be spread across multiple data nodes. However, incorporating capacity-based partitioning will break ALEX's sophisticated design for the following reasons: (1) the ultimate goal of key space partitioning is to make sure that each data node covers a sub-key space whose key distribution is roughly linear; and (2) key space partitioning enables ALEX to use simple linear models for internal nodes. Capacity-based partitioning would have required internal nodes to use expressive, non-linear models as it is highly likely that the key distribution within a child data node is non-linear. Another possible defense is by disabling the option for duplicate keys. However, this will inevitably hurt the versatility and flexibility of ALEX as a core database index structure. In the next section we discover the feasibility of mitigating our attack by changing how ALEX handles catastrophic events.

3.3 Time ACA

In this subsection, we introduce a new category of ACA that aims to deteriorate ALEX's time performance. In §3.2, we introduced a space ACA that exploits the fact that catastrophic events trigger sideways splits. A potential mitigation of our space ACA is the following: in case of a catastrophic event, perform an expansion operation instead of a sideways split. As we show in the following, when expansions happen in the context of a catastrophic event, ALEX needs to retrain the linear regression model of the corresponding data node. However, retraining is a time-consuming operation (a fact that we capitalize on in our time ACA). In this subsection, we analyze a *hypothetical* scenario where ALEX is patched to mitigate our space ACA vulnerability and now performs an expansion instead of a sideways split in case of a catastrophic event[1]

3.3.1 Exploitable Design Choice. Recall that a catastrophic event happens when ALEX detects that the cost of a data node is too high. A reason that contributes to the high cost is that the linear regression model prediction is no longer accurate to support efficient lookups and insertions. While applying an expansion instead of a split thwarts our attack from §3.2, it will cause inaccurate models that result in significant runtime performance cost due to increased training cost. An attacker can exploit this observation of the modified ALEX by strategically inserting sequences of densely clustered keys to carefully chosen data nodes to cause a discrepancy between the key distributions and the linear models of the data nodes.

3.3.2 Attack Method. The goal of an attacker is to leverage the above-described observation to deteriorate the performance of ALEX by increasing its time performance [14, 18, 58]. This will result in: (1) query latency increase when serving highly-concurrent requests, and (2) ultimately, a DoS when the request processing capability offered by the database tier is not able to catch up with the speed of query backlog accumulation. Data node expansion implies that the current model of the data node is no longer accurate, thus a model retraining is required. However, retraining will not fix the inaccuracy issue, simply because the key distribution of the data node *is no longer linear*. Such inaccuracy will cause more exponential searches for lookups and more memory shifts for insertions. More importantly, expanding a data node in ALEX is a time-consuming process due to the overhead of linear model retraining and memory copy. Therefore, an effective adversarial strategy for degrading ALEX's performance is to continuously trigger "catastrophic" expansions (see §2.2) in large data nodes. To this end, we investigate time ACA strategies in three different settings: white-box, gray-box, and black-box.

The white-box setting assumes that the attacker has complete knowledge of the target ALEX, including keys and the global key distribution. In this case, the attacker targets the largest data node. Specifically, the attacker inserts a total number of N attacking keys into the middle position of the largest segment within the largest data node. The attacker spreads the budget of N keys across batches with a size of B, where B is set to 200 and 600 in our tests and all keys within a single batch are consecutive keys with an incremental difference of 10^{-13} for `double`-typed keys and 1 for `int`-typed keys.

In the gray-box setting, the attacker knows the approximate distribution of the keys but does not know the exact keys. With the distribution, the attacker utilizes a kernel density estimation (KDE) [3] method to construct a substitute ALEX structure that approximates the structure of the target (original) ALEX. The substitute ALEX is designed to have the same size as the target ALEX and contains different keys, which fall within the same key range and follow the same key distribution. Subsequently, the attacker devises an attack plan by generating attacking keys targeting the largest data node in the substitute ALEX, same as the strategy used in the white-box attack. The attacker then inserts the N budget keys to the target ALEX in N/B batches during the workload.

[1]We also tested the vulnerability of vanilla ALEX to our time ACA in §3.3.3.

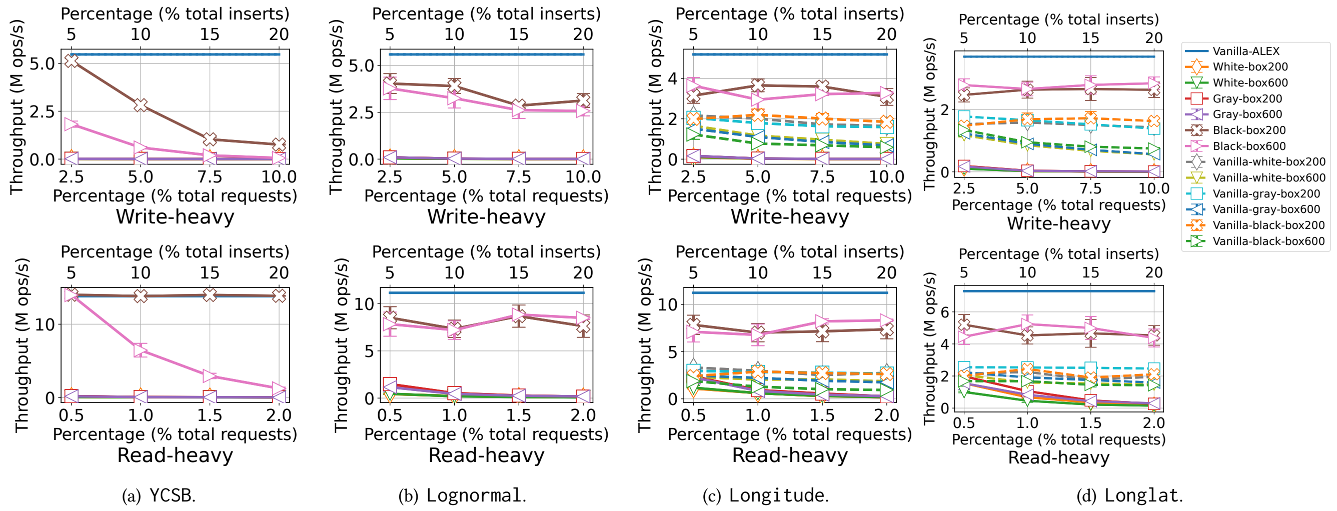


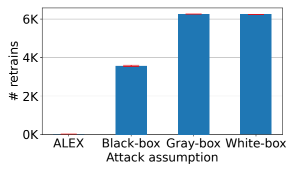
Figure 5: The throughput for different attack settings and percentages of adversarial insertions for vanilla ALEX and modified ALEX. *We varied the percentage of adversarial insertions with respect to (w.r.t.) the total number of inserts from* 5% *to* 20%. *The top X-axis shows the ratios of adversarial insertions to the total number of insertion requests in the workload. The bottom X-axis shows the normalized ratios of adversarial insertions to the total number of requests in the workloads: e.g., a ratio of* 2.5% *for a write-heavy workload w.r.t. the bottom X-axis means, out of the* 10*M requests,* 2.5% *of them are adversarial insertions generated by the attacker; by referring to the top X-axis,* 2.5% *of total requests means* 5% *of total inserts as our write-heavy workload has a* 50% : 50% *read:write ratio.* `Vanilla-ALEX` *is our baseline performance without any adversarial manipulation. To measure the performance of our attacks, we applied them to both vanilla/original ALEX and modified ALEX.* `Vanilla-white-box200` *denotes a white-box attack applied to the vanilla ALEX.* `White-box200` *means a white-box attack mounted against the modified ALEX. The values after the attack setting denote the batch sizes; e.g.,* `White-box200` *means the white-box attack with a batch size of 200. Each data point is the average of five runs with error bars showing the min-max variance.*

Lastly, the black-box setting assumes that the attacker does not know about the key distribution but can probe the existing key range of ALEX via sampling. The attacker then randomly choose N/B locations from within the estimated key range and insert the N/B batches of attacking keys to mount the time ACA.

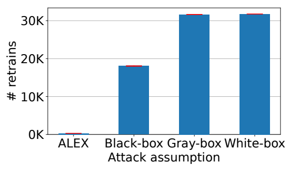
3.3.3 Evaluation. We evaluated the time ACAs on an EC2 `m5` VM instance with 96 vCPUs and 384GB memory. We used the average throughput as the performance metric. We mounted our time ACA against both the vanilla ALEX (the original ALEX) and our modified ALEX (patched to mitigate the space ACA vulnerabilities as discussed at the beginning of §3.3). We conducted the experiments using a write-heavy and a read-heavy workload drawn from the four datasets. The write-heavy workload consists of 50% inserts and 50% lookups, while the read-heavy workload has an insert:lookup ratio of 10% : 90%. Both workloads have a total of 10 million operations. The lookup keys were selected from all existing keys in one of the four datasets, following a Zipfian distribution. Given a dataset, we first initialized an ALEX by using the first 10 million keys via `bulk_load()` and then initiated the 10-million-request workload. We issued a total of 10 batches for all attack settings.

Figure 5 shows the throughput comparisons between a modified ALEX (with expansion-only mode as described in §3.3) and the vanilla ALEX in different attack settings with baseline. For write-heavy workloads, the white-box attacks and gray-box attacks in modified ALEX achieved an average throughput degradation between 31× and 1, 641× across all batch size and budget parameters, compared to the baseline ALEX.

This is because all attack keys were inserted into the largest data node, causing the biggest damage to performance. Black-box attacks, on the other hand, saw relatively fluctuating performance trends for `longitude`, `longlat`, and `lognormal` (Figure 5(c)–5(b)), due to the fact that the ALEX trees constructed under these three datasets had many data nodes. As a result, a single batch of adversarial insertions may end up falling into more than one data node, with reduced likelihood of triggering a "catastrophic" expansion. Interestingly, the performance degradation on `YCSB` increases as both the batch size as well as the attack budget increase, as shown in Figure 5(a), because there were much fewer data nodes used in ALEX to store the linearly distributed keys in the `YCSB` dataset, which increased the chances of a "catastrophic" expansion being triggered and minimized the attack budget wastage. As a result, the throughput of `Black-box600` decreases sharply as the budget ratio increases; with only 2% of requests being adversarial insertions, our black-box-based time ACA caused a performance degradation of up to 11× compared to the baseline case for the read-heavy workload (see the bottom sub-figure of Figure 5(a)), almost approaching the same level of degradation caused by a white-box attack. For attacks against vanilla ALEX, our time ACA achieved 2.1× to 8.8× performance degradation on write-heavy workload and up to 11.99× on read-heavy workload for `longitude` compare to baseline ALEX (note that we were not able to collect the throughput results as our time ACA triggered an OOM event on `YCSB` and `lognormal` against the vanilla ALEX). Figure 6 provides additional insight into the throughput performance drops under different attack assumptions. Both gray-box and white-box approaches incur significant higher number of retrains when compared to ALEX with legitimate workload and the black-box setting. This indicates that the decrease in throughput can be primarily attributed to the number of retrains.



(a) Read-heavy Workload.



(b) Write-heavy Workload.

Figure 6: Number of retrains under various attack assumptions for `Longitudes`. Adversarial insertions are 20% of the total number of insertions. *Each bar in the plot represents the average of five runs with red-colored error bars showing the variance.*

Potential Mitigation. One possible defense is to apply more robust ML models (e.g., polynomial regression) to represent the empirical CDF of keys at data-node level. An important caveat of this mitigation strategy is that applying more expressive ML models deviates from the original motivation of using piece-wise linear regression models to approximate empirical data distribution CDF, and the time overhead of model training and inference might outweigh the robustness these complicated models introduce.

4 RELATED WORK

Other Learned Index Structures. LIS use ML models to replace traditional index structures based on the observation that index structures can be viewed as a CDF [35]. FITING-Tree [24] employs piece-wise linear functions with a predetermined error bound during construction. In terms of dynamism of LIS, a method of minimizing error caused by updating index was proposed by Hadian and Heinis in [26]. The work by Tang et al. [57] studies the distribution of the workload and propose to re-train the model as the queries pattern change. Other works propose learned multi-dimensional index structure [20, 44]. PGM [22] proposes models that utilize an auto-tuned RMI design to optimize space and latency while supporting updates by using an index-level shared write buffer (a strategy similar to the LSM-tree structure [45]). Based on PGM, RadixSpline [31] proposes an RMI that features an alternate linear interpolation based indexing. Lately, a line of works aim to improve dynamic learned index structure in robustness, concurrency, persistence, and the capability to operate under limited DRAM resources [37–39, 61, 62, 66]. Inspired by LIS, SageDB [34] is a database system that adapts to an application through code synthesis and learning techniques. A set of works benchmark the performance of learned index structures [30, 43, 56, 60]. Our work fills a missing gap in the literature by investigating ACAs against emerging dynamic learned index structures [19, 38].

Algorithmic Complexity Attacks. ACAs were initially introduced [14] to exploit worst-case algorithmic design choices in common data structures such as hash tables. Several other studies explored ACAs across diverse applications, such as hash tables [6, 7], regular expression matching [17, 52, 63], automata-based multi-string pattern matching [53], PDF (portable document file) decompression [27], and TCP reassembly [18, 58]. [36, 49] propose solutions that automatically detect inputs that cause algorithmic complexity vulnerabilities. On the defense side, SurgeProtector [5] proposes a general framework to make network functions resilient against ACAs. None of these works explore vulnerabilities of dynamic learned indexes, which we do in this work.

Data Poisoning Attacks. The literature on data poisoning attacks [8, 21, 64, 65] focuses on adversaries who intentionally augment the training data to manipulate the outcomes of predictive models. For instance, Biggio et al. [8] introduced maliciously crafted training data to alter the decision function of support vector machines (SVMs) and increase the test error. Yang et al. [65] proposed gradient-based methods to generate poisoning points for neural networks. Suciu et al. [55] presented a framework to evaluate realistic adversaries conducting poisoning attacks on machine learning algorithms. The study by Jagielski et al. [29] proposes an optimization framework for poisoning attacks on linear regression and introduced a defense mechanism named Trim. The ACAs we propose in this work are different from data poisoning attacks. The objective of poisoning attacks is to maximize the mathematical function that captures the error of the ML model, whereas our ACAs maximize the use of critical resources based on how the system is designed (which cannot always be captured by a simple error function).

Comparison with KRT. The poisoning attacks presented in [33] focus on *static* learned index structures. This means that all inserted keys in [33] are chosen during the initialization. On the contrary, in this work our attacks are performed on a *dynamic* LIS and the keys are chosen based on the current state of the LIS. Another difference is that at the core of the attacks [33] is the observation that injections of keys in cumulative distribution functions cause a cascading error effect. The above error cannot be dealt with at runtime due to the static nature of the LIS. On the contrary, the attacks in this work are centered around the runtime expansion mechanisms when dealing with full-occupancy of a node or increased errors. These mechanisms only appear in the dynamic setting.

5 CONCLUSION

We present a comprehensive study that focuses on the security aspects of an emerging dynamic learned index structure. We propose new ACAs that exploit design choices that balance the trade-off in ALEX's memory usage and runtime efficiency. Our attacks aim to overload the memory and CPU resources. We evaluate the effectiveness of these ACAs through extensive experiments. Our findings have demonstrated that our space ACAs cause out-of-memory with only a few hundred adversarial insertions and our time ACAs lead to a significant degradation in throughput by up to three orders of magnitude, compared to ALEX. Our findings can be used to inform the future generations of robust learned-index-based systems that are not prone to adversarial manipulations.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their valuable feedback and comments. This work was sponsored in part by NSF grants: CCF-2318628, CCF-1919113, OAC-2106446, CNS-2154732, and the Meta Security Research Award.

REFERENCES

[1] 2010. *Amazon Mechanical Turkon.* Retrieved Decmber 10, 2023 from https://www.mturk.com/

[2] 2017. *OpenStreetMap Public Data Set Now Available on AWS.* Retrieved Decmber 10, 2023 from https://aws.amazon.com/about-aws/whats-new/2017/06/openstreetmap-public-data-set-now-available-on-aws/

[3] 2023. *KernelDensity.* Retrieved Decmber 10, 2023 from https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KernelDensity.html

[4] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H. Chi, Lyric Pankaj Doshi, Tim Klas Kraska, Xiaozhou (Steve) Li, Andy Ly, and Chris Olston (Eds.). 2020. *Learned Indexes for a Google-scale Disk-based Database.* https://arxiv.org/pdf/2012.12501.pdf

[5] Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, and Justine Sherry. 2022. SurgeProtector: Mitigating Temporal Algorithmic Complexity Attacks Using Adversarial Scheduling. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 723–738. https://doi.org/10.1145/3544216.3544250

[6] Noa Bar-Yosef and Avishai Wool. 2007. Remote Algorithmic Complexity Attacks against Randomized Hash Tables. In *E-business and Telecommunications - 4th International Conference, ICETE 2007, Barcelona, Spain, July 28-31, 2007, Revised Selected Papers (Communications in Computer and Information Science)*, Joaquim Filipe and Mohammad S. Obaidat (Eds.), Vol. 23. Springer, 162–174. https://doi.org/10.1007/978-3-540-88653-2_12

[7] Udi Ben-Porat, Anat Bremler-Barr, and Hanoch Levy. 2013. Vulnerability of Network Mechanisms to Sophisticated DDoS Attacks. *IEEE Trans. Comput.* 62, 5 (2013), 1031–1043. https://doi.org/10.1109/TC.2012.49

[8] Battista Biggio, Ignazio Pillai, Samuel Rota Bulò, Davide Ariu, Marcello Pelillo, and Fabio Roli. 2013. Is Data Clustering in Adversarial Settings Secure?. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security* (Berlin, Germany) *(AISec '13)*. Association for Computing Machinery, New York, NY, USA, 87–98. https://doi.org/10.1145/2517312.2517321

[9] Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson, William Zhang, Yu Xia, and Andrew Pavlo. 2022. Tastes Great! Less Filling! High Performance and Accurate Training Data Collection for Self-Driving Database Management Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 617–630. https://doi.org/10.1145/3514221.3517845

[10] Xiang Cai, Yuwei Gui, and Rob Johnson. 2009. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *2009 30th IEEE Symposium on Security and Privacy*. 27–41. https://doi.org/10.1109/SP.2009.10

[11] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. USENIX Association, Seattle, WA. https://www.usenix.org/conference/osdi-06/bigtable-distributed-storage-system-structured-data

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[14] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *12th USENIX Security Symposium (USENIX Security 03)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/12th-usenix-security-symposium/denial-service-algorithmic-complexity-attacks

[15] Levente Csikor, Dinil Mon Divakaran, Min Suk Kang, Attila Kőrösi, Balázs Sonkoly, Dávid Haja, Dimitrios P. Pezaros, Stefan Schmid, and Gábor Rétvári. 2019. Tuple Space Explosion: A Denial-of-Service Attack against a Software Packet Classifier. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 292–304. https://doi.org/10.1145/3359989.3365431

[16] Levente Csikor, Vipul Ujawane, and Dinil Mon Divakaran. 2020. On the Feasibility and Enhancement of the Tuple Space Explosion Attack against Open vSwitch. https://arxiv.org/abs/2011.09107.

[17] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 246–256. https://doi.org/10.1145/3236024.3236027

[18] Sarang Dharmapurikar and Vern Paxson. 2005. Robust TCP Stream Reassembly in the Presence of Adversaries. In *14th USENIX Security Symposium (USENIX Security 05)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/14th-usenix-security-symposium/robust-tcp-stream-reassembly-presence-adversaries

[19] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 969–984. https://doi.org/10.1145/3318464.3389711

[20] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (oct 2020), 74–86. https://doi.org/10.14778/3425879.3425880

[21] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. 2020. Local Model Poisoning Attacks to Byzantine-Robust Federated Learning. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1605–1622. https://www.usenix.org/conference/usenixsecurity20/presentation/fang

[22] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[23] Michael J. Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. 2011. CrowdDB: Answering Queries with Crowdsourcing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/1989323.1989331

[24] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1189–1206. https://doi.org/10.1145/3299869.3319860

[25] Gaston H. Gonnet. 1981. Expected Length of the Longest Probe Sequence in Hash Code Searching. *J. ACM* 28, 2 (apr 1981), 289–304. https://doi.org/10.1145/322248.322254

[26] Ali Hadian and Thomas Heinis. 2019. Considerations for Handling Updates in Learned Index Structures. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Amsterdam, Netherlands) *(aiDM '19)*. Association for Computing Machinery, New York, NY, USA, Article 3, 4 pages. https://doi.org/10.1145/3329859.3329874

[27] Nathan Hauke and David Renardy. 2019. Denial of Service with a Fistful of Packets: Exploiting Algorithmic Complexity Vulnerabilities. Retrieved December 10, 2023 from https://www.blackhat.com/us-19/briefings/schedule/#denial-of-service-with-a-fistful-of-packets-exploiting-algorithmic-complexity-vulnerabilities-16445

[28] Adam Jacobson and David Renardy. 2019. *Algorithmic Complexity Vulnerabilities: An Introduction.* Retrieved Decmber 10, 2023 from https://twosixtech.com/algorithmic-complexity-vulnerabilities-an-introduction/

[29] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. 2021. Manipulating Machine Learning: Poisoning Attacks and Countermeasures for Regression Learning. arXiv:1804.00308 [cs.CR]

[30] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. https://arxiv.org/abs/1911.13014.

[31] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned Index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Portland, Oregon) *(aiDM '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/3401071.3401659

[32] Aniket Kittur, Boris Smus, Susheel Khamkar, and Robert E. Kraut. 2011. CrowdForge: Crowdsourcing Complex Work. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology* (Santa Barbara, California, USA) *(UIST '11)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/2047196.2047202

[33] Evgenios M. Kornaropoulos, Silei Ren, and Roberto Tamassia. 2022. The Price of Tailoring the Index to Your Data: Poisoning Attacks on Learned Index Structures. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1331–1344. https://doi.org/10.1145/3514221.3517867

[34] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p117-kraska-cidr19.pdf

[35] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International*

Conference on Management of Data (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[36] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[37] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A Fine-Grained Learned Index Scheme for Scalable and Concurrent Memory Systems. *Proc. VLDB Endow.* 15, 2 (oct 2021), 321–334. https://doi.org/10.14778/3489496.3489512

[38] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *Proc. VLDB Endow.* 15, 3 (nov 2021), 597–610. https://doi.org/10.14778/3494124.3494141

[39] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. 2022. FILM: A Fully Learned Index for Larger-Than-Memory Databases. *Proc. VLDB Endow.* 16, 3 (nov 2022), 561–573. https://doi.org/10.14778/3570690.3570704

[40] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (Atlanta, GA, USA) *(HotNets '16)*. Association for Computing Machinery, New York, NY, USA, 50–56. https://doi.org/10.1145/3005745.3005750

[41] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[42] Adam Marcus, Eugene Wu, David Karger, Samuel Madden, and Robert Miller. 2011. Human-Powered Sorts and Joins. *Proc. VLDB Endow.* 5, 1 (sep 2011), 13–24. https://doi.org/10.14778/2047485.2047487

[43] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (sep 2020), 1–13. https://doi.org/10.14778/3421424.3421425

[44] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 985–1000. https://doi.org/10.1145/3318464.3380579

[45] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (jun 1996), 351–385. https://doi.org/10.1007/s002360050048

[46] Hyunjung Park, Hector Garcia-Molina, Richard Pang, Neoklis Polyzotis, Aditya Parameswaran, and Jennifer Widom. 2012. Deco: A System for Declarative Crowdsourcing. *Proc. VLDB Endow.* 5, 12 (aug 2012), 1990–1993. https://doi.org/10.14778/2367502.2367555

[47] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. 2018. Automated Synthesis of Adversarial Workloads for Network Functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 372–385. https://doi.org/10.1145/3230543.3230573

[48] Laurent Perron and Vincent Furnon. 2023. *OR-Tools*. Google. Retrieved Decmber 10, 2023 from https://developers.google.com/optimization/cp/cp_solver/

[49] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2155–2168. https://doi.org/10.1145/3133956.3134073

[50] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems* (3 ed.). McGraw-Hill, Inc., USA.

[51] Roei Schuster, Jin Peng Zhou, Thorsten Eisenhofer, Paul Grubbs, and Nicolas Papernot. 2023. *Learned Systems Security.*

[52] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: Crafting Regular Expression DoS Attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 225–235. https://doi.org/10.1145/3238147.3238159

[53] Govind Sreekar Shenoy, Jordi Tubella, and Antonio González. 2012. Improving the Resilience of an IDS against Performance Throttling Attacks. In *Security and Privacy in Communication Networks - 8th International ICST Conference, SecureComm 2012, Padua, Italy, September 3-5, 2012. Revised Selected Papers (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering)*, Angelos D. Keromytis and Roberto Di Pietro (Eds.), Vol. 106. Springer, 167–184. https://doi.org/10.1007/978-3-642-36883-7_11

[54] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. 2020. Learning Relaxed Belady for Content Distribution Network Caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 529–544. https://www.usenix.org/conference/nsdi20/presentation/song

[55] Octavian Suciu, Radu Marginean, Yigitcan Kaya, Hal Daume III, and Tudor Dumitras. 2018. When Does Machine Learning FAIL? Generalized Transferability for Evasion and Poisoning Attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1299–1316. https://www.usenix.org/conference/usenixsecurity18/presentation/suciu

[56] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 16, 8 (jun 2023), 1992–2004. https://doi.org/10.14778/3594512.3594528

[57] Chuzhe Tang, Zhiyuan Dong, Minjie Wang, Zhaoguo Wang, and Haibo Chen. 2019. Learned Indexes for Dynamic Workloads. https://arxiv.org/abs/1902.00655.

[58] Juha-Matti Tilli. 2018. *CVE-2018-5390: Linux Kernel TCP Reassembly Algorithm Lets Remote Users Consume Excessive CPU Resources on the Target System.* Retrieved Decmber 10, 2023 from https://ubuntu.com/security/cve-2018-5390

[59] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[60] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *Proc. VLDB Endow.* 15, 11 (jul 2022), 3004–3017. https://doi.org/10.14778/3551793.3551848

[61] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proc. VLDB Endow.* 14, 8 (apr 2021), 1276–1288. https://doi.org/10.14778/3457390.3457393

[62] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: Robust Learned Index via Distribution Transformation. *Proc. VLDB Endow.* 15, 10 (jun 2022), 2188–2200. https://doi.org/10.14778/3547305.3547322

[63] Valentin Wüstholz, Oswaldo Olivo, Marijn J. Heule, and Isil Dillig. 2017. Static Detection of DoS Vulnerabilities in Programs That Use Regular Expressions. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*. Springer-Verlag, Berlin, Heidelberg, 3–20. https://doi.org/10.1007/978-3-662-54580-5_1

[64] Huang Xiao, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli. 2015. Is Feature Selection Secure against Training Data Poisoning?. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37* (Lille, France) *(ICML'15)*. JMLR.org, 1689–1698.

[65] Chaofei Yang, Qing Wu, Hai Li, and Yiran Chen. 2017. *Generative Poisoning Attack Method Against Neural Networks.* Retrieved Decmber 10, 2023 from https://arxiv.org/abs/1703.01340

[66] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, Guoan Wu, and Andy Rudoff. 2022. PLIN: A Persistent Learned Index for Non-Volatile Memory with High Performance and Instant Recovery. *Proc. VLDB Endow.* 16, 2 (oct 2022), 243–255. https://doi.org/10.14778/3565816.3565826