Locker: An Oblivious Datastore for Protecting Data Access Patterns

Ismail Ahmed
University of California, Santa Cruz

Sallar Farokhi
University of California, Santa Cruz

1 Abstract

Cloud customers often face unspoken distrust toward remote services, particularly regarding the security of the underlying hardware infrastructure. Although communications and data are typically encrypted in transit and at rest, vulnerabilities in on-premises hardware can leak sensitive data access patterns to adversaries. This weakness presents particular challenges in Kubernetes environments, where the etcd key-value store maintains cluster state with unencrypted access patterns visible by default. We present Locker, a practical implementation of oblivious computation that hides data access patterns between a fully trusted client and an untrusted server by leveraging a fully trusted third-party proxy. Our implementation extends the theoretical Waffle framework to protect etcd, a critical component of modern Kubernetes and Docker infrastructure. Our evaluation demonstrates that Locker provides strong privacy and security guarantees with only a 22.7% performance overhead compared to unprotected etcd, making it practical for deployment in cloud environments.

2 Introduction

Data access patterns represent a significant security vulnerability in cloud computing. Even when data is encrypted both in transit and at rest, the mere observation of which data is accessed and when can leak sensitive information to adversaries capable of frequency analysis and inference attacks. This problem is particularly acute in Kubernetes, where the etcd distributed key-value store maintains all cluster state and is accessible to potentially compromised components.

In a typical Kubernetes cluster, the kube-api-server communicates with etcd to manage cluster state. An adversary with access to etcd's access patterns could infer information about internal processes, deployed services, and other sensitive cluster operations. While several solutions exist for encrypting etcd at rest and in transit, addressing access pattern leakage remains an open problem.

We present Locker, a system that implements oblivious computation to hide data access patterns. Our approach routes all client requests through a fully trusted proxy server, which generates dummy requests alongside real requests such that an untrusted server cannot distinguish between them. By combining batching, caching, and careful request padding, we achieve privacy-preserving access to etcd with acceptable performance overhead.

3 Motivation and Problem Statement

3.1 Kubernetes and etcd Architecture

Kubernetes has become the dominant container orchestration platform for cloud-native applications. At its core, Kubernetes relies on etcd, a distributed key-value store, to maintain cluster state including pod definitions, configuration, secrets, and internal metadata. All state management flows through the kube-api-server, which communicates extensively with etcd.

A critical vulnerability in this architecture is that etcd's access patterns are observable by default. An adversary with access to etcd—whether through a compromised node, insecure network access, or rogue administrator—can observe which keys are accessed, in what order, and with what frequency. This information can leak substantial details about cluster operations without ever seeing the encrypted data itself.

3.2 Threat Model

We consider two adversaries in our threat model:

1. **Honest-but-curious server:** The untrusted original server executes all requests correctly but may attempt to learn information from access patterns and data content.

2. **Eavesdropping adversary:** An adversary capable of observing all network traffic between the proxy and server can perform frequency analysis to identify real versus dummy requests.

The security goal is to prevent any information leakage from client requests, regardless of access patterns or data content. We assume TLS/SSL provides message integrity and that adversaries cannot inject malicious queries or decrypt ciphertexts. We do assume adversaries have virtually unlimited computational and data collection capabilities and can persist undetected within their accessible portions of the system.

4 System Design

4.1 Architecture Overview

Locker employs a three-party model: a fully trusted client, a fully trusted proxy server, and an untrusted original server. All client requests are routed through the proxy, which implements the Waffle oblivious computation protocol.

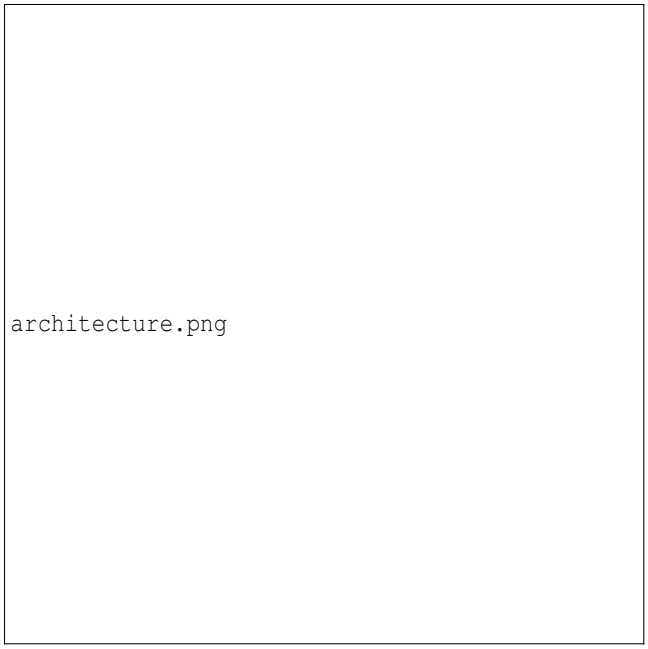

Figure 1: Locker system architecture with trusted proxy

4.2 Waffle Protocol Implementation

The Waffle protocol operates as follows. When a batch of client requests arrives at the proxy:

1. **Deduplication:** The proxy deduplicates requests and removes client identity information.

2. **Caching:** Recently accessed values are stored in a cache to improve response times.

3. **Real values tree:** Recently accessed keys are stored in a binary search tree, sorted by access time.

4. **Dummy values tree:** A separate binary search tree generates dummy keys.

5. **Batch construction:** A batch of size B is constructed as follows: half the batch consists of requests for dummy values, while the remaining slots are filled with least-recently accessed real values and new real requests.

This ensures that to any external observer, exactly half of all requests are for dummy values, making it impossible to distinguish real from fake requests based on frequency analysis.

4.3 Initialization Innovation

A key contribution of Locker over the original Waffle design concerns initialization. The Waffle paper assumes the proxy must be pre-populated with real data values to initialize the binary search tree. We identify this as a potential side-channel vulnerability: an eavesdropping adversary observing initialization could learn information about which keys exist in the system.

Instead, we initialize the real values tree as empty. For the first requests, we pad remaining batch slots with dummy requests. While this reduces performance initially, it eliminates a source of information leakage that could compromise privacy at system startup.

5 Implementation

5.1 Technology Choices

We implemented Locker primarily in Go. While Python and JavaScript have etcd libraries, neither supports etcd version 3.5, which is required for our implementation. Go provides several advantages:

- **Performance:** Go is compiled, providing significantly better performance than interpreted languages.

- **Concurrency:** Go's native goroutines and multi-threading capabilities are essential for handling multiple concurrent clients at scale.

- **etcd Integration:** Google authored both Go and etcd, enabling seamless interoperability through native libraries.

We also developed Bash scripts for rigorous testing and benchmarking, with code stored in a GitHub repository licensed under Creative Commons Attribution 4.0 International. The repository is available at: https://github.com/sfarokhi/CSE239A.

5.2 Proxy Implementation Details

The proxy implementation manages the following:

```
type WaffleProxy struct {
    realValues    *BinarySearchTree
    dummyValues   *BinarySearchTree
    cache         map[string][]byte
    batchSize     int
    dummyRatio    float64
}

func (wp *WaffleProxy) ProcessBatch(requests []Request) []Response {
    deduplicated := deduplicateRequests(requests)
    batch := constructBatch(deduplicated, wp.batchSize, wp.dummyRatio)

    results := forwardToServer(batch)
    return filterRealResults(results, deduplicated)
}
```

6 Evaluation

6.1 Evaluation Methodology

We benchmark Locker against a baseline of native etcd running without the Waffle proxy. Our evaluation parameters include:

1. **Database size:** Three datasets of increasing size (385, 80,000, and 717,900 values).
2. **Cache and dummy size:** Configurable cache sizes and dummy value counts.
3. **Read-write ratio:** Varying proportions of reads to writes (tested at 70-30 read-write split).
4. **Concurrent users:** 10 simultaneous users accessing the system.

All benchmarks were conducted on an Ubuntu Linux virtual machine running on a standard commercial desktop computer.

6.2 Experimental Configuration

- Maximum batch size: 100
- Maximum value size: 10 bytes
- Cache size: 10,000 entries
- Maximum dummy values: 10,000
- Read-write ratio: 70% reads, 30% writes
- Concurrent users: 10

Configuration	Execution Time (seconds)
Native etcd (baseline)	15.226
Waffle proxy	19.701

Figure 2: Benchmark results

6.3 Results

The Waffle proxy implementation required 19.701 seconds compared to 15.226 seconds for native etcd, representing a performance overhead of approximately 22.7%. Despite the batching and multi-threaded architecture of the proxy, the overhead is higher on a single desktop computer than would be expected in a distributed cloud environment. However, the overhead is modest enough to be practical for real-world deployment.

We anticipate that performance would improve significantly in actual cloud environments where distributed computation can better leverage the proxy's batching and parallelism capabilities.

7 Security Analysis

7.1 Privacy Guarantees

Against an honest-but-curious server, Locker provides the following guarantees:

- **Data hiding:** All requests are encrypted, preventing plaintext data leakage.
- **Access pattern hiding:** The server cannot distinguish real requests from dummy requests.
- **Client anonymity:** The server has no knowledge of client identity.

Against an eavesdropping adversary, Locker ensures:

- **Frequency obliviousness:** All batches contain the same ratio of real to dummy requests, preventing frequency analysis.
- **Timing protection:** Batch processing obscures the timing of individual requests.

7.2 Limitations

The security of Locker relies on several critical assumptions:

1. The client is fully trusted and secure.
2. The proxy server is fully trusted and secure.
3. Network encryption (TLS/SSL) is secure and unbroken.

If any of these assumptions are violated, the security guarantees may be compromised.

8 Future Work

While Locker demonstrates the feasibility of practical oblivious computation for key-value stores, several avenues for future work remain:

8.1 Kubernetes Integration

Our current implementation focuses on etcd in isolation. A natural extension would be full integration with Kubernetes, allowing the system to protect access patterns across entire clusters rather than just the underlying key-value store.

8.2 Relaxing Trust Assumptions

The assumption of a fully trusted third-party proxy is unrealistic in many practical scenarios. Future work should explore mechanisms to relax this assumption while maintaining acceptable performance, privacy, and security guarantees. Approaches might include:

- Hardware-based trusted execution environments (TEEs) such as Intel SGX or ARM TrustZone.
- Oblivious hardware enclaves for the proxy.
- Multi-party computation schemes involving multiple semi-trusted proxies.

8.3 Helm Deployment

We would like to publish Locker as a Helm chart, enabling users to easily deploy the oblivious proxy alongside Kubernetes clusters without extensive configuration.

8.4 Performance Optimization

Additional optimizations could reduce the performance overhead:

- Better caching strategies to reduce cache misses.
- Adaptive batch sizing based on request arrival patterns.
- GPU acceleration for cryptographic operations.

9 Conclusion

Locker demonstrates that practical oblivious computation for data access pattern protection is feasible. By extending the Waffle framework to protect etcd, a critical infrastructure component for Kubernetes and Docker, we have created a system that provides meaningful privacy and security benefits to cloud customers and operators alike.

The 22.7% performance overhead, while non-trivial, is reasonable given the strength of security guarantees provided. In distributed cloud environments, we expect this overhead to be even smaller due to better parallelism.

Our work enables greater security and privacy for cloud computing infrastructure while reducing legal liability for both cloud customers and operators. However, the unrealistic assumption of a fully trusted proxy creates opportunities for future research to develop more practical variants that relax this requirement while maintaining strong security properties.

10 Implementation Credits

Sallar Farokhi wrote the majority of the Waffle proxy implementation, with Ismail Ahmed contributing to portions of it. Ismail Ahmed wrote the majority of initialization, benchmarking, and testing code, and authored the majority of the project report and presentation, with Sallar Farokhi contributing to these documents as well.