QUICKeR: Quicker Updates Involving Continuous Key Rotation

Lawrence Lim
University of California, Santa Barbara
lawrenceklim@ucsb.edu

Wei-Yee Goh
University of California, Santa Barbara
weiyeeg@gmail.com

Divyakant Agrawal
University of California, Santa Barbara
agrawal@cs.ucsb.edu

Amr El Abbadi
University of California, Santa Barbara
amr@cs.ucsb.edu

Trinabh Gupta
University of California, Santa Barbara
trinabh@ucsb.edu

ABSTRACT

Since the advent of cloud computing, storing large quantities of sensitive data on remote services have demanded secure encryption practices to maintain data privacy. Merely encrypting data at-rest leaves ciphertexts vulnerable to inevitable key leakage from mismanaged keys, side-channel attacks, and security bugs. To combat this, we propose QUICKeR, a system to facilitate continuous key rotation and ciphertext updates. Unfortunately, continuous ciphertext updates from key rotation interferes with routine access operations by introducing race conditions and performance bottlenecks that preclude prior works from practical use. QUICKeR addresses these concerns through protocols that eliminate race conditions and even achieve the stronger linearizability correctness criterion. In addition, QUICKeR proposes three approaches to implement frequent ciphertext updates. We show how these different update schemes perform under different database bottlenecks. Finally, we demonstrate that QUICKeR is practical through end-to-end experiments showing that routine access operations are not significantly impacted by ciphertext update operations.

KEYWORDS

Key Rotation, Updatable Encryption, Linearizability

1 INTRODUCTION

Cloud computing has enabled the distribution of computing resources over the Internet. The ubiquity of cloud computing in everyday life means that external hardware continues to store more private data than ever before, necessitating strong security and privacy measures. In order to ensure confidentiality, standard practice involves encrypting data at-rest to prevent complete privacy compromise from unauthorized access to a database.

However, merely encrypting data at-rest still leaves a major problem: cryptographic keys are susceptible to leakage over time in numerous ways. Mishandled keys can be inadvertently revealed or deliberately stolen by rogue employees and insiders, resulting in decryptable ciphertexts [23, 33]. Malware [32] and attacks on buggy code [21] can steal sensitive cryptographic secrets. Finally, cryptographic operations are vulnerable to side-channel attacks such as power output analysis, timing based attacks, electromagnetic radiation analysis and more attacks that analysts have yet to uncover [22, 29, 30]. Leaked keys are often the cause of large data breaches, resulting in millions of dollars in monetary damage, loss of reputation, and other severe repercussions [33].

One standard practice to limit the effects of inevitable key leakage is by periodically retiring old cryptographic keys for new ones, called *key rotation* [2, 28]. Periodic key rotation shortens the lifetime of a cryptographic key, called a cryptoperiod, by transitioning an operational key to a post-operational key, which is an accessible key for decrypting data but no longer encrypts new data [2]. Not only are short cryptoperiods good practice, they are a recommended government standard that limits ciphertext exposure from key compromise, limits the available time for computationally intensive cryptanalysis, and restricts the time available for an attacker to penetrate access mechanisms protecting cryptographic keys [2, 28]. While government standards define cryptoperiods of up to 2 years, cloud service providers have default cryptoperiods of a month [1, 8, 10, 15].

In addition to key rotation, updating ciphertexts (also known as re-encrypting) to the most recent key limits the effects of key leakage because it provides *post-compromise security*, meaning that messages encrypted by new keys maintain confidentiality even when outdated keys are compromised. However, safely destroying post-operational keys without losing access to the data is impossible without updating the ciphertext. Hence, modern systems require periodic ciphertext updates.

This presents a problem because the costs of key rotation and updating ciphertexts at-rest grow as the amount of data stored in untrusted infrastructure continues to grow. These costs cannot be ignored because updating ciphertexts interferes with the routine access operations of storing and retrieving encrypted data. If performed offline, then key rotations and ciphertext updates would require the system to be periodically taken down for maintenance, which can cost companies millions of dollars. An offline solution may still be insufficient anyway because the amount of sensitive data stored could grow to the point that the only solution is constantly updating ciphertexts to fit within a cryptoperiod. If key rotations are performed online, then updating ciphertexts to new keys creates race conditions between updates and routine operations, resulting in incorrect executions. Finally, key rotation and ciphertext updates produce performance bottlenecks like network

Proceedings on Privacy Enhancing Technologies 2024(1), 58–73

https://doi.org/10.56553/popets-2024-0005

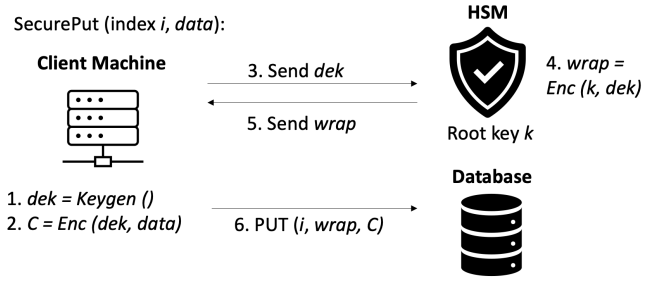

Figure 1: SecurePut

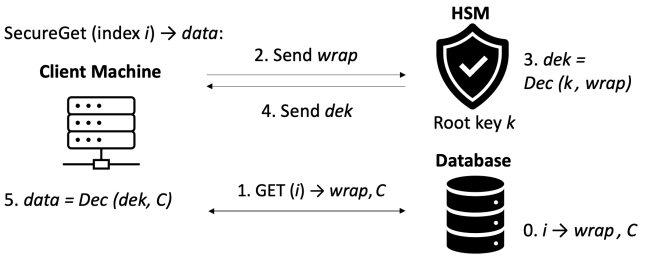

Figure 2: SecureGet

bandwidth that limit how frequently both ciphertext updates and routine access operations can be performed.

Prior work has introduced updatable encryption, where encrypted data can be updated to a new key dynamically [4–6, 11–13, 17, 18, 20, 24, 25, 27]. However, these works do not consider the impact of ciphertext updates on routine access operations. In addition, database systems, which implement their own concurrency control, are not optimized to handle the need to update an entire database that key rotation demands.

We introduce QUICKeR, a system which enables online key rotations and ciphertext updates that occur concurrently with routine access operations of storing and retrieving encrypted data. QUICKeR ensures correctness of concurrent operations by eliminating race conditions. To achieve this, a key contribution of our work is modifying routine access operations and ciphertext updates to enable the stronger correctness guarantee of linearizability. We support this with a formal proof of linearizability. In addition, QUICKeR introduces three practical approaches for updating ciphertexts with tradeoffs in performance and security. Our first approach is fairly straightforward, having the fewest security assumptions but high network costs. Our second and third approaches utilize updatable encryption, which result in lower network costs but requires stronger security assumptions. We show how the different update approaches perform under different database bottlenecks such as network bandwidth and CPU. We also implement an end-to-end system for QUICKeR and experimentally demonstrate its scalability to more data. Hence, QUICKeR improves the standard cloud practice for encrypting data at-rest, a process known as envelope encryption. Envelope encryption is a two-level encryption practice where the plaintext data is encrypted using a data key, and the data key is encrypted with a root key.

The remainder of the paper is organized as follows. Section 2 elaborates on the problem statement. Section 3 discusses QUICKeR's system architecture and the modifications made to SecureGet, SecurePut, and ciphertext updates needed to achieve linearizability as a correctness criterion. Section 4 discusses our straightforward ciphertext update approach. Section 5 describes our ciphertext update approach using updatable encryption. Section 6 evaluates the performance of our proposed approaches through experimental results. Finally, Section 7 concludes the paper.

2 PROBLEM STATEMENT

In this section, we explain the problem statement by first describing a standard deployment of envelope encryption [9]. Then, we explore key rotation for this deployment in practice and in prior works. Based on this overview, we identify goals and limitations of prior work that we aim to address.

2.1 Data Model

In a standard cloud deployment, data can be stored on a key-value store with a PUT request and retrieved with a GET request. We use the term *index* in place of key-value pair keys to disambiguate them from cryptographic keys, so a PUT request takes as parameters index i and value v to store v in a database server. A GET request takes as parameters index i and returns a value v. We define SecurePut and SecureGet as operations that store and retrieve encrypted data values using PUT and GET requests with a plaintext index, respectively.

We assume a simplified database backend with a single key-value store. Although we do not assume transactional processing guarantees, we assume that the key-value store ensures correctness by locking data items until completion. Typical database backends replicate and shard data amongst many different machines for availability and scalability reasons. We leave open as future work research questions related to applying our techniques to a distributed system and transactional processing system.

2.2 Envelope Encryption

Private data can be securely stored on cloud databases when encrypted by a cryptographic key. However, government and industry regulations [2, 28] mandate that the key must be stored and operated over cryptographic modules such as Hardware Security Modules (HSMs) to prevent adversaries from accessing the key. HSMs are certified to provide international standards of physical security, including the highest level of FIPS 140-2 security [26].

Since HSMs are limited in their ability to operate on large data sizes and to scale to large numbers compared to conventional machines, encrypting many megabyte-sized messages on HSMs becomes computationally expensive and a primary bottleneck. To solve this problem, a recommended design pattern by the Payment Card Industry Data Security Standard (PCI-DSS) [28] and standard practice by many cloud service providers such as Google [9], Microsoft [15], IBM [10], and Amazon [1] is to envelope encrypt data.

Envelope encryption is the practice of encrypting plaintext messages with a data key into a ciphertext and encrypting the data key with a root key into a wrap.[1] The wrap and ciphertext together form the envelope encryption and can be safely stored on the cloud database. This design eliminates the HSM as a bottleneck because common conventional machines encrypt plaintext data with data keys. Instead of HSMs encrypting many large plaintexts, HSMs are only required to encrypt small data keys with root keys. Since data keys and wraps are orders of magnitude smaller in size (less than 100 bytes) than typical plaintext data (kilobytes to megabytes), the HSM workload is significantly reduced. Additionally, HSMs have limited key storage capacity and this setup ensures that HSMs do not store many unique data keys.

SecurePut and SecureGet are routine access operations for storing and retrieving envelope encrypted data. These operations typically involve three types of machines: (1) a client machine, (2) the Hardware Security Module (HSM), and (3) the database server. To SecurePut a plaintext data value (Figure 1) for an index i, a client machine generates a new data key dek, encrypts the plaintext data with the data key dek into a ciphertext, and then the HSM encrypts dek with the root key k into a wrap. Finally, the wrap and ciphertext are stored in the database. To SecureGet data (Figure 2) for an index i, the reverse operation is performed. The client machine communicates with the HSM to decrypt the wrap using the root key to retrieve the corresponding data key, and finally decrypts the ciphertext using this retrieved data key to access the private plaintext. While other server side encryption system architectures allow the database to interface directly with the HSM to encrypt and decrypt data on the database locally, we study a client side encryption architecture because a compromised database in a server side encryption model completely compromises all data. Hence, this client side encryption approach is more secure.

2.3 Key Rotation in Practice

In practice, plaintext data can be large in size, incurring high costs when an entire database of ciphertexts is updated. For instance, Google reports storing data in ciphertext chunks typically between 256 KB and 8 MB in size [7]. Hence, current cloud providers may at most rotate root keys and update wraps [1, 8, 10, 15] while avoiding data key rotations and ciphertext updates because updating small key wraps incurs significantly lower costs. These updates are performed by generating a new root key on the HSM, retrieving wraps from the database, decrypting wraps with the old root key, encrypting new wraps with the new root key, and finally uploading the new wraps to the database. Eschewing data key updates still has a major flaw: the data keys are not rotated and thus have no post-compromise security. If either root keys or data keys are compromised, then ciphertexts are still vulnerable. Hence, we aim to rotate both data keys and root keys, thereby updating the entire ciphertext to limit the size of the vulnerability window.

[1] Data keys are also known as data encryption keys (DEK) and root keys also known as key encryption keys (KEK).

2.4 Threat Model & Goals

To achieve a practical solution to frequent key rotation, we discuss the threat model and our goals in rotating keys within a cryptoperiod while minimally impacting routine access services.

Threat Model: We allow an adversary to learn any outdated keys and learn ciphertexts updated to current keys. Since our goal is to provide post-compromise security, then attacks that compromise outdated root and data keys and view currently encrypted data on the database should not provide an adversary any power to violate confidentiality. We remark that this is similar to the security models of many updatable encryption works [4–6, 11–13, 17, 18, 20, 24, 27] and most similar to Jarecki *et al.*'s security model [16], except we have a stronger model that additionally allows for data key compromises.[2] We discuss how our security assumptions differ between our three proposed solutions in Section 5. We do not prevent denial-of-service attacks, so if an adversary compromises the database and blocks update operations, then our system no longer limits vulnerability windows. However, even if a rogue database were to deviate from the protocol, we still aim to ensure that the privacy of ciphertexts in the database is maintained and that no extra information is leaked. Finally, we allow an adversary who compromises the database to modify or insert ciphertexts, so client machines should be able to verify the authenticity of a plaintext.

Security Goal: Since it is difficult to provide any security guarantees when an adversary gains access to a ciphertext decryptable by a compromised key (also known as *trivial wins*), we aim to update all ciphertexts from old cryptographic keys to new keys by the time a cryptoperiod expires. This limits the vulnerability window that an adversary has to perform an attack by providing post-compromise security. Higher key rotation rates mean that more ciphertexts can be updated to satisfy a cryptoperiod and shorter vulnerability windows during which the database continues to store ciphertexts encrypted by compromised keys.

Correctness Goal: Besides ensuring that client updates (SecurePut) and ciphertext updates lead to ciphertexts that can be correctly retrieved (SecureGet), one problem is that concurrent routine access and ciphertext update operations create race conditions that can result in incorrect executions. We aim to eliminate such race conditions by modifying routine access operations and ciphertext updates in order to provide and prove linearizability guarantees. This ensures that routine access and ciphertext update operations can execute concurrently and still guarantee a correct ordered sequence of events.

Performance Goal: Our system aims to perform concurrent updates alongside routine access operations where key rotations and ciphertexts updates can occur frequently with minimal loss in routine access operation throughput. In addition, routine access operations should not incur high latency penalties for using schemes that enable continuous key rotation. Finally, our solution should scale to realistic workloads, capable of rotating databases comprising of a large number of ciphertexts with message sizes between 256 KB to 8 MB within a cryptoperiod.

[2] Although Jarecki *et al.*'s security model further ensures that the HSM be oblivious to the data key, their techniques can also be applied our solutions to achieve the same guarantees.

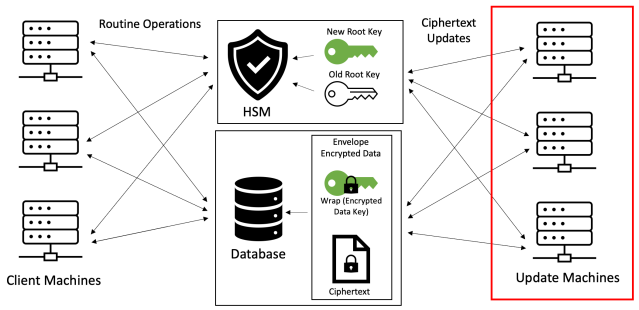

Figure 3: System Architecture

2.5 Prior Work

Prior work has proposed different updatable encryption schemes [4–6, 11–13, 17, 18, 20, 24, 27] to enable more efficient key rotations, but no prior works consider a practical deployment where factors such as concurrent operations and overall system performance become important. They either do not rotate both root keys and data keys or fail to achieve the outlined goals of practical, frequent updates while minimizing its impact on routine access operations. Jarecki *et al.* [16] and ReCrypt [11] are the only two related works that explore a similar system model of envelope encryption. Jarecki *et al.* [16] proposes an oblivious updatable key management system based on Diffie-Hellman assumptions that updates only root keys; however, this work is unable to provide post-compromise security to data keys. Everspaugh *et al.* [11] observe that merely rotating root keys provides insufficient security and proposed an updatable encryption scheme called ReCrypt that updates both root and data keys. Because ReCrypt relies on key-homomorphic pseudo-random functions (PRFs), they report performance costs of encryption and decryption 1000x slower than standard private-key encryption schemes. Using ReCrypt would therefore impose an undesirable high latency penalty on routine access operations. We aim for a system that is practical for basic operations and rotates keys often to minimize the vulnerability window of ciphertext and keys. In addition, both works do not consider the impact of update operations on routine access operations. For instance, updating frequently can interrupt routine access service by creating race conditions or consuming network bandwidth and processors shared by SecurePut and SecureGet operations, which decreases the throughput of these routine access operations. These are critical concerns in our system, QUICKeR.

Lastly, though conventional solutions to concurrency control have long existed, such as multiversion concurrency control (MVCC) [3] and optimistic concurrency control (OCC) [19], each has its flaws in our domain. MVCC may leave old ciphertexts on the database (due to long running transactions, backup and recovery, or delayed garbage collection) which can violate post-compromise security, while OCC may trigger many aborts and retries which affects performance. We deviate from these approaches and provide linearizability guarantees by carefully considering every concurrency violation.

3 QUICKER ARCHITECTURE

We introduce our system, QUICKeR, which modifies the standard envelope encryption deployment. Our system architecture is generalized to three different approaches of ciphertext updates with varying trade-offs.

3.1 QUICKeR System Overview

QUICKeR consists of a secure database (a key-value store) storing client data encoded with envelope encryption; an HSM that stores root key information; a set of client machines that issue secure routine access operations on the database and a set of trusted update machines that continuously rotate keys and update ciphertexts concurrently with the routine access operations issued by the client machines (Figure 3).

Key rotations and ciphertext updates operate in *rounds*. Before the round, the HSM initially has a *prior root key* generated from the previous round. The update machines begin a new round with a brief setup step where it communicates with the HSM to generate a *new root key* and notifies the database and client machines of the new root key. For the entire round, the HSM stores both the new root key and the prior root key, which we call the *old root key*. The new root key is operational, i.e., it is the current, active key for encrypting and decrypting data. The old root key is post-operational, i.e., it can be used for decrypting data but no longer encrypts new data. For this paper, we assume a single old root key and new root key, but QUICKeR extends to systems that want multiple root keys for different data by having multiple old and new root keys in a round. In each round, update machines update one-by-one all envelope encrypted ciphertexts in the database from the old root and data key to the new root key and the newly generated data key. This update is performed by one of the three ciphertext update approaches. Once the update machines finish updating a ciphertext, the new ciphertext is PUT into the database, any data using the old envelope encrypted ciphertext is removed from the database, and the update machines continue to update other outdated ciphertexts. This means that during a round, some ciphertexts are still encrypted by the old root key, some ciphertexts are in the process of being updated, and the rest have already been updated to the new root key. A round of key rotation is completed when all ciphertexts encrypted by the old root key have been updated the new root key and the HSM destroys the old root key to ensure it is not accessible to an adversary. A new round of key rotations can subsequently start. The time taken for a round of updates to start and end is the maximum vulnerability window of a ciphertext and should be no longer than the standard cryptoperiod of keys.

Performing frequent key rotations and ciphertext updates introduces race conditions where concurrent SecureGet, SecurePut, and ciphertext updates can lead to incorrect executions. Hence, QUICKeR provides a generalized solution to ensure linearizability [14], which is applied to all three approaches for updating ciphertexts. Linearizability means that the operations reflect an ordered sequence of events, thereby ensuring that executing concurrent operations always results in a correct execution. We now discuss the challenges with concurrent operations and our modifications to routine access operations and ciphertext updates.

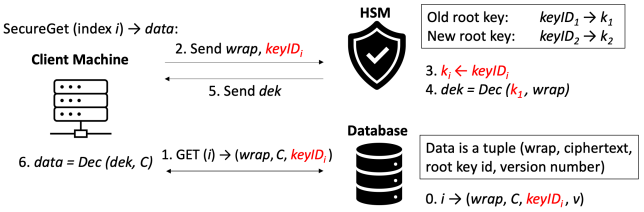

Figure 4: QUICKeR SecureGet

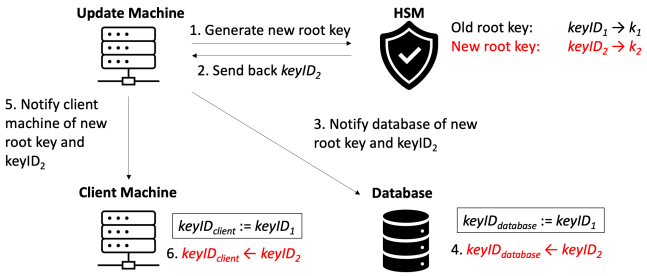

Figure 5: Setup step where an update machine notifies the database and client machines.

3.2 SecureGet Correctness & Concurrency

Update machines frequently rotating keys and updating ciphertexts introduces several challenges for executing SecureGet:

(1) Client machines need to know which root key to use to decrypt ciphertexts.
(2) Retrieving a ciphertext in the process of being updated can result in incorrect retrieval.
(3) A client machine may retrieve a ciphertext encrypted by a destroyed root key.

Challenge (1): Since ciphertexts update operations occur one-by-one during a round, some ciphertexts on the database will be encrypted by the old root key and some ciphertexts will have been updated to the new root key. The HSM stores both the old and new root key so the client machines need to distinguish which root key decrypts a ciphertext.

To solve (1), QUICKeR helps client machines distinguish between a wrap encrypted by an old and a new root key by storing root key identification, called a *root key id* along with each ciphertext and wrap on the database. A root key id uniquely identifies the root key on the HSM used to encrypt the wrap. When an update machine updates a ciphertext to the new root key, the root key id, wrap, and ciphertext all replace the old data. A client machine executing SecureGet is now able to GET the root key id, wrap, and ciphertext. The client machine then uses the root key id to decrypt the wrap, correctly accessing the data encryption key, and then decrypt the ciphertext. Whether the ciphertext has been updated to the new root key or not, the client machine will correctly determine the root key used to decrypt the data.

Challenge (2): A ciphertext may be in the process of being updated to the new root key. Data retrieval for such ciphertexts can result in incorrect executions. For example, a SecureGet and ciphertext updates can concurrently occur for the same index of the database. The client machine executing SecureGet may GET a partially updated ciphertext or updated wrap with the old ciphertext, resulting in incorrect retrieval.

To solve (2), QUICKeR requires that ciphertext update operations *atomically* replace the old ciphertext data on the database and that SecureGet atomically GET data from the database. Atomically replacing data means that although ciphertext update operations take processing time, the operation modifies nothing in the database until all of the processing time completes. When the processing time completes, all three of the wrap, ciphertext, and root key id are PUT into the database at once. An atomic GET retrieves the wrap, ciphertext, and root key in one database operation before starting another database operation. This helps guarantee linearizability because SecureGet can only GET untouched or completely updated data. A SecureGet will GET the completely updated ciphertext data if it executes after the ciphertext atomically updates and will otherwise GET the old ciphertext data, resulting in a linearizable execution where the SecureGet occurs before or after a ciphertext update.

Challenge (3): A client machine may retrieve an envelope encrypted ciphertext that it cannot decrypt because the HSM has destroyed its root key. This can occur towards the end of a round when a client machine executes a SecureGet and retrieves a ciphertext still encrypted by the old key. As this is happening, the update machines finish updating all ciphertexts to the new root key and the HSM destroys the old root key, marking the end of the round. The client machine is now unable to decrypt the envelope encrypted ciphertext.

To solve (3), the client machine retries the SecureGet by retrieving an updated ciphertext. Since a round of ciphertext updates has completed, the retry is guaranteed to GET the updated ciphertext.

SecureGet Summary: In accordance to these challenges, Figure 4 shows the modifications necessary to SecureGet. Given index i, a client machine atomically GETS the ciphertext C, wrap $wrap$, and root key id $keyID_i$ from the database. Using $keyID_i$, the client machines identify the correct root key on the HSM to decrypt the wrap $wrap$ and access the data key dek. Though not depicted in Figure 4, if the HSM has destroyed the key corresponding to $keyID_i$, then the client machine simply retries the entire operation. Finally, the client machine can decrypt C using the dek into the plaintext data. Each ciphertext also has an associated *version number* v_i, which is needed for modifications to SecurePut that we discuss later.

3.3 SecurePut Correctness & Concurrency

Frequently rotating keys and updating ciphertexts introduces challenges for executing SecurePut:

(1) Client machines may SecurePut data by envelope encrypting with a root key that is no longer operational.
(2) Ciphertext update operations may incorrectly overwrite concurrent SecurePut operations.

Challenge (1): Since the operational root key changes every round, the client machines may not be aware of new root keys and encrypt using root keys that are no longer operational (either have become post-operational or deleted completely). This is problematic because this can result in lost access to plaintext data when its root

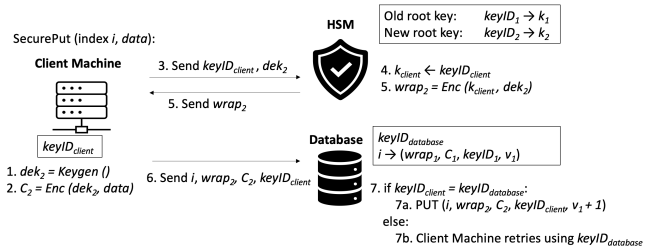

Figure 6: QUICKeR SecurePut

key is deleted and could pose a security flaw when messages are encrypted with expired keys.

To solve (1), QUICKeR has a setup step at the beginning of a round where update machines notify all client machines and the database of the new root key. This enables client machines to encrypt with the new root key and the database to reject ciphertexts not encrypted by the new root key. If the database rejects a ciphertext, then client machines retry the SecurePut by encrypting their data with the new root key. Figure 5 summarizes the setup step. The client machines and database each locally keep track of the new root key id with local values $keyID_{client}$ and $keyID_{database}$, respectively. The client machines and database initially have $keyID_1$ assigned as the new root key id referring to k_1. When a new round begins, the update machines (1) generate a new root key k_2 on the HSM, (2) obtain the root key id $keyID_2$, and (3) notify the database of the new key id $keyID_2$. The database (4) updates its local value $keyID_{database}$ to $keyID_2$ so the database rejects ciphertexts that aren't encrypted with the new root key. This ensures that all accepted ciphertexts are encrypted by an operational root key. The update machines then (5) notify the client machines, which (6) update their local value $keyID_{client}$ to $keyID_2$. QUICKeR notifies the database before the client machines so that the database only receives ciphertexts encrypted by root keys that it is aware of. After the setup step, client machines only complete SecurePut operations by encrypting with the new root key so that new plaintext data moving forward is encrypted with an operational root key.

Challenge (2): A client machine executing a SecurePut for a ciphertext concurrently with the ciphertext update process can result in incorrect execution. Consider a ciphertext update operation in the process of updating a specific ciphertext. The ciphertext update process GETS the original ciphertext to update and during its processing time, a concurrent SecurePut operation PUT newer data; however, this newer data is overwritten when the ciphertext update operation completes by PUTTING a ciphertext correponding to the original data, resulting in a lost SecurePut.

To solve (2), one classic approach to handle race conditions and ensure linearizability is by blocking (using locks) either update operations or SecurePut operations until the other operation completes. A problem with this approach is that since the ciphertext update operation may last a long time, waiting for an update to complete means that several SecurePut operations can queue up, significantly impacting throughput and the performance of the system. QUICKeR introduces a more efficient solution by using atomic PUT in SecurePut and selectively aborting ciphertext updates to avoid overwriting newer data. Ciphertext update operations are aborted when the ciphertext has been modified by a SecurePut during its execution. QUICKeR uses a *ciphertext version number* for each ciphertext to detect such changes. The version number is incremented whenever a SecurePut PUTS newer data, so that during the ciphertext update process, the database atomically checks the version number for changes before replacing with the ciphertext updated to the new root key. If the version number has changed, the ciphertext update is aborted for this particular ciphertext. Recall that SecurePut is using the new root key, and hence its ciphertext reflects new data encrypted by the new data and root key. This ensures linearizability without completely blocking operations because either the ciphertext update PUTS an updated ciphertext before the ciphertext is overwritten by a concurrent SecurePut, or a SecurePut atomically PUTS new data and increments the version number.

SecurePut Summary: Figure 6 summarizes QUICKeR's modifications to SecurePut which performs an update to the data of the entry with index i in the key-value store. The client machine and database each have a root key id: $keyID_{client}$ and $keyID_{database}$ respectively, which is relayed by an update machine at the beginning of a round. The client machine uses $keyID_{client}$ to envelope encrypt its data, and the database uses $keyID_{database}$ to ensure that the ciphertext is encrypted by the active key (Step 7). If the ciphertext is encrypted by the operational key, then the database atomically replaces the entry corresponding to index i with the updated ciphertext and increments the corresponding ciphertext version number (or 0 if uninitialized).

Insertions and Deletions: SecurePut operations for new indices (i.e. insertion) do not cause concurrency bugs because SecurePut always encrypts with the most recent key during a round. In future rounds, update machines can still update them to newer keys. Regarding deletion, the database can abort a concurrent ciphertext update if the index has been deleted and still guarantee linearizability.

3.4 Linearizability Proof

We provide an informal proof sketch here and defer our formal proof for linearizability correctness of QUICKeR's SecurePut, SecureGet, and ciphertext updates to Appendix A.

Theorem: For all concurrent executions of SecureGet, SecurePut, and ciphertext updates in QUICKeR, there exists a linearization where the order of non-overlapping operations is maintained and the behavior of QUICKeR is correct.

The main ideas behind the proof are as follows: 1) We assume that some steps of operations are atomic by the implementation of the database (such as the ones described in SecurePut, SecureGet, and ciphertext updates). 2) Based on those assumptions, we identify steps where the operations effectively take place atomically, called *linearization points*. For example, when QUICKeR's SecurePut atomically updates the version number and PUTS a new ciphertext, the SecurePut operation from the perspective of other concurrent operations appears to have taken effect instantly. 3) Finally, based on the linearization points, we show that every concurrent execution has an equivalent sequential execution with the same behavior.

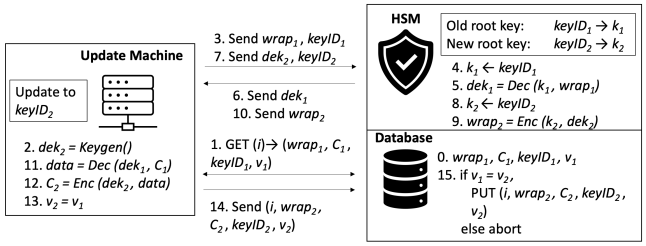

Figure 7: Naive Ciphertext Updates

4 NAIVE CIPHERTEXT UPDATES

QUICKeR's first method of updating the ciphertexts to new root and data keys is straightforward: update machines retrieve the old ciphertexts, re-encrypt, and finally store the updated ciphertext. Figure 7 shows the steps required for each ciphertext in the key-value store. The update machine (1) GETS the wrap $wrap_1$ and ciphertext C_1 from the database with the corresponding version number v_1 and root key id $keyID_1$ and (2) generates a new data key dek_2. Then in (3-6) the update machine communicates with the HSM to decrypt $wrap_1$ into old data key dek_1 and (7-10) encrypt dek_2 into new wrap $wrap_2$. Next, the update machine (11) decrypts C_1 into message data m and (12) encrypts m with dek_2 into ciphertext C_2. Finally, (13-15) the new wrap $wrap_2$ and ciphertext C_2 are atomically PUT in the database if the version number v_1 of this ciphertext has not changed and not been deleted, otherwise, the ciphertext update is aborted. (Step 13 shows v_1 renamed to v_2 to disambiguate the old version number from a possibly modified version number.)

Naive ciphertext updates provide better security than current practice, which at most updates wraps, by trivially provide post-compromise security of both data and root keys. Since the new ciphertext is encrypted by an entirely new root and data key, compromising previous keys provides an adversary no additional power to learn the plaintext information. Naive ciphertext updates also provides integrity guarantees by using an authenticated encryption scheme such as AES-GCM. In addition, naive ciphertext updates maintain conventional encryption schemes, so routine access operations of SecurePut and SecureGet do not incur extra latency from using a different encryption scheme. Naive ciphertext updates are also easy to integrate into current practice because it does not require significant modifications to client machine source code.

The main problem with naive ciphertext updates is that ciphertext updates can adversely affect the performance of routine access operations. Naive ciphertext updates require downloading and uploading many large ciphertexts. This consumes a significant portion of the database's network bandwidth, which is shared by concurrent routine access operations of SecurePut and SecureGet. Network bandwidth can become the primary bottleneck in databases, and it limits how frequently routine access operations and naive updates can concurrently execute. In our evaluation, we show experiments for exactly this: A linear tradeoff between the throughput of routine access operations and the throughput of ciphertext updates operations when network bandwidth is a bottleneck. Surprisingly, we also show that the high network costs of naive ciphertext updates lead to CPU bottlenecks, which affects routine access operations.

Updatable encryption solutions such as ReCrypt [11] avoid transferring entire ciphertexts and thus have significantly lower network costs by instead generating a small update token for the database to update the ciphertext.

5 UPDATABLE ENCRYPTION

Since naive ciphertext updates have a major performance bottleneck in network bandwidth, QUICKeR proposes an alternative ciphertext update process that enables direct updates of encrypted ciphertexts *in place on the database*, thus reducing bandwidth costs drastically.

5.1 Updatable Encryption Background

Much theoretical groundwork has introduced and studied updatable encryption [4–6, 11–13, 17, 18, 20, 24, 27]. To avoid the expensive costs of downloading and uploading ciphertexts for key rotation, the key idea behind updatable encryption is to upload a small update token that the database uses to locally update the ciphertext.

Updatable Encryption Definition: An updatable encryption scheme, UE, is composed of algorithms for a data owner and an outsourced storage server. The data owner wants to encrypt and store their data m on the storage server, and later update ciphertexts to a new key. Formally, an updatable encryption scheme is composed of the following five algorithms:

- $k \leftarrow \texttt{UE.KeyGen}(1^\lambda)$: First, the data owner locally runs UE.KeyGen to generate a key k with security parameter λ.
- $C \leftarrow \texttt{UE.Enc}(k, m)$: Then, the data owner locally encrypts its data m with k to compute a ciphertext C, which is stored on the storage server.
- $m \leftarrow \texttt{UE.Dec}(k, C)$: When the data owner wants to later access their data m, the owner first downloads their ciphertext C from the storage server and decrypts it with the current key k.
- $\Delta \leftarrow \texttt{UE.TokenGen}(k_1, k_2)$: To update the ciphertext to a new key, the data owner first generates a new key k_2 from UE.KeyGen. Next, the data owner executes UE.TokenGen to generate a small update token Δ, which is sent to the storage server to update the ciphertext to the new key k_2. Some updatable encryption schemes are *ciphertext-independent* [6, 13, 17, 18, 20, 24, 27], meaning that generating a token Δ does not depend on any information from the original ciphertext. Other schemes are *ciphertext-dependent* [4, 11] and require downloading a small portion of the ciphertext called a header C_{header} to generate a token $\Delta \leftarrow \texttt{UE.TokenGen}(k_1, k_2, C_{header})$.
- $C_2 \leftarrow \texttt{UE.Update}(\Delta, C_1)$: On receipt of a token Δ, the storage server executes UE.Update to generate a new ciphertext C_2, which is the encryption of data m by the new key.

An updatable encryption scheme guarantees correctness in that ciphertexts are decryptable by the current key and guarantees post-compromise security in that messages are confidential, even when past keys are compromised. Although prior work has shown that updatable encryption can guarantee post-compromise security even in the face of compromised keys [13, 27], there are other combinations of compromised update tokens and keys that inherently leak information. For instance, suppose an adversary has access to a ciphertext $C_1 \leftarrow \texttt{UE.Enc}(k_1, m)$, which is message m

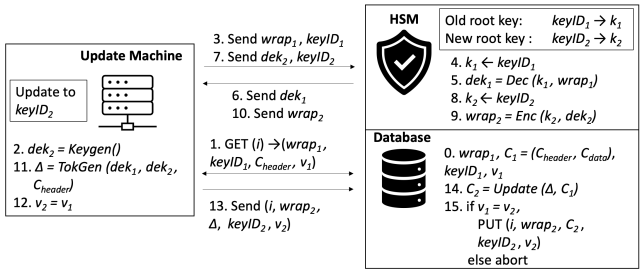

Figure 8: Updatable Encryption Approach

encrypted with key k_1. Also suppose that the adversary has compromised a future key k_2 and token $\Delta_{1,2} \leftarrow \texttt{UE.TokenGen}(k_1, k_2)$ was generated from keys k_1 and k_2. Then the adversary can always decrypt the message by first updating ciphertext C_1 from key k_1 to k_2 using $\Delta_{1,2}$ and then decrypting with key k_2 into message $m \leftarrow \texttt{UE.Dec}(k_2, \texttt{UE.Update}(\Delta_{1,2}, C_1))$. This unpreventable attack means that updatable encryption schemes require security assumptions that adversaries cannot compromise certain combinations of update tokens and keys whereas the naive ciphertext update approach avoids these security drawbacks. Despite these security assumptions, updatable encryption schemes are useful to study for their significantly lower network costs, which can lead to better overall performance.

5.2 An Updatable Encryption Approach

QUICKeR's second approach minimizes network overheads of updating ciphertexts by using updatable encryption. The protocol design capitalizes on the observation that the most expensive part of updating envelope encrypted ciphertexts is from updating old to new data keys. First, we choose to rotate root keys naively because downloading and uploading a wrap is inexpensive (at only 48 bytes). Second, we encrypt data and rotate data keys with an updatable encryption scheme. This approach is generalized for any updatable encryption scheme (including both ciphertext-dependent and ciphertext-independent schemes). Instead of downloading and uploading the full ciphertext, the update machine generates and sends an update token for the database to update the ciphertext from the old data key to the new data key. Since the wrap needs to be naively updated anyway, QUICKeR piggybacks uploading update token with a new wrap to reduce the number of network round trips and overall latency. For ciphertext-dependent updatable encryption schemes which require downloading a small header C_{header}, QUICKeR piggybacks downloading the header with the old wrap.

QUICKeR's updatable encryption approach replaces the plaintext data encryption in SecurePut and SecureGet with an updatable encryption scheme to efficiently update ciphertexts. The ciphertext update operation for updating an entry with index i to a new root key with root key id $keyID_2$ using updatable encryption is described in Figure 8. An update machine (1) GETS the wrap $wrap_1$, ciphertext header C_{header} (for ciphertext-dependent encryption schemes), root key id $keyID_1$, and version number v_1 from the database for index i and (2) generates a new data key dek_2. Then, (3-6) the update machine communicates with the HSM and uses $keyID_1$

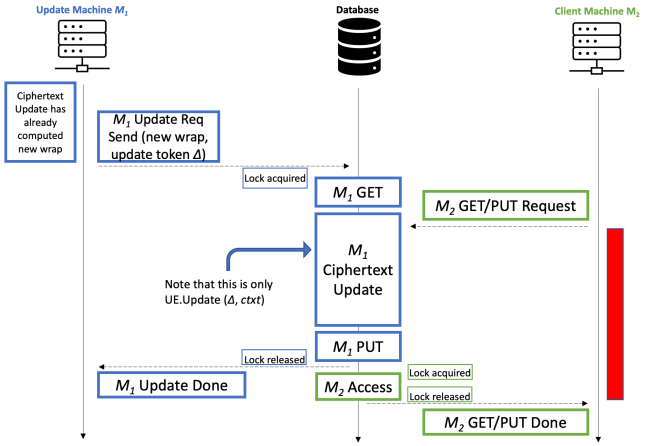

Figure 9: Ciphertext update operations on the database monopolize the single thread of execution, blocking other routine access operations until the update completes.

to decrypt $wrap_1$ into data key dek_1 and (7-10) uses $keyID_2$ to encrypt dek_2 to a new wrap $wrap_2$. Next, (11) the update machine generates an update token $\Delta = \texttt{UE.TokenGen}(dek_1, dek_2, C_{header})$. Finally, (12-13) the update machine sends the index i, token Δ, new wrap $wrap_2$, new root key id $keyID_2$, and version number v_1 (which is renamed to v_2 in the figure) to the database, which (14-15) first generate $C_2 = \texttt{UE.Update}(\Delta, C_1)$ and then atomically PUTS the new ciphertext C_2, the new wrap $wrap_2$, and root key id $keyID_2$ if the version number v_1 hasn't changed.

QUICKeR's updatable encryption approach provides post-compromise security and integrity guarantees because updatable encryption schemes are designed to provide both guarantees (with some assumptions for post-compromise security). In terms of impact on concurrent routine access operations, the updatable encryption approach improves on the naive ciphertext update approach by significantly reducing the bandwidth overhead of updates from the size of an entire ciphertext to merely bytes: A key wrap is small (only 48 bytes) and ciphertext headers for ciphertext-dependent encryption schemes are also inexpensive. The update machine need only upload a new wrap and an update token to the database. Whereas the naive ciphertext update approach is primarily bottlenecked by the database's bandwidth, which significantly trades throughput of routine access operations for the throughput of ciphertext updates, the updatable encryption approach eliminates bandwidth as a primary bottleneck. Therefore, this reduces the tradeoff between the performance of routine access operations and the size of the vulnerability window.

Unfortunately, a straightforward application of updatable encryption would still adversely impact the performance of routine access operations. Since many databases are often designed to synchronize operations, databases that execute updatable encryption's update computation `UE.Update` block concurrent GET and PUT operations by obtaining exclusive locks until the computationally expensive ciphertext update operation completes. These long update

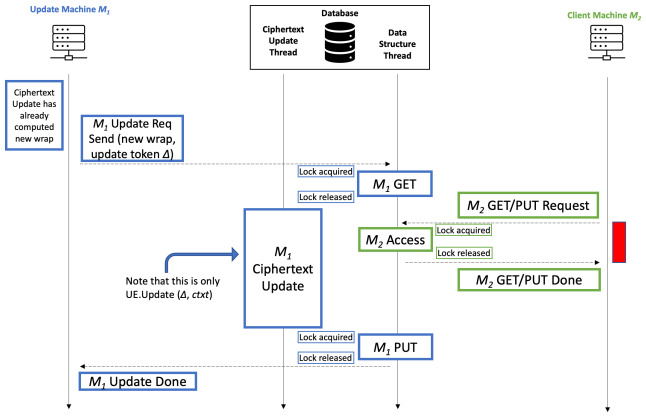

Figure 10: Only the data accesses (GET and PUT) are synchronized. The update computation is performed on a separate thread so that other concurrent GET or PUT transactions can execute.

operations monopolize the single thread of operation execution,[3] blocking and queuing up other routine access operations that GET or PUT data. As shown in Figure 9, a typical database performing an update blocks a concurrent routine access operation until after the update completes. Increasing the frequency of updates would therefore detrimentally impact the performance of routine access operations.

5.3 Multithreaded Updatable Encryption

QUICKeR's third approach eliminates the problem of updatable encryption monopolizing the database's single thread of transaction execution by multithreading parts of the ciphertext update operation. Instead of locking when the database executes `UE.Update` and blocking other GET and PUT requests, QUICKeR only locks in the portion of the ciphertext update that needs to GET old data or PUT updated data. The ciphertext update operation can be broken down to three steps: 1) the database GETS the old ciphertext data, 2) the database performs `UE.Update`, which computes an updated ciphertext using the old ciphertext data and provides the update token, and 3) the database PUTS the updated ciphertext data, which includes a ciphertext, wrap, root key id, and version number. Locking during the GET and PUT in steps 1 and 3 is necessary to ensure that the database maintains correctness with other operations. However, the database executes `UE.Update` (step 2) on a separate thread to ensure that `UE.Update` doesn't block other concurrent database operations. Figure 10 shows how this multithreading approach avoids impacting routine access operations. When an update machine sends an update token and wrap to update a ciphertext to a new key, only the initial ciphertext data access and PUT are locked. This avoids blocking concurrent access operations during `UE.Update`.

Overall correctness of the multithreaded updatable encryption approach is still guaranteed using the techniques discussed in Section 3.2 and 3.3. The approach has the same security guarantees as the single-threaded updatable encryption approach. Finally, since it no longer monopolizes the database's single thread of execution, frequent ciphertext updates no longer severely impact the throughput of routine access operations. As we show in our evaluation, the multithreading approach has the best performance.

[3] Although multithreaded databases do process network I/O and other miscellaneous actions in separate threads, they often still lock data items until completion.

5.4 Choice of Updatable Encryption Scheme

We choose the best updatable encryption scheme by considering their security guarantees and conducting updatable encryption microbenchmarks in our evaluation (Section 6.1). We compare three ciphertext-dependent updatable encryption schemes (ReCrypt [11], Nested UAE [4], and KH-PRF [4]) and one ciphertext-independent encryption scheme (SHINE [6]).

First, we consider the security of different schemes. The four cryptographic schemes offer similar security guarantees, assuming no trivial wins. All four provide integrity and confidentiality for both messages and updates, but there are nuanced differences:

- **Confidentiality:** All four schemes ensure message and update confidentiality, which encompasses post-compromise security. While ReCrypt satisfies update and *chosen-plaintext* (CPA) security, KH-PRF and Nested UAE provide stronger update and CPA security in which updated ciphertexts are indistinguishable from newly encrypted ciphertexts. However, Nested UAE only guarantees security up to a limited number of updates, which we explain in greater detail later. Lastly, SHINE satisfies an even stronger definition for ciphertext-independent schemes which additionally defends against *chosen-ciphertext attacks* (CCA).
- **Integrity:** While all four provide integrity guarantees, Nested UAE satisfies a stronger integrity guarantee than ReCrypt and KH-PRF. In Nested UAE, adversaries can also update malformed ciphertexts without compromising integrity [4]. SHINE satisfies an analogous (though modified) integrity guarantee to Nested UAE's integrity guarantee for ciphertext-independent schemes.
- **Token Security:** All four schemes also share a vulnerability when tokens and keys are compromised. Suppose that an adversary has access to recently updated ciphertext $C_2 \leftarrow \texttt{UE.Update}(\Delta_{1,2}, C_1)$ from a ciphertext $C_1 \leftarrow \texttt{UE.Enc}(k_1, m)$ and update token $\Delta_{1,2} \leftarrow \texttt{UE.TokenGen}(k_1, k_2, C_{header})$ or $\Delta_{1,2} \leftarrow \texttt{UE.TokenGen}(k_1, k_2)$ in the case of SHINE. Also suppose that the adversary has access to old key k_1 and token $\Delta_{1,2}$. In all four schemes, access to update token $\Delta_{1,2}$ allows the adversary to downgrade ciphertext C_2 to C_1 and then decrypt it with k_1, breaking security.

From this analysis, we conclude that all four schemes have comparable security, with ReCrypt having the weakest security guarantees of the four. Next, we compare the performance of the four updatable encryption schemes. From the microbenchmark experiments (Table 1), we determined that Nested UAE had the best performance. In addition, downloading Nested UAE's ciphertext header C_{header} (84 bytes) and uploading Nested UAE's token $\Delta_{1,2}$ (184 bytes) are inexpensive to piggyback with downloading and uploading wraps. For performance and security reasons, we choose to use Nested UAE in QUICKeR.

When applied to QUICKeR, Nested UAE provides post-compromise security with two caveats: (1) As previously mentioned, compromised update tokens along with certain keys can leak information. Although recent works have proposed updatable encryption schemes that do not have this vulnerability [13, 27], they rely on public key encryption which is significantly more computationally expensive, especially for large ciphertexts. Due to the available attacks, the updatable encryption approach has an extra security assumption that adversaries cannot obtain update tokens to downgrade ciphertexts to a compromised key, which the naive approach does not have. To help prevent update token compromises, tokens need to be deleted from the database as soon as updates complete. (2) Nested UAE can only guarantee security until the number of ciphertext updates exceed a configurable number. The configurable number trades off the size of the ciphertext and longer decryption time for more updates, so QUICKeR sets the maximum number of ciphertext updates to a reasonable number (such as 64). Since Nested UAE updates ciphertexts to a new key by encrypting a new layer over the old ciphertext, decryption time scales linearly with the number of times a ciphertext has been updated. When the number of updates run out, QUICKeR defaults to a naive ciphertext update to refresh the number of updates. Despite naively updating ciphertexts once the number of updates run out, the updatable encryption approach can decrease the impact ciphertext updates have on performance of routine access operations, which we show in our experimental results.

6 EVALUATION

We evaluate the practicality of routine operations and concurrent key rotation and ciphertext updates. The main questions we aim to answer are as follows:

- Which updatable encryption schemes provide the least performance penalty compared to naive encryption and would be most suitable QUICKeR?
- What are the potential bottlenecks of key rotation?
- What impact do key rotations have on routine operations?
- How well does QUICKeR scale to more data?

To answer each of these questions, we present three different experiments: 1) updatable encryption microbenchmarks 2) bottleneck experiments and 3) end-to-end experiments. We first run updatable encryption microbenchmarks to determine which updatable encryption scheme has the least overhead compared to naive schemes and is hence most suitable for QUICKeR. Next, we examine the potential bottlenecks of updating ciphertexts on a database with a simplified experimental setup. The simpler experimental setup enables us the opportunity to isolate factors that might bottleneck the database. Finally, we share experimental results from our end-to-end implementation on different workloads and conduct experiments to examine QUICKeR's scalability with different amounts of data.

6.1 Microbenchmark Experiments

We compare the performance of several recent updatable encryption schemes over different plaintext sizes to determine the schemes with the least overhead compared with a naive AES-GCM encryption scheme. The updatable encryption schemes we compared are SHINE [6], ReCrypt [11], Nested UAE [4], and KH-PRF [4]. Like the four updatable encryption schemes, the naive encryption scheme also provides message integrity guarantees. For the KH-PRF scheme, we use the best performing Ring Learning With Errors parameters with modulus size $|q| = 60$. We set the Nested UAE scheme (which only does a limited number of key rotations) to allow up to 64 updates. Both ReCrypt and SHINE rely on elliptic curve cryptography; however, the implementation of ReCrypt utilizes Curve25519 whereas SHINE employs secp256k1. We ran these microbenchmarks on a `t2.medium` machine.

Keygen

Naive Encryption	SHINE [6]	ReCrypt [11]	Nested UAE [4]	KH-PRF [4]
3.1×10^{-6}	6.58×10^{-5}	1.20×10^{-5}	3.28×10^{-6}	3.50×10^{-6}

Encryption

Ptxt Size	Naive Encryption	SHINE [6]	ReCrypt [11]	Nested UAE [4]	KH-PRF [4]
1 KB	3.31×10^{-6}	5.12×10^{0}	1.73×10^{-1}	1.02×10^{-6}	2.44×10^{-4}
10 KB	5.85×10^{-6}	1.32×10^{1}	1.64×10^{0}	1.28×10^{-5}	2.86×10^{-4}
100 KB	3.19×10^{-5}			3.99×10^{-5}	2.02×10^{-3}
1 MB	2.90×10^{-4}			3.02×10^{-4}	1.97×10^{-2}
10 MB	2.99×10^{-3}			2.99×10^{-3}	1.96×10^{-1}

Decryption

Ptxt Size	Naive Encryption	SHINE [6]	ReCrypt [11]	Nested UAE [4]	KH-PRF [4]
1 KB	1.72×10^{-6}	8.37×10^{-2}	1.62×10^{-1}	6.28×10^{-5}	1.65×10^{-4}
10 KB	1.00×10^{-5}	8.24×10^{-1}	1.54×10^{0}	1.38×10^{-4}	2.12×10^{-4}
100 KB	8.53×10^{-5}			1.37×10^{-4}	1.88×10^{-3}
1 MB	7.96×10^{-4}			9.42×10^{-3}	1.85×10^{-2}
10 MB	8.03×10^{-3}			1.02×10^{-1}	1.84×10^{-1}

Token Gen

SHINE [6]	ReCrypt [11]	Nested UAE [4]	KH-PRF [4]
6.07×10^{-5}	4.90×10^{-3}	8.13×10^{-5}	8.88×10^{-5}

Ciphertext Update

Ptxt Size	SHINE [6]	ReCrypt [11]	Nested UAE [4]	KH-PRF [4]
1 KB	1.56×10^{-2}	1.52×10^{-1}	5.47×10^{-6}	1.66×10^{-4}
10 KB	1.44×10^{-1}	1.49×10^{0}	8.38×10^{-6}	1.79×10^{-4}
100 KB			2.09×10^{-5}	1.60×10^{-3}
1 MB			3.26×10^{-4}	1.61×10^{-2}
10 MB			3.39×10^{-3}	1.60×10^{-1}

Table 1: Updatable Encryption Microbenchmarks (in seconds per operation)

Table 1 shows microbenchmark results for different plaintext sizes. We did not obtain results for SHINE and ReCrypt at plaintext sizes of 100 KB and greater because the schemes did not efficiently scale to those sizes. The microbenchmarks show that Nested UAE [4] has the least performance cost compared to a naive encryption scheme. Nested UAE has a small latency penalty when compared to naive encryption and is the updatable encryption scheme that we use for the remaining experiments. The microbenchmarks also suggest that ReCrypt, an updatable encryption scheme which updates both root and data keys, is too computationally expensive for typical data sizes of 32 KB and 8 MB [7] when compared to naive encryption.

6.2 Bottleneck Experiments

In these set of experiments, we seek to understand how different ciphertext update approaches perform under different performance

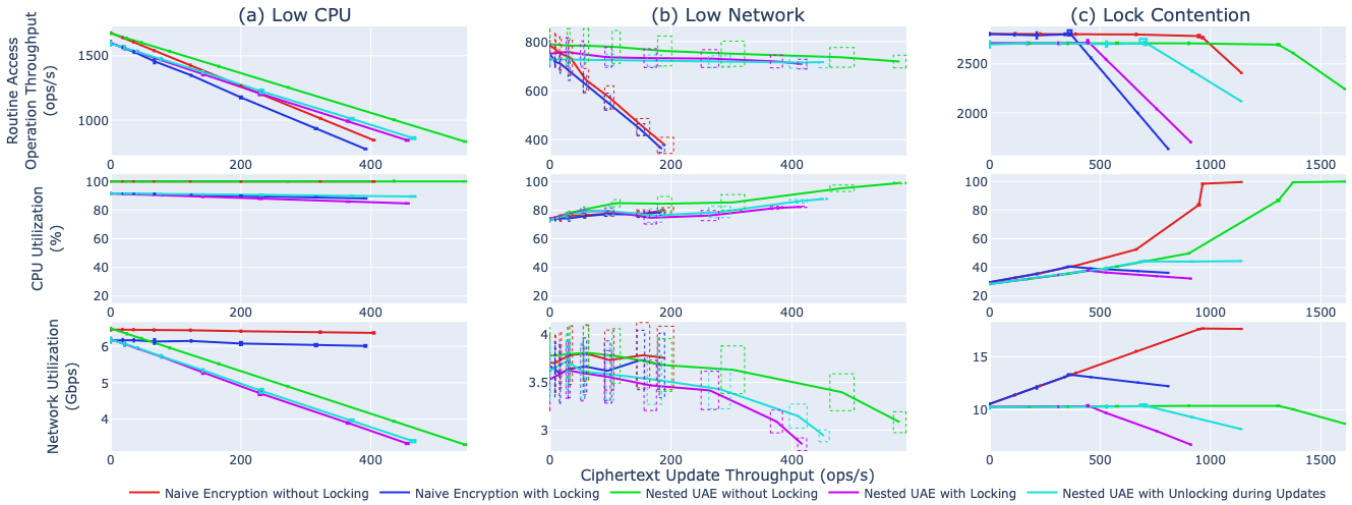

Figure 11: Bottleneck Experiments

bottlenecks. We consider three potential bottlenecks on the database: **(a)** low CPU, **(b)** low network bandwidth, and **(c)** lock contention. Each of these three bottlenecks can inadvertently arise in a real system, and the goal of these experiments is to understand how different update approaches affect the performance of routine operations.

6.2.1 Experimental Setup. We implemented this experimental setup in C++ consisting of one update machine, one client machine, and one database server. This is not an end-to-end QUICKeR system, does not include a HSM, and does not attempt to guarantee correctness to test the effects of locking. To isolate the database bottlenecks, we kept the client and update machines on a `r5n.8xlarge` instance (so each has plenty of bandwidth and CPU) and changed the database machine to have different CPU and network bandwidth resources. The database machines for **(a)** low CPU, **(b)** low network bandwidth, **(c)** lock contention were `r5n.2xlarge`, `t3.2xlarge`, and `r5n.8xlarge` instances, respectively. During the experiments, the client machine has 45 threads connected to 45 database threads that continuously encrypt and PUT or GET uniformly at random and decrypt (for convenience, we'll call them routine access operations) a 1 MB plaintext. Simultaneously, the update machine has a number of threads (0, 1, 2, 4, 8, 15, 30, or 45) connected to the database continuously updating a same-sized ciphertext. Modulating the number of update threads helps us understand how the frequency of updates affects the throughput of routine operations and network and CPU loads on the database. We examine how locking affects performance by testing naive updates with and without database locks, and Nested UAE [4] with database locks, without locks, and unlocking during updates (as described in Section 5.3). We ran these experiments over a 200 second experiment period and seeded the randomness so different experiments can be better compared. We ran each experiment 30 times for different random seeds and report 99% confidence intervals.

6.2.2 Experimental Results. The experimental results are shown in Figure 11 for **(a)** low CPU, **(b)** low network bandwidth, and **(c)** lock contention. The graphs show how ciphertext update throughput on the x-axis affects 1) routine access operation throughput, 2) CPU utilization, and 3) network utilization on the y-axis. 99% confidence intervals for both the x and y axes are depicted in the graphs as boxes. Most data points have such small confidence intervals that they are barely visible points. A scheme shows better performance when increasing the throughput of ciphertext updates does not decrease the throughput of routine access operations. In the graphs, the schemes extend to different peak ciphertext update throughputs (on the x-axis) because the ciphertext update throughput is determined by the number of update machine threads.

When **(a)** CPU is the primary bottleneck, we were expecting to see the naive encryption schemes perform better since using updatable encryption would require CPU. However, we instead saw that increasing the throughput of ciphertext updates steeply decreases the routine access operation throughput for both kinds of schemes. This is explainable because handing a high network workload is surprisingly CPU intensive, resulting in a tradeoff between routine access operation throughput and update throughput.

When **(b)** the network bandwidth is the primary bottleneck, both naive encryption schemes (with and without locking) suffer the same linear throughput loss in routine access operations. The Nested UAE schemes did not suffer the same tradeoffs as the naive schemes did because network bandwidth was not the limiting factor. We remark that these experiments had the largest confidence intervals because the database was run on a burstable instance with occasionally higher performance.

When neither CPU nor network are bottlenecks, then **(c)** lock contention can become the primary bottleneck. These experiments show that the schemes with no locking perform better than all other schemes. Following that, unlocking during Nested UAE updates performs better than Nested UAE which locks the entire update. Note that the CPU utilization for the schemes that lock remains low at around 35%, meaning that the CPU is waiting for locks whereas the CPU utilization for the schemes that have no locking rise to full utilization.

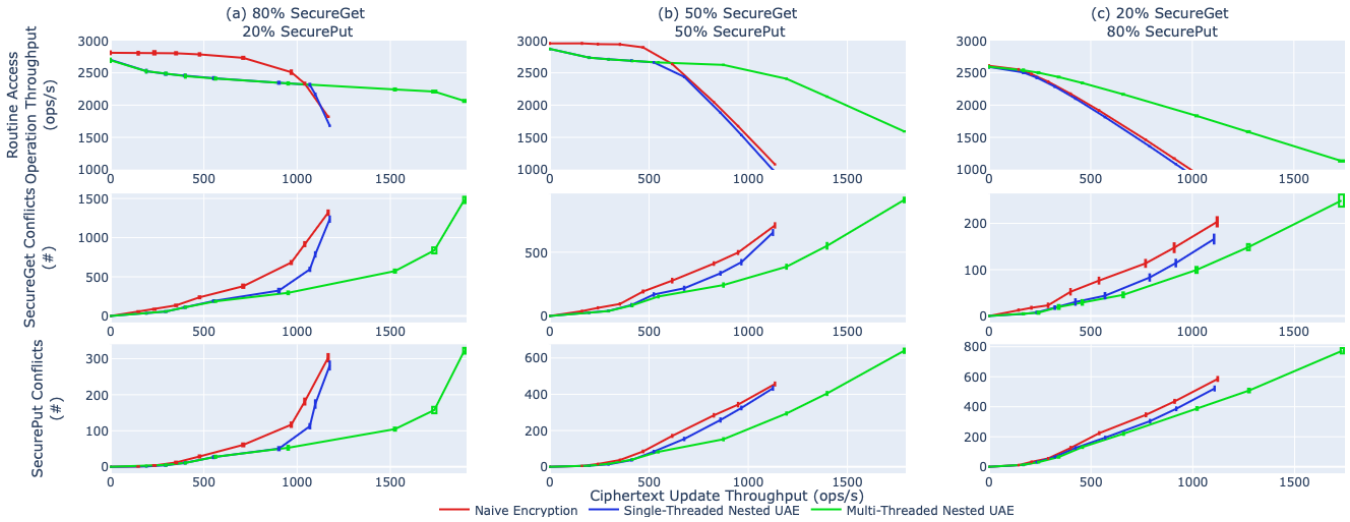

Figure 12: End-to-end Experiments

6.3 End-to-end Experiments

In these set of experiments, we seek to understand QUICKeR's end-to-end performance under different workloads and different update throughputs.

6.3.1 Experimental Setup. We developed an end-to-end implementation of QUICKeR in C++. In this experiment, we have a client, an update machine that interact with our database server. The client and update machines interface with three HSMs to encrypt and decrypt data encryption keys. The client machine, update machine, and database server are all `r5n.8xlarge` instances. We first initialize our database with 20000 indices (key-value pairs) of encrypted 1 MB plaintext. Our client machine connects to the database on 45 threads and continuously performs SecurePut and SecureGet operations over indices randomly sampled from a Zipfian distribution. We disregard insertion and deletion of indices and focus on SecurePut for existing indices since insertion and deletion do not substantially change our results. Simultaneously, the update machine connects with the server on different thread counts (0, 1, 2, 4, 8, 15, 30, 45, and 90) and continuously updates ciphertexts in the database in rounds. We compared naive encryption, single-threaded Nested UAE, and multi-threaded Nested UAE over different ratios of SecureGet and SecurePut (80% and 20%, 50% and 50%, and 20% and 80% SecureGet and SecurePut, respectively). Over a 200 second experiment period, we measure the throughput of operations, the number of concurrent SecureGet and ciphertext update operations over the same index, and the number of concurrent SecurePut and ciphertext updates over the same index. We ran each experiment 30 times and seed the randomness appropriately.

6.3.2 Experimental Results. The results for these experiments are shown in Figure 12 with bounding boxes representing 99% confidence intervals for both the x and y axes. The graphs show different peak ciphertext update throughputs (on the x-axis) for different schemes because the ciphertext update throughput is determined by the number of update threads. The graphs for ciphertext update

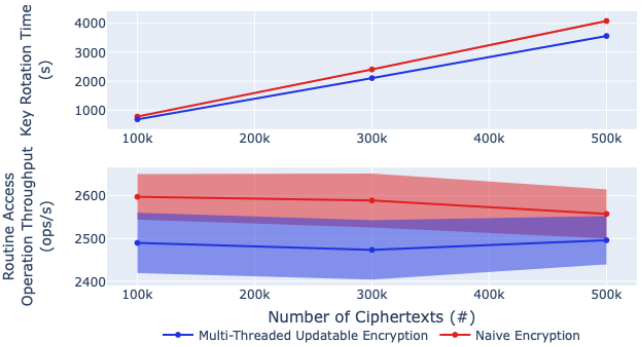

Figure 13: Scalability Experiment

throughput vs routine access operation throughput show that naive encryption initially performs better than the updatable encryption approaches because of the extra latency of updatable encryption. Nested UAE has higher decryption latency when a ciphertext has been updated more (because of needing to decrypt each additional layer). Hence, our results show that the two updatable encryption schemes perform better during higher workloads of SecurePut, which also resets the number of updates on a ciphertext. At higher update throughput, the multi-threaded updatable encryption approach performs the best because it avoids holding locks during the update computation.

All three schemes showed generally the same amount of concurrent SecureGet and ciphertext updates as well as concurrent SecurePut and ciphertext updates (labeled in Figure 12 as SecureGet conflicts and SecurePut conflicts), with the multi-threaded updatable encryption scheme having more conflicts because it supports a higher throughput updates. There are generally more SecureGet conflicts because we implemented QUICKeR with read and write locks that allow multiple concurrent reads but only a single writer.

6.4 Scalability Experiment

Finally, we ran experiments to determine how QUICKeR scales with the size of the database.

6.4.1 Experimental Setup. Using our end-to-end system, we had an update machine, client machine, and database server each on `r5n.24xlarge` instances and 3 HSMs. The database server is initialized with different amounts of data to measure scalability which are 100,000, 300,000, and 500,000 ciphertexts of 1 MB plaintext data, totalling 100, 300, and 500 GB of data. The update machine has one thread continuously performing ciphertext updates and simultaneously the client machine has 45 threads continuously performing 50% SecurePut and 50% SecureGet operations on indices sampled from a zipfian distribution. We ran these experiments 30 times for naive encryption and multi-threaded updatable encryption with seeded randomness for 200 seconds and calculated the key rotation time.

6.4.2 Experimental Results. Figure 13 show how the key rotation time and routine operation throughput on the y-axis changes with the number of indices on the x-axis. 99% confidence intervals are depicted for the y-axis on both graphs. The top graph has barely visible confidence intervals because the intervals are smaller than 35 seconds for all points. The top graph demonstrates that for both schemes the key rotation time increases linearly with the amount of data (as we would expect). Meanwhile, the bottom graph shows that the throughput of routine access operations for both schemes doesn't change significantly with the number of indices. Together, these graphs show that QUICKeR scales well for higher data amounts. Lastly, although the updatable encryption scheme can perform a round of key rotation faster, the updatable encryption scheme impacts the throughput of routine access operations more due the algorithm's latency.

In the context of a database at the scale of a petabyte, we argue that QUICKeR's performance is practical with these results. This experiment shows that using a single thread, QUICKeR can rotate keys for 500 GB in less than 70 minutes with the naive scheme and around 1 hour with the multi-threaded updatable encryption scheme. In addition, our single-threaded updatable encryption scheme would likely perform similarly to multi-threaded updatable encryption scheme in this experiment based on our end-to-end experiments (Figure 12) showing similar performance for low update throughputs. A database at the scale of a petabyte would necessarily be sharded into smaller database servers that can each independently use QUICKeR's system to update ciphertexts. For instance, suppose there are 2000 database servers each hosting 500 GB of data (a petabyte total), then a conservative estimate for all three schemes would only require 4 QUICKeR update machines continuously running a single thread to update an entire petabyte of plaintext data in one month. Without considering transactional processing and replication, this suggests that all three of QUICKeR's approaches are practical.

7 CONCLUSION

In this work, we introduced QUICKeR, a system for frequent key rotations and ciphertext updates. QUICKeR improves the standard cloud deployment for encrypting data at-rest by minimizing the vulnerability windows of ciphertexts. QUICKeR operates in an online manner where key rotations and ciphertext updates occur concurrently with routine access operations. We achieved this by introducing modifications to SecureGet, SecurePut, and ciphertext updates to provide linearizability guarantees. We proposed three approaches to performing frequent ciphertext updates with tradeoffs in security assumptions and performance. Our experimental evaluation show how different performance bottlenecks affect naive and updatable encryption differently. Our end-to-end implementation and scalability experiments show that QUICKeR can rotate keys and update ciphertexts efficiently without adversely affecting routine operations and scale to real world data sizes.

7.1 Limitations and Future Work

In this paper, we assumed a simplified database backend with a single, multi-threaded key-value store that ensures correctness by locking data items until completion. Future directions can explore applying QUICKeR to distributed and transactional processing systems. For instance, QUICKeR does not solve transactional processing systems with underlying security flaws such as multiversion concurrency control leaving old ciphertexts on the database. Further research need also examine the challenges of ciphertext updates amidst replication.

7.2 Ciphertext Unlinkability

Many updatable encryption schemes are also designed to provide ciphertext *unlinkability*. Unlinkability means that an adversary would be unable to link an old ciphertext to its updated ciphertext. QUICKeR trivially cannot provide unlinkability because database indices link an old ciphertext to a new ciphertext. Future work can explore providing unlinkability from a system perspective, which requires eliminating indices, such as with ORAM [31].

ACKNOWLEDGMENTS

This work is supported in part by a 2021 UCSB Academic Senate Grant, a UCSB Early Career Faculty Acceleration Award, DARPA under agreement number HR001118C0060, National Science Foundation under Grant No. 2126327, and AWS Cloud Credit for Research program. We would like to thank the anonymous reviewers and shepherd whose insightful feedback significantly improved this paper.

REFERENCES

[1] Amazon Web Services, Inc. 2018. *AWS Key Management Service Cryptographic Details*. Amazon Web Services, Inc. https://d1.awsstatic.com/whitepapers/KMS-Cryptographic-Details.pdf.

[2] Elaine. Barker, National Institute of Standards, and Technology (U.S.). 2020. *Recommendation for key management. Part 1: General* (5 ed.). National Institute of Standards and Technology, Technology Administration [Gaithersburg, MD]. 158 pages.

[3] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (dec 1983), 465–483. https://doi.org/10.1145/319996.319998

[4] Dan Boneh, Saba Eskandarian, Sam Kim, and Maurice Shih. 2020. Improving Speed and Security in Updatable Encryption Schemes. Cryptology ePrint Archive, Report 2020/222. https://ia.cr/2020/222.

[5] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. 2013. Key Homomorphic PRFs and Their Applications. In *Advances in Cryptology – CRYPTO 2013*, Ran Canetti and Juan A. Garay (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 410–428.

[6] Colin Boyd, Gareth T. Davies, Kristian Gjøsteen, and Yao Jiang. 2020. Fast and Secure Updatable Encryption. In *Advances in Cryptology – CRYPTO 2020*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer International Publishing, Cham, 464–493.

[7] Google Cloud. 2020. *Encryption at rest in Google Cloud*. Technical Report. https://services.google.com/fh/files/misc/security_whitepapers_march2018.pdf.

[8] Google Cloud. 2022. Cloud Key Management. https://cloud.google.com/security-key-management. Accessed: 2022-05-11.

[9] Google Cloud. 2022. Envelope Encryption. https://cloud.google.com/kms/docs/envelope-encryption. Accessed: 2022-05-26.

[10] IBM Cloud. 2022. Key Protect. https://cloud.ibm.com/catalog/services/key-protect. Accessed: 2022-05-11.

[11] Adam Everspaugh, Kenneth Paterson, Thomas Ristenpart, and Sam Scott. 2017. Key Rotation for Authenticated Encryption. In *Advances in Cryptology – CRYPTO 2017*, Jonathan Katz and Hovav Shacham (Eds.). Springer International Publishing, Cham, 98–129.

[12] Andrés Fabrega, Ueli Maurer, and Marta Mularczyk. 2021. A Fresh Approach to Updatable Symmetric Encryption. Cryptology ePrint Archive, Report 2021/559. https://ia.cr/2021/559.

[13] Yao Jiang Galteland and Jiaxin Pan. 2022. Backward-Leak Uni-Directional Updatable Encryption from Public Key Encryption. Cryptology ePrint Archive, Report 2022/324. https://ia.cr/2022/324.

[14] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972

[15] Microsoft Inc. 2022. Best practices for secrets management in Key Vault. https://docs.microsoft.com/en-us/azure/key-vault/secrets/secrets-best-practices. Accessed: 2022-05-11.

[16] Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. 2019. Updatable Oblivious Key Management for Storage Systems. Cryptology ePrint Archive, Report 2019/1275. https://ia.cr/2019/1275.

[17] Yao Jiang. 2020. The Direction of Updatable Encryption Does Not Matter Much. In *Advances in Cryptology – ASIACRYPT 2020*, Shiho Moriai and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 529–558.

[18] Michael Klooß, Anja Lehmann, and Andy Rupp. 2019. (R)CCA Secure Updatable Encryption with Integrity Protection. In *Advances in Cryptology – EUROCRYPT 2019*, Yuval Ishai and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 68–99.

[19] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (jun 1981), 213–226. https://doi.org/10.1145/319566.319567

[20] Anja Lehmann and Björn Tackmann. 2018. Updatable Encryption with Post-Compromise Security. In *Advances in Cryptology – EUROCRYPT 2018*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 685–716.

[21] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[22] Olivier Markowitch, Liran Lerman, and Gianluca Bontempi. 2011. Side channel attack: an approach based on machine learning.

[23] Michael Meli, Matthew R. McNiece, and Bradley Reaves. 2019. How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. *Proceedings 2019 Network and Distributed System Security Symposium* (2019).

[24] Peihan Miao, Sikhar Patranabis, and Gaven Watson. 2022. Unidirectional Updatable Encryption and Proxy Re-encryption from DDH or LWE. Cryptology ePrint Archive, Report 2022/311. https://ia.cr/2022/311.

[25] Steven Myers and Adam Shull. 2017. Efficient Hybrid Proxy Re-Encryption for Practical Revocation and Key Rotation. Cryptology ePrint Archive, Paper 2017/833. https://eprint.iacr.org/2017/833 https://eprint.iacr.org/2017/833.

[26] National Institute of Standards and Technology 2019. *FIPS Security Requirements for Cryptographic Modules*. National Institute of Standards and Technology.

[27] Ryo Nishimaki. 2022. The Direction of Updatable Encryption Does Matter. In *Public-Key Cryptography – PKC 2022*, Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe (Eds.). Springer International Publishing, Cham, 194–224.

[28] PCI Security Standards Council 2022. *Payment Card Industry (PCI) Data Security Standard*. PCI Security Standards Council. v4.0.

[29] Mark Randolph and William Diehl. 2020. Power Side-Channel Attack Analysis: A Review of 20 Years of Study for the Layman. *Cryptography* 4, 2 (2020). https://doi.org/10.3390/cryptography4020015

[30] Thomas Roche, Victor Lomné, Camille Mutschler, and Laurent Imbert. 2021. A Side Journey To Titan. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 231–248. https://www.usenix.org/conference/usenixsecurity21/presentation/roche

[31] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.

[32] Patrick Stewin and Iurii Bystrov. 2012. Understanding DMA Malware. In *DIMVA*.

[33] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1296–1310. https://doi.org/10.1109/SP.2019.00009

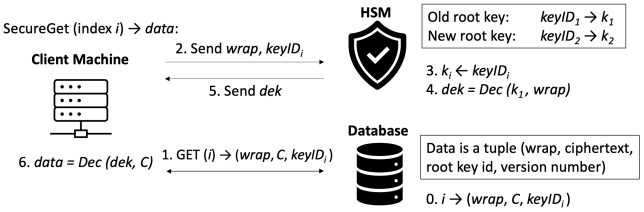

Figure 14: Abstract Implementation for SecureGet

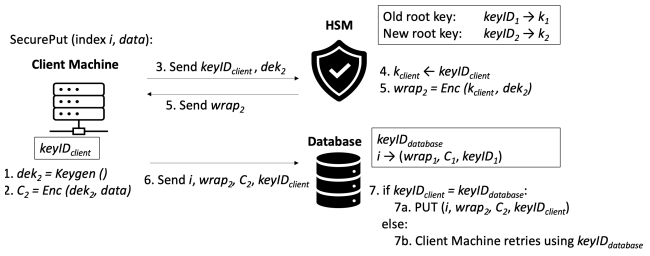

Figure 15: Abstract Implementation for SecurePut

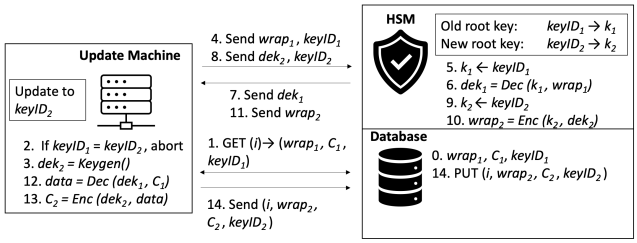

Figure 16: Abstract Implementation for Naive Key Rotation

A LINEARIZABILITY PROOF

We prove that QUICKeR satisfies linearizability correctness. In essence, linearizability means that operations appear to take effect instantaneously in time [14]. This implies that concurrent operations do not lead to wrong execution results. We first define several terms and then formally prove that concurrent executions in QUICKeR are linearizable by showing its equivalence to a sequential execution.

Definition 1: Concurrent operation executions are modelled with a *history* of operations. A history is a finite sequential order of events, which are either *invocation events* or *response events*. An *invocation event* represents when an operation begins and a *response event* represents when an operation completes. Invocation and response events are labeled by its process ID, operation and arguments. A history is *valid* if for all response events there exists a unique, matching invocation event labeled the same preceding the response event in the history.

Definition 2: A *history H* is *sequential* if (1) the first event is an invocation event and (2) all invocation events are immediately followed by a matching response.

Definition 3: Two operations are *concurrent* if the operations overlap. Two operations do not overlap if one operation's invocation event occurs after the response event of the other operation; otherwise, the operations overlap.

Definition 4: An implementation is *linearizable* if for every *history* of operations there exists a sequential history, called a *linearization*, where (1) the order of non-overlapping operations are maintained and (2) the behavior of the implementation matches a sequential history for an abstract implementation. (1) means that if an invocation event for some operation comes after the response event (or the completion) of another operation, then a linearization needs to preserve this quality. (2) means that the implementation achieves the same result as a sequential execution.

We define abstract implementations for SecureGet, SecurePut, and naive ciphertext rotation in Figures 14, 15, and 16, respectively. Note that in the abstract implementation of key rotation, it is not necessary to update a ciphertext which is already encrypted by the new root key for the round, so the operation aborts in that case. Assume without loss of generality that QUICKeR is rotating from root key k_i to k_{i+1}, then we prove the following:

Theorem A.1. *For all valid histories H of concurrent executions of SecureGet, SecurePut, and ciphertext updates in QUICKeR, there exists a linearization where the order of non-overlapping operations in H is maintained and the behavior of QUICKeR matches the behavior of the sequential history of the abstract implementation.*

Proof: We prove through forward simulation that given a history H, there exists a linearization L that maintains H's partial order for QUICKeR. Take Figures 4, 6, and 7 as QUICKeR's implementations of SecureGet, SecurePut, and (without loss of generality) ciphertext updates, respectively, where each operation is composed of multiple atomic steps to complete the operation. (For instance, the first step for SecureGet (Figure 4) is to atomically get the wrap, ciphertext and root key ID.) Our forward simulation works as follows: sequentially iterate through each event in H. On an invocation event, add the operation to a pending operation list. On a response event, simulate the possible orders of the atomic steps in the pending operation list to mark the completion of this event and move on to the next one. To show linearizability correctness, we prove that for any history H and any order of steps, the results are the same as a linearization of the abstract implementations.

We identify *linearization points* in QUICKeR's implementation for all three operations. A linearization point represents when the operation appears to have been instantaneously completed to other concurrent operations in the linearization. The linearization point of SecureGet is in step 1 when the client machine gets the wrap, ciphertext, and root key ID. The linearization point for SecurePut is in step 7a when the wrap, ciphertext, root key ID, and version number are atomically put into the database. Finally, the linearization point for naive ciphertext updates is in step 15 when either the new ciphertexts are atomically put into the database or the operation is aborted. As each of these linearization points of these operations are simulated, the invocation and response event of the corresponding abstract implementation is added to the linearization L.

We now prove that for any history H and order of steps results in linearization L. Observe that the primary shared resource is the data access and storage on the database which cause race conditions. (Note that encrypting or decrypting wraps on the HSM does

not change the root key.) For SecureGet (Figure 4), the only step that causes races conditions is step 1. During this step, the client atomically GETS the wrap, ciphertext, and root key ID. All other steps are local operations exactly the same as the abstract implementation of SecurePut (Figure 14). This means that step 1 is when the operation "instantaneously" takes effect, since other concurrent operations no longer affect the result of SecureGet. Since SecurePut and ciphertext updates only atomically PUT the wrap, ciphertext, and root key ID, then an atomic GET will only obtain values where all three are updated.

For SecurePut (Figure 6), the only step that causes race conditions is step 7a. All other steps are local operations exactly represented by the abstract implementation, except for the additional ciphertext version numbers. (The ciphertext version number has no impact on the result of SecurePut.) At step 7a, all values (wrap, ciphertext, root key ID, and ciphertext version number) are atomically PUT into the database, so this step is when the operation "instantaneously" takes effect. If there are multiple concurrent SecurePuts, then the wrap, ciphertext, and root key ID will never be in an inconsistent state because either all or none of the wrap, ciphertext, and root key ID are atomically PUT into the database.

For ciphertext updates (Figure 7), there are two operations which may cause race conditions: obtaining old ciphertext data and placing new ciphertext data, which are steps 1 and 15, respectively. Except for ciphertext version numbers, all other steps are the same local operations as the abstract implementation (Figure 16). Step 1 of QUICKeR's implementation atomically GETS the wrap, ciphertext, root key ID, and version number. Similar to SecureGet, the retrieved data is correctly decryptable because other concurrent operations atomically PUT all of the values at once.

Finally, step 15 in QUICKeR's implementation only aborts if the version number stays the same from step 0 to 15. We prove that this results in the same behavior as a linearized abstract implementation, which aborts if the ciphertext is already encrypted by root key k_{i+1}. If the ciphertext version number has changed, then a concurrent SecurePut for the same index i has already atomically PUT new values for the wrap, ciphertext, root key ID, and version number before step 15 in QUICKeR's ciphertext update implementation. Since SecurePut only PUTS new values if the ciphertext's root key ID is the new root key ID, then the new ciphertext must already be encrypted by the next root key k_{i+1}. Therefore, QUICKeR's implementation results in the same behavior as the abstract implementation.