Master Capstone Project: Locker 2.0 - Oblivious Computation for the etcd Key-Value Datastore

Ismail Ahmed, *Graduate Student, The University of California, Santa Cruz*

***Abstract*—This Master Capstone Project Report applies various oblivious data structures to the etcd key-value datastore to provide private and secure client-server computation using oblivious computation. The subfield of oblivious computation is a hot topic in cybersecurity, as it was invented in the late 1980s and was revived in the early 2010s, as indicated by the PathORAM research paper. After PathORAM made oblivious data structures' implementation and performance viable for real-world use, researchers worldwide have recently been applying oblivious computation to cloud computing, as the previous standard of end-to-end encryption is nowadays insufficient to preserve client security and privacy and prevent adversarial attacks from servers and third parties, causing massive legal liability concerns for server operators. This paper applies three recent oblivious data structures (PathORAM, vORAM+HIRB, and EMM) to an etcd client protocol with comprehensive testing, verification, and benchmarking.**

***Index Terms*—oblivious computation, oblivious data structures, ODS, oblivious RAM, ORAM, vORAM, access patterns, HIRB, vORAM+HIRB, secure deletion, encrypted multi-map, EMM, searchable encryption, structured encryption, searchable symmetric encryption, provable security, privacy, storage.**

I. Introduction

PRIVACY and security have been a concern for human beings since the development of human civilizations [1]. Concealed messages have been used since ancient times and are frequently documented in historical records. Privacy and security have changed dramatically with the advent of electronic computers, which process information at a highly rapid speed and scale [2]. This led to the development of the scientific subfield of cybersecurity, as it underpins all modern electronics, computers, and electronically controlled devices. In other words, cybersecurity is an essential component of contemporary life, as without it, no sensitive computation could reasonably occur, and computers could not be used in any fields dealing with sensitive information [3].

The first modern cybersecurity subfield was end-to-end encryption, starting with the Diffie-Hellman key exchange algorithm in 1976 [4]. The end-to-encryption paradigm was quite successful, as it enabled almost perfect security until the early 2010s, when quantum computers were built (as future powerful quantum computers could easily break key exchange algorithms with Shor's Algorithm), and inference attacks were developed, as described in Islam et al. and Grubbs et al, [5] [6]. This forced the classic end-to-end encryption-based security approach to be supplemented by stronger methods.

Also, around the early 2010s, an old subfield of cybersecurity was rediscovered: oblivious computation. Goldreich and Ostrovsky founded the subfield of oblivious computation in the late 1980s, which was initially intended to secure commercial software against piracy [7]. They developed the oblivious random access memory (ORAM) data structure, in which all accessed data is continuously shuffled and encrypted to obscure data access patterns. ORAM ensures that an eavesdropping adversary cannot detect any pattern to the client's ORAM accesses, and thus the privacy and security of the client's computation are preserved. However, the performance of Goldreich and Ostrovsky's ORAM was impractical, especially on the far less powerful computers of their day.

The first breakthrough in modern oblivious computation was by Emil Stefanov et al. in 2012, who invented a simple yet far more performant ORAM data structure, called PathORAM, by using a novel binary search tree design that limited its worst-case overhead to $O(\log^2 n)$ [8]. That made it practical to use in computer software and hardware, and cloud computing environments were no exception [9]. Due to this breakthrough, many researchers have worked in this subfield, as they have seen the potential to redefine modern cybersecurity [10].

In 2016, Roche et al. followed up by developing the vORAM+HIRB data structure, with vORAM defining novel ways to interact with a generic ORAM efficiently and HIRB being an efficient binary tree [11]. The authors integrated vORAM and HIRB to form the vORAM+HIRB oblivious map/dictionary data structure. Its performance was similar to the original PathORAM with some additional metadata and client-side overhead, still being $O(\log^2 n)$, thus making it just as practical as the original PathORAM. PathORAM could be used as the generic ORAM for vORAM+HIRB.

In 2022, Alexandra Boldyreval and Tianxin Tang created the Encrypted Multi-Map (EMM) data structure, which allowed for no leakage of data access patterns, thus completely preventing all adversarial inference and frequency analysis attacks, assuming trustworthy client software and hardware (although side-channel attacks on the client are still a potent threat) [12]. This guarantees high security for all client-server communication, especially in a cloud computing environment. The authors accomplished these impressive results using PathORAM as a generic ORAM and any oblivious map/dictionary as a generic oblivious map/dictionary. Its remarkable performance was $O(m \log^4 n)$, thus paying a reasonable cost for such strong privacy and security guarantees.

Using a fictional but realistic example, let us demonstrate a common cybersecurity problem in modern computing. Suppose an electronic health record (EHR) company stores sensitive financial and medical data collected from doctors' offices to provide easy and convenient access to sensitive information, including for high-profile individuals. As is common

nowadays, the EHR company stores its data as a database for greater performance, scalability, and more straightforward data computation. They store the database and perform all computations online with a third-party cloud provider to allow consistent and fast access for the EHR client. Fearful of the obvious security risks of storing sensitive data in plaintext, the EHR company encrypts its database with proper end-to-end encryption using a symmetric key-exchange algorithm such as RSA. However, the EHR company was shocked to discover that, one day, the sensitive information of all users was leaked for sale by cybercriminals and bought by tabloids, who publicized the sensitive personal information of high-profile individuals. Infuriated by the public exposure of sensitive information that caused them great embarrassment and lowered their reputations, high-profile individuals who had their privacy violated sued the EHR company for legal liability, won the legal case, and the EHR company paid an enormous amount of legal fees to their lawyers and damages to the high-profile individuals. This negative publicity causes doctors' offices to stop using the EHR company due to patient demands and legal liability reasons, which forces the EHR company to cease operation due to a lack of revenue. How was this accomplished, despite using "modern security techniques"? The answer lies in the lack of oblivious computation. As the third-party cloud server provider, in any reasonable cybersecurity model for sensitive information, obviously cannot be trusted, the weak link in the end-to-end security model of the EHR was that the data access patterns were leaked to the server or to an eavesdropping adversary, which allowed the attacker to perform an inference/frequency analysis attack to compromise the security and privacy of the EHR's clients. This example demonstrates that we need better network, cloud, and database security methods, as the state-of-the-art end-to-end encryption for databases stored on a server is insufficient to prevent inference/frequency analysis attacks.

Therefore, we proposed that we integrate PathORAM, vORAM+HIRB, and EMM and apply them to a plaintext etcd client-server communication protocol in such a fashion that all communication will be secured entirely from all non-client adversaries with all known attacking techniques, including malicious server and network eavesdropper attackers, assuming the client is to be trusted. This would guarantee that the attacks in the fictional example will never occur, as the untrusted client-server network connection and etcd server could never perform an inference/frequency analysis attack, as the client will not leak any data access patterns. However, it is still possible that an attacker could compromise the client itself. Thus, the privacy and security of all safe clients would be safeguarded, and the legal liability of the server operator would be reduced (as all remaining attacks target the client, which is not the operator's responsibility). This effort is called Locker 2.0.

II. Related Work

We have chosen the three oblivious data structures (ODS) used in Locker 2.0: PathORAM as a generic ORAM, vORAM+HIRB as a generic oblivious map/dictionary, and EMM as a generic oblivious multi-map. EMM utilizes vORAM+HIRB while vORAM+HIRB uses PathORAM to function, so we have included a detailed explanation of all three oblivious data structures. We will cover PathORAM, vORAM, HIRB, vORAM+HIRB, EMM, and ORAM system designs in that order as separate Subsections.

A. PathORAM

PathORAM is a variant of TreeORAM, which was the first binary-tree-based ORAM data structure when it was published in 2011. However, the cybersecurity community did not utilize TreeORAM much, as TreeORAM is quite complicated and inefficient compared to later ORAM data structures. PathORAM was the first modern ORAM adopted by the cybersecurity community, as it was much more efficient and easier to understand and implement. It is still the current consensus ORAM data structure, as many later research papers either explicitly utilize PathORAM or allow any generic ORAM data structure to be used, including PathORAM. It disguises all data access patterns from all network eavesdropping adversaries and malicious servers, preventing all inference/frequency analysis attacks. However, it is still vulnerable to a side-channel attack on the client.

PathORAM, like TreeORAM, has a classic binary tree of N leaf nodes storing a bucket of Z blocks and log_N levels ($Z = 4$ is typically optimal). For data access, a recursive position map is created and stored on the server to fetch data access paths to/from the binary tree using the leaf identifier (lid) metadata on each block. The client-to-server eviction process occurs after every data access. By these methods, the untrusted server cannot distinguish whether a read or write operation has happened, as it can only observe that the client fetches a consistent data path and puts it back onto the server with fresh randomness. Deleting data involves writing an empty value, while updating data consists of writing a new value to its existing memory location.

However, that is where the similarities between PathORAM and TreeORAM end. PathORAM creates a local client storage space stash that will tell the client to fetch a whole path from the server and write to the client's stash, bound to $O(\log N)\omega(1)$, for data accesses and re-assigning the target block with a randomly generated leaf identifier, thus leaving the equivalent server's tree-path empty. This method forces the client to read/write all data locally in the stash whenever possible. However, there are cases when blocks must be evicted from the client's stash to the server's binary tree, which is conducted greedily from the bottom of the empty path to the top by iterating over the client's stash and filling the server's tree with blocks from the stash. This greedy eviction process allows all buckets to keep the Z storage requirement constant while ensuring that the client's stash overflowing would be a negligible prospect. PathORAM takes $O(\log^2 N)$ time for the greedy eviction process and $O(\log N)$ time for the recursive position map operations, thus making PathORAM take $O(\log^3 N)$ time with a $O(\log^2 N)$ network bandwidth. The client's stash is bounded by $O(\log N)$, and when $Z = 4$ (as recommended), the server requires $O(4N)$ space.

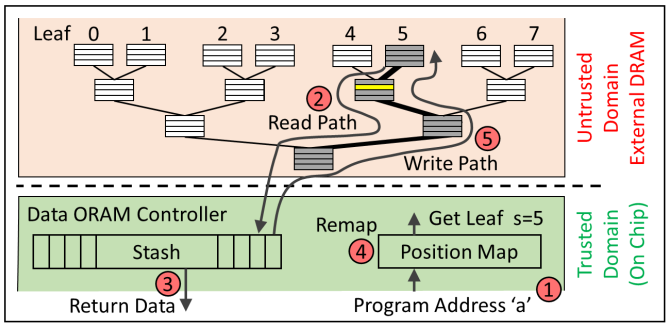


Fig. 1. Marten van Dijk's, one of the original authors of the PathORAM paper, illustration of the high-level PathORAM system design.

B. vORAM

To support an oblivious dictionary/map, it is necessary to modify a generic ORAM data structure, called vORAM, to allow for varying block sizes without trivial padding and to create a history-independent data structure, called HIRB, that uses vORAM as a subcomponent. vORAM and HIRB are then integrated to form an oblivious data structure for a key-value map/dictionary to support the creation, reading, updating, and deleting (CRUD) data operations while preserving the history of all old operations. All client operations (including their data access patterns) and the client's history of old operations are entirely hidden from any malicious server or network eavesdropping adversary, which prevents them from performing any inference/frequency analysis attacks. These methods provide a high security standard.

The vORAM construction of a generic ORAM data structure aims to hide the size of varying-sized items from all network eavesdropping adversaries and malicious adversaries by utilizing a more efficient method than trivial padding. vORAM is an extension of ORAM with variable-sized blocks and was implemented in the research paper using PathORAM with a fixed bucket size ($Z = 4$ as described in the PathORAM Subsection), but with each bucket allowed the freedom to contain as many variable-sized blocks/partial blocks as the bucket space allows. vORAM also permits blocks to be stored across multiple buckets in the same path. These alterations do not affect the time and space complexity of PathORAM while allowing HIRB to be implemented, thus paving the ground for the vORAM+HIRB data structure.

C. HIRB

The History-Independent Randomized B-tree, shortened to the HIRB tree, is a variant of the classical B-tree data structure that supports all the features necessary for oblivious computation. In short, the HIRB tree must be a bounded-height tree-based data structure that supports unique representation, which is needed for security and performance reasons. The essential features of the HIRB tree are listed:

1) The HIRB tree must have a unique representation such that the pointer contents and structure are defined by some initial randomization and the set of $(label, value)$ pairs stored within. This property is needed to guarantee strong historical independence for security reasons.
2) The HIRB tree must support the easy partitioning of blocks with a geometrically-bounded vORAM storage bound for the block's expected size. This property is needed for performance and implementation reasons.
3) The HIRB tree's memory access patterns must be bounded by the chosen ORAM's fixed parameter. This property is needed to support all ORAMs.
4) The HIRB tree must support pointers, and the structure of blocks and pointers must be a valid arborescence with at most one pointer per block. This property is needed to support non-recursive ORAMs.

The HIRB tree has the same functionality as a B-Skip List that forms a top-down tree, removes pointers between skip nodes, and sorts labels using a hash function. It is a B-tree, with a fixed-height at $H = \log_\beta n + 1$, defined in such a way that the level for a newly-inserted item is computed by $\log_\beta n$ trials of pseudorandom-biased coin flipping, where β is the expected blocking factor for the initialized blocking factor B. The HIRB tree allows for merges and splits, but rotations are prohibited. These constraints ensure that every operation causes at most $2H$ nodes to be visited, thus leading to performance gains compared to a standard AVL tree or B-Skip List.

An usual HIRB tree has large nodes with a branching factor of k within the range $[B+1, 2B]$ according to some B-tree parameters $B \geq 1, k$, with a k-node having $k-1$ values, $k-1$ labels, and k children. Using these terms as standard terminology, let us define a HIRB tree with its unique initialization parameters: the height H and the expected branching factor β (contrasted to the standard B-Skip list's branching factor of B). The other non-initialization parameters are the length of hash digests γ, the length of the hash function $|Hash(label)| = max(2H \lg \beta + \gamma, \lambda)$ for some $\lambda \in \{0, 1\}$, and the maximum number of distinct labels n. A HIRB tree node i with a branching factor of k has size $nodesize_i = (k+1)(2T+\gamma+1)+k(|Hash(label)|+|value|)$, $k-1$ values, k vORAM identifiers for child nodes' pointers with size $2T+\gamma+1$, and $k-1$ label hashes. The height must always be $H \geq log_\beta n$ to prevent root nodes from growing too much. For best results with a node i, β must be computed by solving $a * nodesize_\beta \leq Z$, where $6 \leq a \leq 20$ and Z is the size of the vORAM bucket.

HIRB trees' initialized branching factor B has a relaxed range of $k \in [1, \infty]$ within a geometric distribution for the ORAM's storage, defined as a random variable X drawn independently from a geometric distribution with expected value β as a B-tree parameter. HIRB trees define the height of a node as the path length from itself to the leaf node (as all nodes share the same distance to the root node in a B-tree, thus making the distance from a node to the root node useless as a path length metric). The height of a newly inserted $(label, value)$ node is calculated by a series of pseudorandom-biased coin flips according to the label's hash, which guarantees that the distribution of selected heights for insertions uniquely

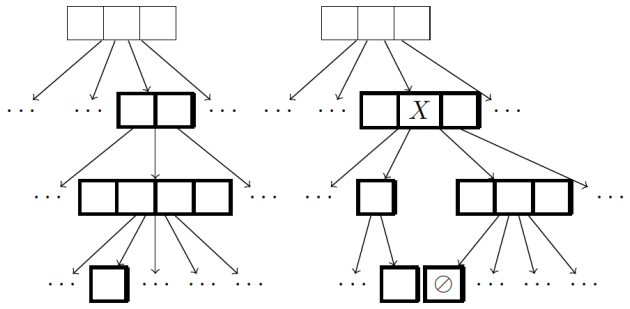


Fig. 2. The authors of the vORAM+HIRB paper's illustration of the insertion/deletion process inside a HIRB tree. The original caption was: "HIRB insertion/deletion of $X = (Hash(label), value)$: On the left is the HIRB without item X, displaying only the nodes along the search path for X, and on the right is the state of the HIRB with X inserted. Observe that the insertion operation (left to right) involves splitting the nodes below X in the HIRB, and the deletion operation (right to left) involves merging the nodes below X."

determines the HIRB tree's structure by its deterministic process (due to the usage of a pseudorandom-biased coin flip and not a truly random one). Thus, the HIRB tree supports the property of unique representation.

For all HIRB trees, initially, an empty HIRB tree of height H is created. Inserting/removing an element onto the HIRB tree requires the respective splitting/merging of nodes along the top-down (from the height of the item down to the leaf node) search path to the chosen element, with insertions/deletions performed at the selected label hash height inside the tree. Setting/deleting an item onto the HIRB tree requires calculating the given label's height with sampling from a geometric distribution with probability $\frac{\beta-1}{\beta}$ and then derandomizing, utilizing a pseudorandom number sequence based on the result of $Hash(label)$ for the derandomization. To support oblivious computation, every operation mandates reading exactly $2H + 1$ nodes by padding with "dummy" accesses as needed, so any operation cannot be distinguished from all others, thus hiding all data access patterns from all network eavesdropper adversaries and malicious server (although the risk of side-channel attacks is still present).

We must use a secure pseudorandom generator, like the SHA-1 software library, for HIRB to be safe. However, various other secure pseudorandom generators exist, such as the Permuted Congruential Generator (PCG) and Linear and Multiplicative with XOR (LXM) pseudorandom generator algorithms [15]. This means that the HIRB tree can only fail with a hardware failure, an incorrect implementation, a faulty/unsafe pseudorandom generator, a hash collision due to too many inserted elements, or a successful side-channel attack.

D. vORAM+HIRB

The vORAM+HIRB construction is relatively simple, as all an implementer needs to do is choose a valid ORAM (like PathORAM), construct the vORAM data structure as a modification of the chosen ORAM, construct the HIRB tree data structure using the vORAM abstraction already built, and expose an external interface for all external libraries to use the vORAM and HIRB together. After all of these steps, the implementation of the vORAM+HIRB constructor is finished.

After a successful implementation, we could guarantee that, under reasonable circumstances, the vORAM+HIRB data structure provides secure deletion, obliviousness, and history independence with a negligible leakage of $O(n + \frac{n\lambda}{\log n})$ data operations. The only reasonable vulnerability it has is side-channel attacks. Therefore, vORAM+HIRB is a very secure and practical oblivious map/dictionary with excellent performance and many convenient guaranteed properties.

E. EMM

The encrypted multi-map data structure, shortened to EMM, is a fundamental data structure intended initially for searchable/structured encryption (the authors claim that EMM is a protocol, but EMM is an oblivious data structure that operates between the trusted client and the untrusted server through an untrusted network connection). It was designed for situations that required robust data security by hiding all data access pattern information, including data and query metadata, access patterns, volume patterns, and query patterns from all non-client adversaries using known attacking techniques, including network eavesdroppers and malicious servers (although side-channel attacks on the client are still a credible threat). It requires a generic ORAM data structure, such as PathORAM, and an encrypted dictionary based on a B-Skip List or AVL tree data structure, such as vORAM+HIRB. EMM supports updates and batching at the cost of some performance, which makes it far more practical than similarly robust security solutions that usually either leak some data access patterns or are static and do not provide the ability to handle data updates. Thus, EMM is well suited for networked database applications, so we chose this data structure as the main one for Locker 2.0.

The authors intended that EMM be used for clients with limited sublinear data storage and the same secret key and sync state connecting to malicious servers with an encrypted database. Notably, the authors allow servers not to use secure trusted execution environments/hardware enclaves (Intel SGX or similar hardware solutions) and to collude against the client maliciously. These assumptions are far more relaxed than some of the current oblivious computation literature's assumptions, which furthers trust in the security and privacy of EMM-based solutions.

The high-level description of EMM is listed:

1) Initialize a modified non-recursive position-based ORAM that behaves like an encrypted array for the encrypted map/dictionary's generic ORAM.
2) Initialize a modified oblivious map/dictionary, using the modified ORAM, that supports the ReadUp and GetUp operations, which will behave like an encrypted map/dictionary.
3) The client assigns each arbitrary label an ORAM index/block identification (block ID) while storing the label's associated value list in the ORAM index's block, padded for volume hiding purposes using the vORAM's technique. The client also stores its secret position map alongside the label-index mapping.
4) Once the client is finished writing all of its pending requests, the encrypted map/dictionary stores the mapping between the ORAM label-index mapping, but not its secret position map, on the server.
5) Once the server is finished computing the client's real and dummy data, the client retrieves the data, filters out its known dummy requests, and obtains its real data.

The generic ORAM they use is modified as a position-based non-recursive ORAM. The oblivious map/dictionary is modified to hide all data access patterns by hiding access, query, and volume metadata with dummy requests. The authors mostly keep the exact implementation details described in PathORAM and vORAM+HIRB. Still, for optimization purposes, they store the label's assigned block identification (ID) and position tag on a server-side encrypted map/dictionary, which is only updated by the client, who is the only one that knows the secret position map. The two special EMM operations are GetUp and ReadUp, which perform a Get then Update operation or a Read then Update operation, respectively, which are combined to improve performance by parallelism.
Overall, EMM is simply integrating an ORAM and an oblivious map/dictionary to ensure robust data security by leaking a negligible amount of data access patterns with two rounds of communication of each $O(\log N_{ram})$ and a constant ORAM access cost of $O(\log N_{ram})$ to form a total cost of $O(\log^2 N_{ram} + \log N_{ram} * l_{mm^\star} * \max_{v \in V_{mm^\star}} *|v|_2) \approx O(m \log^4 n)$, where N_{ram} is the number of RAM requests.

F. Goldreich and Ostrovsky, Oblivious RAM (ORAM)

Oded Goldreich and Rafail Ostrovsky were the two cybersecurity researchers who invented the subfield of oblivious computation. Goldreich was the first to publish a research paper in the subfield in 1987 while Ostrovsky followed up with one of his own in 1990. They were chiefly concerned with the problem of securing computer software from piracy, known as the problem of "software protection", as it was a pressing problem in those days. At that time, there was no comprehensive way to secure software from piracy, as there were just one-off "fixes" and probabilistic heuristics, which could not guarantee comprehensive security. As a result, commercial software makers of the time lost large sums of money due to the twin tasks of allowing users to execute programs, yet not allowing them to redistribute programs, which was very hard to guarantee simultaneously. Goldreich and Ostrovsky's goals were the following:

1) Mathematically formalize the problem of protection against illegitimate duplication as part of the software protection problem.
2) Mathematically formalize the problem of protection against redistribution/fingerprinting software as part of the software protection problem.
3) Encapsulate the problems of protection against illegitimate duplication and protection against redistribution/fingerprinting software as the key problem of the attacker's knowledge about a program from its execution.
4) Reduce the key problem of learning about a program from its execution into an oblivious random-access memory (ORAM) concept.
5) Devise an efficient software protection scheme to simulate an arbitrary RAM program on a probabilistic oblivious RAM.
6) Mathematically demonstrate that, assuming one-way functions exist, the software protection scheme is robust against a polynomial-time adversary who is allowed the freedom to change RAM contents during execution dynamically.

Their methods were achieved by first defining a Software-Hardware-package (SH-package) that physically shields a computer's Central Processing Unit (CPU) and runs an encrypted software program, similar to ATMs' CPUs. The SH-package is rendered functional due to the necessity of physically protecting CPUs and encrypting software programs from being tampered with by any adversary with physical and digital access. A computer with an SH-package contains a small Read-Only Memory (ROM) unit that includes a decryption key for an encrypted program, such that only programs successfully decrypted by the ROM unit key shall be allowed to execute on the CPU. In other words, the CPU is physically shielded while the software, I/O devices, and all external components are digitally shielded. After this basic level of security against tampering, we move on to another problem: How could we prevent the user from knowing any compromising information that the SH-package does not hide from the adversary?
The answer is found within oblivious random access memory (ORAM). They modeled the threat of an adversarial user who attempts to learn compromising information about the program that would enable illegal duplication and redistribution. The adversary had the power to observe and modify everything in the SH-package except for the shielded CPU's registers and ROM unit key. The adversary conducts experiments, which are initiated executions of the shielded CPU on the encrypted program and an adversarial-selected input that can watch and modify CPU-memory communication and memory state, to learn compromising information for an inference/frequency analysis attack. A software-protected program is considered

secure if reconstructing a functionally equivalent software-protected program is not easier when conducting polynomial-timed experiments on the SH-package than naively. More formally, software protection is safe if a polynomial-time adversary's actions on encrypted software running on a shielded CPU are the same as when having access to a specification oracle that, when given an input, outputs the corresponding output and running time. In other words, a software-protected program must behave like a black-box that, upon any input, computes and then returns a special output such that the only information exposed to the adversarial user is its running time and I/O behavior. This requires not only a normal SH-package but also preserving the independence of the memory access pattern, which is accomplished with ORAMs. Stated informally, a shielded CPU defeats experiments with corresponding encrypted programs if a probabilistic polynomial-time (PPT) adversary cannot tell whether it is experimenting with the real shielded CPU or an ideal CPU that never enters infinite loops and returns its theoretical output as if it had never infinite-looped.

The theoretical ORAM system is as follows. The CPU and memory are both Interactive Turing Machines (ITMs). ITMs are bounded work, bounded space, and message multi-tape Turing Machines with write-only output and communication tapes, read-only input and communication tapes, and read-and-write work tapes. The memory's write-only communication tape is the CPU's read-only communication tape, and shares the message length c. The CPU has a linear work tape in message length, while the memory has an exponential work tape. Every message has a register in the CPU's work tape and/or an address in the memory's work tape. The memory has finite control, which is its response to the CPU, while the CPU's finite control differs per CPU. The size of any work tape is $2^k * k'$, and the message length of any message is $k + 2 + k'$ for any parameter k, k'. The memory is probabilistic, while the CPUs can access a random oracle. Assuming one-way functions exist, the CPU's random oracle has one read-only and one write-only oracle tape. All oracle invocation state changes are made in one step to the read-only oracle tape following the write-only oracle tape's queue. A probabilistic CPU is an oracle CPU that also has access to a uniformly-selected function $f : \{0,1\}^{O(k)} \rightarrow \{0,1\}$. Thus, an oblivious RAM (ORAM) is a probabilistic random access memory (probabilistic RAM) such that the access patterns of two separate inputs are identically distributed with the same complexity analysis runtime. Therefore, we need an identical oracle and a random oracle for the oblivious simulation of RAM, as in experiments.

In the end, Goldreich and Ostrovsky pioneered the oblivious computation landscape that is currently of such importance. They were the first to define the subfield as distinct from the rest of the cybersecurity world, mathematically formalize many key assumptions and results that are still relied upon today, and provide the general theoretical framework for the subfield. However, they could not make the field of oblivious computation practical, as their ORAM's runtime of $O(t \log t^3)$ was far too large and their ORAM too complicated to implement to be feasible in a real-world system. In short, they laid the rails but did not build the train of modern oblivious computation's railway. However, we must note that they were decades ahead of everyone else in the cybersecurity community in noticing that security could not be guaranteed with end-to-end encryption (as it would not work for software protection back then), even though they could never have imagined how modern cybersecurity researchers would use their work today to enable global, secure, private, high-speed, and scalable computation.

III. Methods

We have split this Methods Section into several Subsections that cover the high-level system design and all related information during the development Phases $1-5$ in Locker 2.0. We have also added an extra Subsection for Locker 2.0's caveats.

A. *Locker* 2.0*'s High-Level System Design*

Locker 2.0's high-level system design follows a similar design paradigm as several other oblivious computation data storage solutions, such as MongoDB's Data Encryption, Cosmian's Findex server, and Clusion's IEX [13] [14] [15]. MongoDB is a for-profit open-source NoSQL database storage solution intended for large-scale and scalable cloud deployment. Cosmian is a for-profit open-source key-value database storage solution for secure cloud deployment. Clusion is an academic open-source database storage solution designed for research and experimentation. These solutions are similar to Locker 2.0 in that they support secure client-server computation with a trusted client, an untrusted network connection, and a malicious server. However, they do not support Kubernetes with etcd. Overall, these competitors to Locker 2.0 certainly have their place, as they are excellent solutions themselves in some instances, but only Locker 2.0 supports secure client-server computation with a trusted client, an untrusted network connection, and a malicious server for Kubernetes using etcd as the backend key-value store data storage solution. Also, how Locker 2.0 is implemented differs from how the other solutions are implemented, even though they have the same security goal.

Here is the high-level system design of Locker 2.0:

1) Phase 1: Develop a modular plaintext etcd client-server communication protocol from scratch for client-server create/read/update/delete (CRUD) operations. This plaintext client-server protocol is insecure but performant and easily extendable, allowing for additional features, such as oblivious computation.
2) Phase 2: Utilize an already-existing and correct PathORAM oblivious data structure (OBS) implementation library as a generic oblivious random access memory (ORAM) data structure. This enables oblivious computation from the client to the server for the CRUD operations, which is both performant and secure from all non-client attacks using known attacking methods (given a correct implementation and under reasonable circumstances).
3) Phase 3: Utilize an already-existing and correct vORAM+HIRB oblivious data structure (OBS) implementation library as a generic oblivious map/dictionary data

structure using PathORAM as a component, enabling basic read/write oblivious computation. This enables key-value-based oblivious computation from the client to the server for the CRUD operations, which is performant and secure from all non-client attacks using known attacking methods (given a correct implementation and under reasonable circumstances).

4) Phase 4: Develop a comprehensive encrypted multi-map (EMM) data structure based on Phase 2's PathORAM and Phase 3's vORAM+HIRB. As we know that PathORAM and vORAM+HIRB are already correct, we can use vORAM+HIRB as a generic oblivious map/dictionary data structure to implement the EMM. This enables multi-map and key-value-based oblivious computation from the client to the server for the CRUD operations, which is both performant and secure from all non-client attacks using known attacking methods (given a correct implementation and under reasonable circumstances).
5) Phase 5: Modify Phase 1's plaintext etcd client-server protocol to use Phase 4's EMM for client-server create/read/update/delete (CRUD) operations. The EMM slows down the secure protocol from its plaintext variant but grants complete security from all non-client adversaries using known attacking techniques while preserving compatibility with external API calls, meaning that a client can call Locker 2.0's secure etcd client-server protocol the same way as with its plaintext counterpart.

B. Phase 1: etcd Client-Server Communication Implementation Details

The plaintext etcd client-server communication protocol used in Phase 1 was written in the Go programming language in the `plaintext.go` file and the original `secure.go` file for the `main` Go module. Go was chosen for Phase 1 because it has automatic memory management and good etcd and concurrency support. It utilizes Go's etcd library to create a modifiable external API address at HTTP port `500/etcd`. The external API supports HTTP POST requests from curl or wget based on a JSON body of the given user's ID, desired operation, key, and value (for a write request).

C. Phase 2: PathORAM Oblivious Random Access Memory (ORAM) Implementation Details

The PathORAM library used in Phase 2 was written by GitHub users "Jasleen1" and "young-du" as an open-source C++ implementation of the PathORAM oblivious random access memory (ORAM). We had to modify the original PathORAM implementation by extending the block size to 64 bytes to support messages of up to 64 bytes. This specific size was chosen because we wanted the blocks to be small enough to preserve obliviousness while being large enough to contain short messages. Other than that, no other changes were made to the source code, except modifications to enable it to compile on a modern Fedora/RHEL Linux machine and remove the git hidden directory to be self-contained inside the GitHub repository.

However, getting the PathORAM library to correctly interface with the `secure.go` file using `cgo` and be self-contained inside the new custom `oram` Go module (to be compatible with Phase 1's `secure.go`) was a challenge in of itself. First, we had to create a `dummy.go` and `dummy.cpp` filled with nothing but `cgo` flags so that the Go compiler could recognize the PathORAM interface code files `PathORAM.go` and `PathORAM.cpp`. Next, we had to encapsulate all the PathORAM library's functionalities in `PathORAM.cpp` before externally exposing it as C code to be used in `PathORAM.go`, which finally was able to be included inside the `oram` Go module. Also, we had to create a Bash script to dynamically build the PathORAM library and link it for the Go compiler. Since this subcomponent has already been built, new users of Locker 2.0 should not need to build it again.

D. Phase 3: vORAM+HIRB Oblivious Map/Dictionary Implementation Details

The vORAM+HIRB library used in Phase 3 was written by GitHub user "dsroche" as an open-source Python implementation of the vORAM+HIRB oblivious map/dictionary, which the research paper's authors wrote. Keeping in tune with Phase 2's PathORAM implementation, we had to modify the original PathORAM implementation by extending the block size to 64 bytes to support messages of up to 64. Other than that, no other changes were made to the source code, except modifications to enable it to run on a modern Fedora/RHEL Linux machine and remove the git hidden directory to be self-contained inside the GitHub repository.

However, getting the vORAM+HIRB library to correctly interface with the `secure.go` file using an internal HTTP server and be self-contained inside the new custom `hirb` Go module (to be compatible with Phase 1's `secure.go`) was also difficult. First, we had to operate an internal HTTP server for the vORAM+HIRB library, inside its source directory, called `obliv_server.py` under HTTP port `8236`. Then, we create a `hirb.go` file that encapsulates the vORAM+HIRB library's functionality and is a client of the HIRB library server for the `hirb` module. Finally, the `hirb.go` file extensively uses the `oram` module to provide the needed generic ORAM implementation. Since this subcomponent has already been built, new users of Locker 2.0 should not need to build it again, but they need to run `python obliv_server.py` in a separate terminal before using `secure.go`.

E. Phase 4: EMM Oblivious Multi-Map Implementation Details

The EMM oblivious data structure (OBS) protocol used in Phase 1 was written in the Go programming language in the `EMM_client.go` file and the `EMM_server.go` file for the new custom `emm` Go module (to be compatible with Phase 1's `secure.go`). It creates an internal HTTP server that supports EMM's functionality at HTTP port `8245/emm` (which is used inside `secure.go`). The `EMM_client.go` file and the `EMM_server.go` file use the `oram` and `hirb` modules extensively to provide the needed generic ORAM and generic oblivious map/dictionary implementations.

F. Phase 5: Secure etcd Client-Server Protocol with EMM Implementation Details

The `secure.go` file, as defined in Phase 1, is now retrofitted to support oblivious computation with EMM as the middleman between the etcd client's requests and the server's consummate responses. The only possible way that any Locker 2.0's four CRUD operations are fulfilled is by the EMM acting as a secure courier between the client and the server. As a result, the external API is altered to support the EMM while retaining backward-compatibility with Original Locker's external API calls.

G. Caveats of Locker 2.0

However, a few implementation quirks in Locker 2.0 backward-compatibility are present. They will be enumerated in the following list:

1) Each unique message has a maximum size of 64 bytes, as defined in `deps/PathORAM/include/Blocks.h`, `deps/obliv/obliv_server.py`, and `libs/emm/EMM_server.go`.
2) Locker 2.0's default URLs are defined in `secure.go`, `mains/proxy.go`, `mains/plaintext.go`, `deps/obliv/obliv_server.py`, `libs/hirb/hirb_client.go`, and `libs/hirb/hirb_server.go`.
3) All multiple-value JSON responses are returned as lists, but all single-value JSON responses are returned as strings.
4) Locker 2.0's external API's HTTP address for `plaintext.go` and `proxy.go` is `127.0.0.1:5000`, contrasted with `secure.go`'s own external API's HTTP address of `127.0.0.1:5000/etcd`.
5) Locker 2.0's etcd client's read JSON requests ignore the `val` field. Therefore, API developers should also overlook the etcd server's response's `val` field.

IV. Results

This Results Section is divided into three Subsections: An Experimental Design Subsection that describes the A/B comparison benchmarking of the Baseline Encrypted MultiMap (EMM) data structure versus the Plaintext MultiMap data structure, an Experimental Results Subsection dedicated to displaying all related Tables and Figures collected from the benchmark data, and another Interpretation of Results Subsection that interprets the benchmark data.

A. Experimental Design

We have conducted an A/B comparison test between Locker 2.0's Baseline Encrypted MultiMap (EMM) data structure, providing high security at a steep performance cost, and a Plaintext MultiMap, offering no security with near-optimal performance. Both scenarios were evaluated using the `testing_emm.go` Go script, making direct library calls without external networking to eliminate network-induced variability.

The benchmark script comprises three phases: A Warm-Up Phase that mixes reads and writes at a given read ratio, a Delete Phase removing roughly half of the previously written keys, and a Verification Phase confirming that the remaining keys were retrievable. These phases were chosen to account for every possible edge case. Tests were performed using the `medium_keys.txt` dataset containing $79,101$ global city names, initializing the etcd server. Each test scenario had 10 users, a batch size of 3, a maximum value size of 3 bytes, 3 warm-up batches, and an initial 50% read ratio. Benchmarks ran on an AMD64 architecture computer running Fedora/RHEL Linux, equipped with an Intel $i7 - 10700F$ processor, GTX 1660 Super GPU, 16 GB DDR4 RAM, and a 256 GB magnetic storage drive.

We measured processor and memory utilization, peak RAM usage, direct disk Input/Output (I/O) utilization, indirect cache I/O utilization, and execution/wall-clock time as key performance metrics. These carefully-chosen metrics reflect the computing resource utilization of Locker 2.0, demonstrating its real-world practicality. The 4 most extreme test runs, the 2 largest and 2 smallest, were discarded, and the remaining 3 test runs are averaged to remove all outliers. CRUD (Create, Read, Update, and Delete) operations were logged temporarily during each test run for recording purposes and then discarded. All measures ensured that the Plaintext MultiMap and Baseline EMM data structures were structurally identical to ensure fairness.

B. Experimental Results

All benchmark data was compiled and logged in Locker 2.0's `EMM_benchmarks_results.log` file in the `root/` directory. Table I summarizes execution/wall-clock time, processor and memory utilization, and peak RAM usage. Table II outlines direct disk I/O utilization and indirect cache I/O utilization. Figure 3 demonstrates the slight differences in memory utilization but apparent differences in processor utilization between the Plaintext MultiMap and the Baseline EMM data structures. Figure 4 shows the negligible differences in peak RAM usage for the Plaintext MultiMap and the Baseline EMM data structures. Figure 5 displays the no or insignificant differences in all disk and cache I/O utilization except for the apparent discrepancies in indirect cache reads for the Plaintext MultiMap and the Baseline EMM data structures.

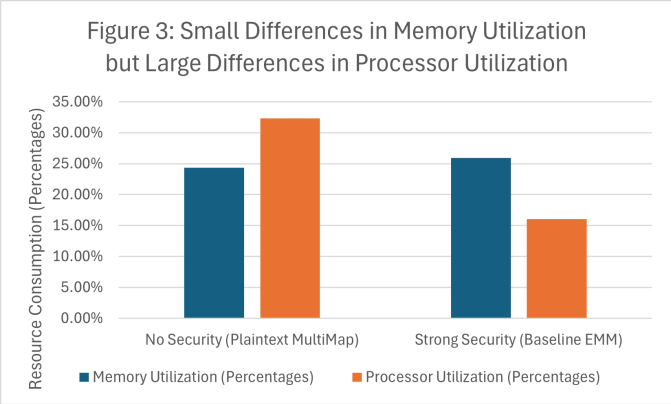


Fig. 3. Relatively small differences in memory utilization but large differences in processor utilization.

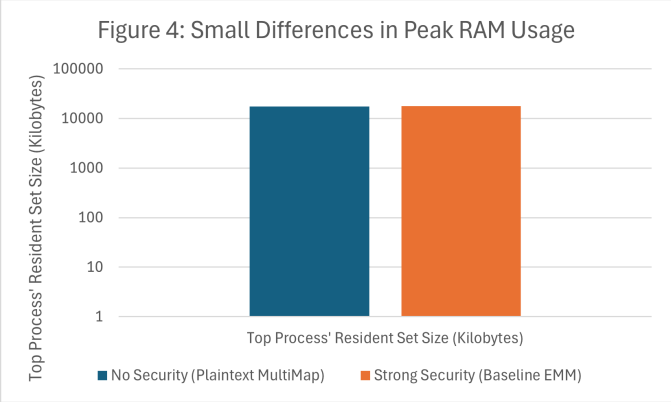


Fig. 4. Negligible differences in peak RAM usage (logarithmic scale on Y-axis).

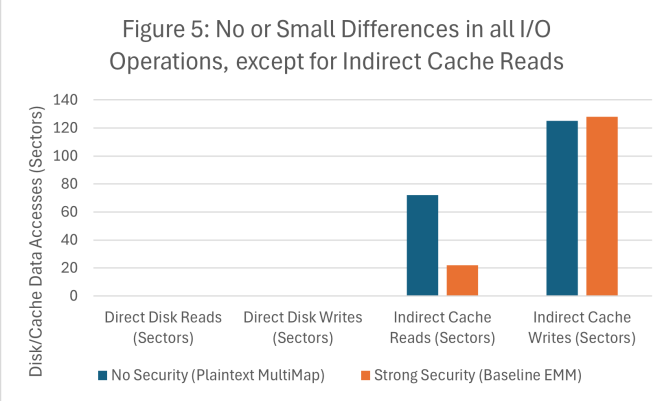


Fig. 5. No or negligible differences in direct disk I/O utilization and indirect cache writes, but clear differences in indirect cache reads.

C. Interpretation of the Experimental Results

The benchmark results display the tradeoff between security and performance. The Baseline EMM execution/wall-clock time was $\frac{22.06-3.05}{3.05} \approx 7$ times slower than the Plaintext MultiMap. Memory utilization showed slight differences, with the Baseline EMM consuming about $\frac{0.2591-0.2433}{0.2433} \approx 7\%$ more memory than its Plaintext MultiMap contemporary. Remarkably, processor utilization was lower for the Baseline EMM by approximately $\frac{0.1607-0.3231}{0.3231} \approx 50\%$ less than its Plaintext MultiMap counterpart. The peak RAM usage metric, determined by the top process' resident set size (RSS), was marginally higher for the Baseline EMM, being $\frac{17877-17329}{17329} \approx 3\%$ more than its Plaintext MultiMap rival.

Disk I/O operations had no results due to Linux's aggressive caching strategy. Still, indirect cache reads were substantially lower at $\frac{22-72}{72} \approx 69\%$ less for the Baseline EMM than the Plaintext MultiMap. Although indirect cache writes exhibited minimal variation, with only a $\frac{128-125}{125} \approx 2\%$ more difference between the Baseline EMM contrasted to the Plaintext MultiMap.

After describing the data in detail, let us analyze how it originated. First, the Baseline EMM implementation is the same data structure as the normal EMM used in Locker 2.0, with the client and server merged to remove the need for an external network connection. This change is not secure, as it would allow a malicious server full access to the client's state and thus allow for a trivial attack since they occupy the same process. However, this A/B comparison test is not used in production; it demonstrates the relative performance of the Baseline EMM data structure versus the Plaintext MultiMap equivalent data structure. Second, this test was conducted on Fedora/RHEL Linux, which has an aggressive caching approach that attempts to avoid direct disk read/writes at all cost, which is why we have 0 direct disk read/writes in our results, due to our memory accesses being warm and close temporally and sequentially, which makes it easier for Linux to cache them. It also appears that the Baseline EMM is disk I/O bounded rather than CPU-bound like the Plaintext MultiMap, which suggests either suboptimal concurrency or long I/O wait times for the Baseline EMM. A visible memory drop during the Baseline EMM's runs is attributed to a likely garbage collection (GC) event. Finally, there is high variance recorded between several Baseline EMM individual runs due to system environment noise.

These benchmark results highlight the ever-present tradeoff between performance and security. Even though the Baseline EMM and Plaintext MultiMap design and testing approaches are identical, we still have varying performance caused by the inherent slowdown that security requires.

V. Future Work

As a single developer developed Locker 2.0 in 10 weeks, there was little time for performance optimization, leaving plenty of room for performance improvements. Locker 2.0's only classic performance optimizations are implementing the optimized version of oblivious data structures (ODS) and batching client requests. Some remaining performance optimizations are caching client requests for massive efficiency improvements and other classical performance optimizations, especially for sequential or frequent read/write operations. Also, using the statistics collected from the rigorous benchmarking, one could optimize the performance of this program based on any bottlenecks present. Of course, optimizing the ODS themselves or using more efficient alternatives is also possible. However, altering the ODS for

TABLE I
SMALL DIFFERENCES IN MEMORY UTILIZATION AND PEAK RAM USAGE, BUT CLEAR DIFFERENCES IN EXECUTION/WALL-CLOCK TIME AND PROCESSOR UTILIZATION.

Table 1:	Execution/Wall-Clock Time (Seconds)	Memory Consumption (Percentages)	Processor Consumption (Percentages)	Peak RAM Usage (Kilobytes)
No Security (Plaintext MultiMap)	3.05	24.33%	32.31%	17329
Strong Security (Baseline EMM)	22.06	25.91%	16.07%	17877

TABLE II
NO OR NEGLIGIBLE DIFFERENCES IN DIRECT DISK I/O UTILIZATION AND INDIRECT CACHE WRITES, BUT CLEAR DIFFERENCES IN INDIRECT CACHE READS.

Table 3:	Direct Disk Reads (Sectors)	Direct Disk Writes (Sectors)	Indirect Cache Reads (Sectors)	Indirect Cache Writes (Sectors)
No Security (Plaintext MultiMap)	0	0	72	125
Strong Security (Baseline EMM)	0	0	22	128

optimization is dangerous because performance optimizations could inadvertently leak data access patterns and open the door to inference/frequency analysis attacks from malicious servers and network eavesdropping adversaries.

Outside of performance concerns, Locker 2.0 would also benefit from adding new features that make it easier for developers to improve Locker 2.0 and system administrators to deploy it in real-world systems. These methods could all be deployed as part of an Infrastructure-as-Code (IaC) automated developer operations (DevOps) effort [16]. To form a complete automated IaC DevOps deployment of Locker 2.0 with GitHub Actions (as it is natural to use it because of Locker 2.0's Git Repository being stored in GitHub), an automatic build script, perhaps as a Bash script or Makefile, would be written that had various options for the cleaning of temporary files, the linting of all code files according to a chosen few standard style guides for visual consistency, and debugging flags that make it easier to find bugs within the codebase [17]. A code scanning tool such as CodeQL could be deployed to catch common errors and cybersecurity vulnerabilities within Locker 2.0 and its dependencies [18]. Semantic versioning could ensure each version is labeled in a structured order, and the status of Locker 2.0's builds, unit testing, and code coverage could be automatically displayed using GitHub Actions to achieve continuous integration/continuous deployment (CI/CD) [19] [20]. Since Locker 2.0 is only tested on a single developer's modern Fedora/RHEL Linux distribution, the application could be containerized with Docker such that all modern Linux, macOS, and Windows operating systems could run Locker 2.0 inside a Docker container [21]. Finally, documentation could be hosted on GitHub Pages on a website built with Markdown and Jekyll [22]. Therefore, as mentioned earlier, all of these tools could be integrated into GitHub Actions and automatically updated with any change onto Locker 2.0. These IaC improvements could make it much easier for a developer to develop in Locker 2.0 or a system administrator to deploy Locker 2.0 in a production system.

VI. CONCLUSION

Privacy and security are fundamental human goals. However, satisfying these goals has become increasingly complex, given the rapid rise of electronic devices processing information at tremendous speeds. Therefore, the modern scientific subfield of cybersecurity was formed. The state-of-the-art cybersecurity paradigm of end-to-end encryption has worked exceptionally well for the last few decades, protecting data content well. Still, the recent development of effective inference/frequency analysis attacks using data access patterns on encrypted data made end-to-end encryption a necessary but insufficient method to achieve realistic and performant privacy and security.

This has led the cybersecurity community to explore different subfields to achieve realistic privacy and security. One promising approach is the subfield of oblivious computation, invented by Goldreich and Ostrovsky in the late 1980s and early 1990s to combat software piracy, which prevents all known attacks by non-client adversaries at a substantial performance cost by allowing clients to access memory obliviously and thus hide their data access patterns. This has gained recent popularity following the PathORAM research paper in 2013, which has caused a flurry of recent research papers in the subfield of oblivious computation. The vORAM+HIRB research paper implemented an oblivious map/dictionary in 2016, while the Encrypted Multi-Map (EMM) research paper implemented an oblivious encrypted multi-map in 2022.

We have decided to apply these theoretical results to practical computer systems, following the footsteps of oblivious data storage solutions such as MongoDB, Cosmian, and Clusion. In particular, we implemented oblivious computation for Kubernetes environments using etcd, an equivalent solution. This effort has borne fruit and is called Locker 2.0. Locker 2.0 prevents all known non-client attacks by implementing an oblivious etcd client-server protocol using EMM as an oblivious data structure (ODS), thus restricting the potential attack surface to only client-side vulnerabilities, primarily side-channel attacks against the client. It is a drop-in client; External users only need to have their JSON requests comply

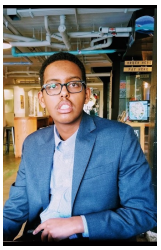
with Locker 2.0's external API, and Locker 2.0 will handle all operations securely. Even though Locker 2.0 has plenty of room for performance optimizations and additional features, it is still the first-of-its-kind oblivious client-server protocol for Kubernetes environments using etcd.

ACKNOWLEDGMENT

As Ismail Ahmed, I thank and acknowledge the extremely essential mentorship and guidance of Professors Jose Renau and Owen Arden for Locker 2.0.

VII. REFERENCES

[1] K. Komamura, "PRIVACY'S PAST: THE ANCIENT CONCEPT AND ITS IMPLICATIONS FOR THE CURRENT LAW OF PRIVACY," *Washington University Open Scholarship*, vol. 96, no. 6, pp. 1337–1365, 2019.
[2] "Cybersecurity History: Hacking & Data Breaches."
[3] G. Lindemulder and M. Kosinski, "What is cybersecurity?" Aug. 2024.
[4] E. Harmoush, "RSA, Diffie-Hellman, DSA: The pillars of asymmetric cryptography."
[5] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Inference attack against encrypted range queries on outsourced databases," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. San Antonio Texas USA: ACM, Mar. 2014, pp. 235–246.
[6] P. Grubbs, T. Ristenpart, and V. Shmatikov, "Why Your Encrypted Database Is Not Secure," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. Whistler BC Canada: ACM, May 2017, pp. 162–168.
[7] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, May 1996.
[8] E. Stefanov, M. van Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," *Journal of the ACM*, vol. 65, no. 4, pp. 1–26.
[9] M. Van Dijk, "Oblivious RAM."
[10] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "A Retrospective on Path ORAM," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1572–1576, Aug. 2020.
[11] D. S. Roche, A. Aviv, and S. G. Choi, "A Practical Oblivious Map Data Structure with Secure Deletion and History Independence," in *2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA: IEEE, May 2016, pp. 178–197.
[12] A. Boldyreva and T. Tang, "Encrypted Multi-map that Hides Query, Access, and Volume Patterns," in *Security and Cryptography for Networks*, C. Galdi and D. H. Phan, Eds. Cham: Springer Nature Switzerland, 2024, vol. 14973, pp. 230–251.
[13] "MongoDB Data Encryption."
[14] "Cosmian Findex server."
[15] S. Kamara and T. Moataz, "Clusion v0.2.0," Jun. 2017.
[16] "What is Infrastructure as Code? - IaC Explained - AWS," https://aws.amazon.com/what-is/iac/.
[17] "GitHub Actions documentation," https://docs-internal.github.com/en/actions.
[18] "CodeQL," https://codeql.github.com/.
[19] T. Preston-Werner, "Semantic Versioning 2.0.0," https://semver.org/.
[20] "Building and testing," https://docs-internal.github.com/en/actions/use-cases-and-examples/building-and-testing.
[21] J. Rajabzadeh, "Build CI/CD pipelines in Go with github actions and Docker," https://dev.to/gopher/build-ci-cd-pipelines-in-go-with-github-actions-and-dockers-1ko7, Jun. 2021.
[22] "Setting up a GitHub Pages site with Jekyll," https://docs-internal.github.com/en/pages/setting-up-a-github-pages-site-with-jekyll.



Ismail Ahmed Ismail Ahmed is a Computer Science and Engineering Master of Science student, after graduating in 2024 with a Bachelor of Science in Computer Science and a Bachelor of Arts in Economics, at the University of California, Santa Cruz. He has conducted this Locker 2.0 Master Capstone Project under the mentorship and guidance of Professors Jose Renau and Owen Arden. His research aims to bridge the gap between applying theoretical concepts in cybersecurity, software engineering, computer networking, operating systems, distributed systems, and databases with real-world systems at massive scales. In other words, his research is focused on connecting low-level digital systems with high-level theoretical computer science concepts, emphasizing cybersecurity, to secure software systems at scale using the latest cybersecurity techniques.

Jose Renau Jose Renau (http://www.soe.ucsc.edu/ renau) is a professor of computer engineering at the University of California, Santa Cruz. His research focuses on computer architecture, including design effort metrics and models, infrared thermal measurements, low-power and thermal-aware designs, process variability, thread level speculation, FPGA/ASIC design, Fluid Pipelines, Pyrope a modern hardware description language, and Live flows improve the productivity of hardware designs. Renau has a PhD in Computer Science from the University of Illinois at Urbana-Champaign.

Owen Arden Owen Arden (https://owenarden.github.io/home/) is an associate professor of computer science at the University of California, Santa Cruz. He studies decentralized security with a focus on using programming language theory and design to build decentralized applications that are secure by construction. His research includes both foundational and practical contributions, from developing novel, more expressive security models and formalized programming languages, to building secure decentralized systems and compilers.