Verifiable Lock Manager using Intel SGX

Bachelor thesis by Henry Helm
Date of submission: December 29, 2021

1. Review: Prof. Dr. Carsten Binnig
2. Review: M.Sc. Muhammad El Hindi
Darmstadt

Computer Science
Department

Data Management Lab

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Henry Helm, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 29. Dezember 2021

Signature

Abstract

We implement a centralized lock manager component to enforce a locking-based concurrency control scheme that works reliably even when malicious clients can arbitrarily deviate from their specifications. Transactions originating from different peers might attempt to maliciously acquire and release locks in order to compromise the concurrency control scheme and violate the integrity of the underlying data. We define possible attacks and derive specific countermeasures against them. One essential part of our lock manager's design is its implementation within a Trusted Execution Environment (TEE), namely Intel SGX, for verifiability and security. Furthermore, we examine the performance overhead and develop an architecture for the lock manager that addresses the memory limitations associated when building applications with Intel SGX.

Contents

List of Figures

List of Abbreviations

ACID Atomicity, Consistency, Isolation and Durability

BOCC Backward-oriented optimistic concurrency control

CA Certificate Authority

DBMS Database Management System

DKE Diffie-Hellman Key Exchange

DoS Denial-of-Service

ECDSA Elliptic Curve Digital Signature Algorithm

EPC Enclave Page Cache

EPCM Enclave Page Cache Metadata

FOCC Forward-oriented optimistic concurrency control

Intel SGX Intel Software Guard Extensions

MAC Message Authentication Code

MVTSO Multiversion Timestamp Ordering

OCC Optimistic Concurrency Control

PKI Public Key Infrastructure

PoET Proof of Elapsed Time

PRM Processor Reserved Memory

TCB Trusted Computing Base

TEE Trusted Execution Environment

2PL Two-Phase Locking

1 Introduction

1.1 Context and Motivation

We imagine a system for decentralized data sharing, where mutually distrusting parties can run transactions on shared data, without a trusted central authority, across borders of companies and institutions. Blockchain is the ideal candidate for serving as the underlying storage layer in such a system, because a Blockchain maintains an immutable transaction log that can easily enable trustworthy audits of past transactions, if needed.

But Blockchain comes with two main challenges that need to be overcome for this idea to turn into reality: First, the scalability and performance issues, and second, the lack of common abstractions that a relational DBMS provides, like an SQL query interface and sophisticated transaction processing with well-defined consistency levels.

In our work, we focus on the later and propose a locking-based protocol as a concurrency control scheme to ensure ACID-compliant transaction processing. We therefore implement a lock manager, which is responsible for keeping track of the locks on the database objects that transactions originating from different peers want to access. It is evident that a simple peer-wise local locking does not fulfill the requirements for a distributed trusted database. A peer-wise local locking would not guarantee isolation between transactions coming from different peers. Therefore, the lock manager needs to be centralized as its own isolated component, with which all peers can communicate remotely. We also cannot implement the locking scheme inside of the Blockchain directly. One reason is because off-chain transaction processing is preferable due to better performance. Additionally, when we want to shard the Blockchain itself to improve its scalability, all shards need to communicate with a central lock manager as well.

1.2 Problem Statement

A byzantine adversary model is necessary within a decentralized system without a trusted authority. According to this model, peers might be malicious and arbitrarily deviate from the protocol to try to break isolation guarantees of locking. This can negatively affect the integrity of the underlying data. A malicious client might attempt to circumvent the lock manager all together and directly send their respective read and write requests to the Blockchain, so the lock manager is not able to enforce its concurrency control. Another attack that needs to be considered is an adversary that tries to deprive the lock manager of its capacities by holding too many locks or never releasing locks, so that other peers are not able to complete their transactions. A trusted lock manager needs to have an appropriate answer to all of those attacks.

A third issue that arises is that the lock manager itself could get potentially compromised. How can clients put trust into a lock manager running on a remote computer and verify that it is acting correctly on their behalf? This issue is referred to as the *Secure Remote Computation Problem* [44]. To tackle this, we deploy the lock manager inside a Trusted Execution Environment (TEE) [37], also called enclave, which is a secure area inside the main processor that guards the code and data of the application loaded into the TEE even against system level adversaries, guaranteeing confidentiality and integrity and additionally enabling remote clients to attest that the expected code is running inside the protected enclave.

However, TEEs like Intel Software Guard Extensions (Intel SGX) [20] that we make use of in our implementation, come with a certain performance overhead and memory limitations. Therefore, we have to ensure that our implementation is memory efficient and as performant as possible.

1.3 Contributions

To tackle the above mentioned challenges, we deploy the security sensitive data and operations of the lock manager inside a TEE, namely Intel SGX (see Section 2.3), which makes it verifiable by remote peers through the means of software attestation (see Section 2.3.2) and immutable, even when the underlying system gets compromised.

A major issue when using Intel SGX is that the protected memory is limited to 128 MB and costly paging to and from untrusted memory sets in when this limit is reached. We show that the resulting paging overhead is indeed a major performance bottleneck when many locks get acquired. Therefore we modify our initial lockmanager implementation to store the memory consuming lock table outside of Intel SGX with additional integrity protection in place. This approach proves to be 8x more efficient.

We describe in detail the possible attacks that a malicious peer can carry out to compromise the concurrency control scheme or to target the availability of the lockmanager and propose countermeasures.

1.4 Thesis Outline

The thesis is divided into 7 chapters. Chapter 1 gives an introduction by motivating the topic and defining challenges and our contributions. Chapter 2 lays the foundation for the following sections by providing important background information on concurrency control, Blockchains and Intel SGX. In chapter 3, the attack model, that we are defending against, is defined and we enumerate the countermeasures we employ against those attacks. Following, we describe our three designs for the lock manager regarding the derived requirements, encompassing the attack countermeasures and specific design challenges related to Intel SGX. Chapter 4 will subsequently follow up with a more technical description of the implementation of our concepts and in chapter 5 we will evaluate our implementation regarding its throughput and memory consumption. Chapter 6 contains related work and chapter 7 gives a summary and highlights directions for future work.

2 Background

Before presenting the design choices of our lock manager, we need to introduce the reader to some fundamental concepts. First, we summarize the most important points about concurrency control and locking, so that the reader can understand the purpose of a lock manager in traditional database systems and the anomalies that concurrency control protects against. Later on, in Chapter 3, we will refer to these anomalies as ”attacks”. We will also pick up on the points mentioned here in Chapter 6, which contains related work that implements an optimistic locking protocol.

Then we explain the foundations of Blockchain for readers that are unfamiliar with the topic. While not necessary to follow with the later chapters, it helps to understand the application context of the lock manager within the envisioned decentralized data sharing system, in which Blockchain can serve as a tamper-proof storage layer. We also introduce database systems on top of Blockchain, which can make use of our lockmanager implementation, most notably, TrustDBle [17].

Lastly, we need to provide further information about Intel SGX. Intel SGX serves as the basis for our implementation and its limitations have a huge influence on our lock manager’s design. Further resources for the interested reader are *Database System Concepts* by A. Silberschatz [40], *Introduction to Modern Cryptography* by J. Katz and Y. Lindell [22] and *Secure Processors* by V. Costan, I. Lebedev and S. Devadas [10].

2.1 Concurrency Control

Concurrency control schemes in databases ensure isolation between several concurrently executing transactions. Isolation is a part of the Atomicity, Consistency, Isolation and Durability (ACID) properties, and therefore a fundamental attribute of transactions. Without isolation, several anomalies can occur, which violate the integrity of the data, e.g. Lost Update or Dirty Read. Information contained in this section is partly taken from *Database System Concepts* by A. Silberschatz, Chapter 18 (p. 835-904) on *Concurrency Control* [40]. Another source is the lecture on Concurrency Control Theory by Andy Pavlo at CMU, 2019 [8].

2.1.1 Transactions and ACID Properties

A transaction is a sequence of database instructions, e.g. SQL queries, to perform a higher level task. In a relational database system, it is often required that transactions adhere to the following four properties, called ACID properties:

1. **Atomicity**: Either all instructions of the transaction are *committed* to the database or none, i.e., the transaction gets *aborted*.

2. **Consistency**: A single transaction leaves the database in a consistent state.
3. **Isolation**: Transactions operate without interference from other concurrent transactions.
4. **Durability**: All data changes from the transaction are persisted after the transaction commits successfully.

Atomicity can be ensured by either *logging*, when a Database Management System (DBMS) logs all actions so they can be undone in case the transaction aborts, or *shadow paging*, where transactions operate on their own copies of pages which are only made visible after the transaction has successfully committed. Keeping the database consistent is usually the responsibility of the application which has to define individual integrity constraints and needs to make sure that requests obey to those constraints. Durability is achieved by writing data to persistent storage, logging and recovery protocols [31].

In the following sections we want to focus on enforcing isolation which is achieved via concurrency control protocols. The challenge is to allow concurrent transactions for performance reasons, but still make it appear as if the transactions run sequentially. A schedule that follows this property is called *serializable*. There are two categories of concurrency protocols:

- **Pessimistic**: Avoid *conflicts* between concurrent transactions. Examples of such protocols are *Lock-Based Protocols*.
- **Optimistic**: They assume conflicts occur only rarely, so that it is more efficient to deal with them after they happened via some repair mechanism. Examples of such protocols include *Timestamp-Based Protocols* and *Multiversion Schemes*.

To make clear, what we mean by potential conflicts, we will describe next what kind of anomalies can occur when several transaction run concurrently. Certain isolation levels exist that when enforced can avoid those anomalies.

2.1.2 Anomalies

Two operations can conflict if they come from different transactions, are on the same object and one of them is a write operation. Hence, three basic kinds of conflicts can occur:

- Read-Write Conflict (”Unrepeatable Reads”)
- Write-Read Conflict (”Dirty Read”)
- Write-Write Conflict (”Lost Update”)

We add pictures for all three anomalies that each illustrate them with an example, taken from CMU's lecture on concurrency control from 2019 [8].

Unrepeatable Read As an example of a Read-Write conflict (Figure 2.1, we see that T_1 reads the same object A twice, but inbetween its value is changed by the concurrently running T_2, which could not have happened in a sequential execution of the two transactions.

Dirty Read In Figure 2.2, we see a Dirty Read: T_2 reads A, whose value stems from a currently running, but yet uncommitted transaction T_1. T_2 updates A based on this value. However, subsequently, T_1

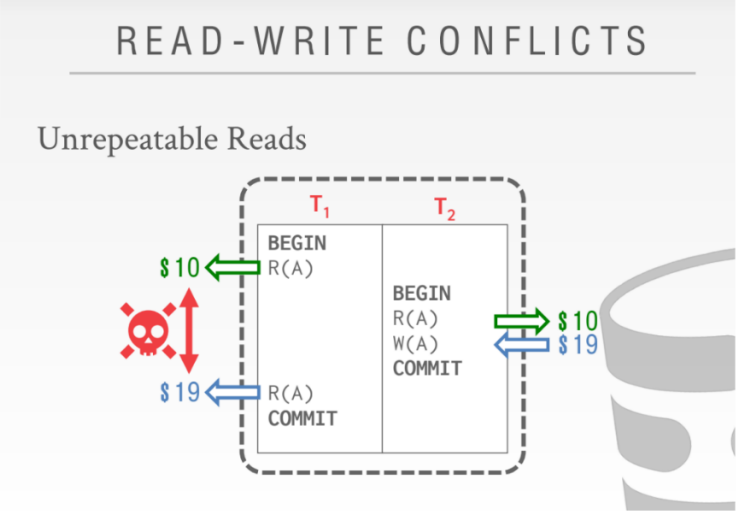

Figure 2.1: Unrepeatable Read (from CMU Lecture on Concurrency Control 2019 [8])

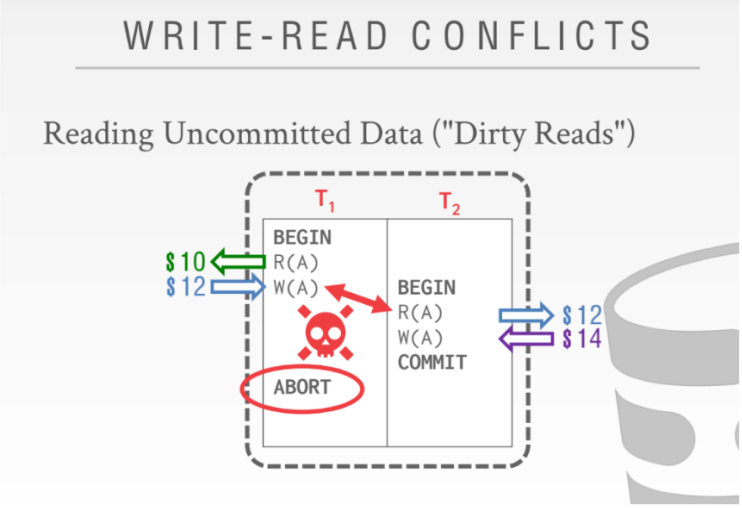

Figure 2.2: Dirty Read (from CMU Lecture on Concurrency Control 2019 [8])

aborts and its change to A is no longer valid. Therefore, T_2 has executed its operation based on invalid, uncommitted data.

Lost Update Similarly, a Write-Write conflict (Figure 2.3) can occur like in the last picture, when T_1 and T_2 each overwrite changes the other transaction has made. Here, T_2 overwrites the change of T_1 on A, then writes B, but that gets overwritten by T_1 afterwards as well, which can leave the database in an inconsistent state.

In subsequent sections, we also refer to those anomalies as ”attacks”, as a malicious client might want to break concurrency control to create such anomalies that affect the consistency of the data. ANSI SQL defines several isolation levels, i.e., rule sets that restrict reads and writes of transactions in such a way that they prevent certain anomalies. Stronger isolation levels prevent a larger subset of anomalies, but also lead to less concurrency. An example would be READ COMMITTED, which only allows reads on committed values and prevents the ”Dirty Read” anomaly (compare again the second

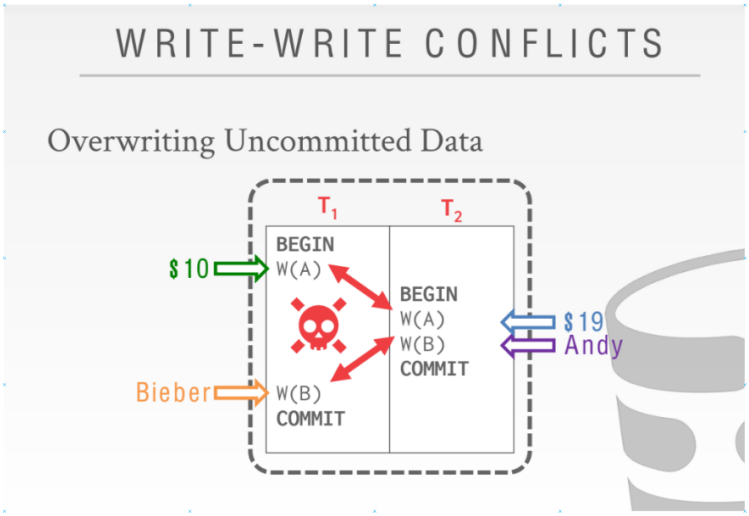

Figure 2.3: Lost Update (from CMU Lecture on Concurrency Control 2019 [8])

picture in Figure 2.2). A more concise definition of several isolation levels and the anomalies they protect against can be found in [5].

We want to ensure *serializability*, i.e., avoid all anomalies and execute transactions concurrently as if they would sequentially. Now, we want to focus on how we can actually enforce certain isolation levels. There exist three basic ideas, that we want to highlight shortly:

- **Locking**: a transaction locks a data item that it accesses, so other transactions cannot access it. Hence, conflicts are avoided. The process that receives lock and unlock requests and grants them according to a locking protocol, is called a *lock manager*. The lock manager maintains the necessary information about pending lock requests and acquired locks in a hash table, called a *lock table*.
- **Timestamps**: each transaction gets a timestamp when it begins and each data item keeps two timestamps, the timestamp of the transaction that most recently read it and the timestamp of the transaction of the most recent write. Using those timestamps, conflicts can be detected.
- **Multiple Versions**: Several versions of a data item are maintained, making it possible that a transaction can read an older value of a data item rather than the update made by an uncommitted transaction.

In the following sections, we will provide further details on locking-based protocols.

2.1.3 Two-Phase Locking

We can ensure isolation by requiring that transaction acquire a *lock* on a data item that they want to access. There are two variants of locks, either *shared*, where multiple transactions can access the same data item while having a shared lock on it, allowing for concurrent reads, or *exclusive*, where only one transaction with that exclusive lock can access that data item, for writes. When a transaction wants to acquire a lock, it has to wait until incompatible locks of other transactions on that data item are released. Under this circumstances, deadlocks can occur, where transactions cannot progress anymore.

For example, imagine a scenario, where T_1 holds an exclusive lock on A and wants to acquire a lock on B, while T_2 has a lock on B, but needs a lock on A. Both transactions cannot make progress in this situation. But on the other hand, if transactions release locks too early, again, inconsistencies can occur. Therefore it is required to follow a certain *locking protocol*, one of those is the *Two-Phase locking protocol* (2PL) that guarantees serializable schedules. It requires that each transactions locks and unlocks data items in two subsequent phases:

1. **Growing Phase**: A transaction can only acquire locks, but not release them.
2. **Shrinking Phase**: A transaction can release locks, but cannot acquire any more locks.

However, Two-Phase Locking (2PL) does not guarantee freedom of deadlocks. Therefore, we have to shortly talk about handling of deadlocks.

2.1.4 Deadlock Handling

There are two ways of dealing with deadlocks. Either we prevent the system to ever enter a deadlock state, called *deadlock prevention*, or we allow that, but need a way to *detect* deadlocks and subsequently *recover* from them.

Deadlock Prevention One possiblity to prevent deadlocks is to require all transactions to acquire all locks before the beginning of the execution. However, it is difficult to know the items that need to be locked beforehand in certain situations. Also, items tend to be locked a lot longer than they usually have to. Another approach are *lock timeouts*, where a transaction only waits a certain amount of time, and if the lock could not be acquired in that time frame, the transaction aborts. For a more detailed analysis of deadlock prevention techniques and their performance, refer to [2].

Deadlock Detection The other strategy is to detect deadlocks. The lock manager maintains a *wait-for graph*, which is a directed graph $G = (V, E)$ where the vertices V resemble the transactions and there exists an edge E out of from T_1 to T_2, if T_1 is waiting to acquire a data item currently locked by T_2. To detect deadlocks, it is only necessary to detect a cycle within that wait-for graph and subsequently determine a victim transaction that gets aborted to break the cycle.

2.1.5 Optimistic Concurrency Control

Pessimistic locking-based protocols are associated with a lot of overhead. If transactions can freely proceed with their read and write operations and only at commit they are checked for conflicts, then this is referred to as an Optimistic Concurrency Control (OCC) protocol. So each transaction executes on a private workspace and before it wants to commit, the DBMS checks for potential conflicts. If there are no conflicts, the data from the private workspace can be transferred into the database, else the transaction gets aborted. This is made possible by the maintenance of a read set $RS(T_k)$, the attributes read by transaction T_k, and the analogously defined write set $WS(T_k)$.

Backward-oriented optimistic concurrency control (BOCC) protocols check T_k on commit against all committed transactions T_i. If T_i committed before T_k started or when when $RS(T_k) \cap WS(T_i) = \emptyset$, there are no conflicts, because it is impossible that T_k has read any uncommitted changes from other transactions. Else T_k aborts.

Forward-oriented optimistic concurrency control (FOCC) protocols analogously check for all running transactions T_i, if $WS(T_k) \cap RS(T_i) = \emptyset$, making sure that no running transaction has read changes made by T_k.

This sums up our coverage on concurrency control. The reader now has the prerequisites to understand the necessity behind concurrency control and can derive the responsibilities for a lock manager within a DBMS. We have looked at the anomalies that concurrency control protocols protect against and which are referred to as attacks in the attack model that we describe in Chapter 3. Then we focused on locking-based protocols, specifically 2PL, which we use in our lockmanager. Lastly, we also mentioned OCC protocols and protocols based on timestamps or multiple versions, which come up again in related work (Section 6.2). Now, we will take a detour to the concepts of Blockchain and Blockchain Databases, as we implement our lock manager in the context of Blockchain Databases.

2.2 Blockchains and Blockchain Databases

In this section, we want to explain the basic concepts behind Blockchain, how it is envisioned to be used as a tamper-proof storage and what challenges still have to be overcome to turn this idea into reality. At the end, we give an example of database systems that leverage Blockchain as their underlying storage layer and resemble as the application context for our lock manager implementation. The information contained in this section is partly taken from *Database System Concepts* by A. Silberschatz, Chapter 26 (p. 1251-1279), on *Blockchain Databases* [40].

2.2.1 Block Structure

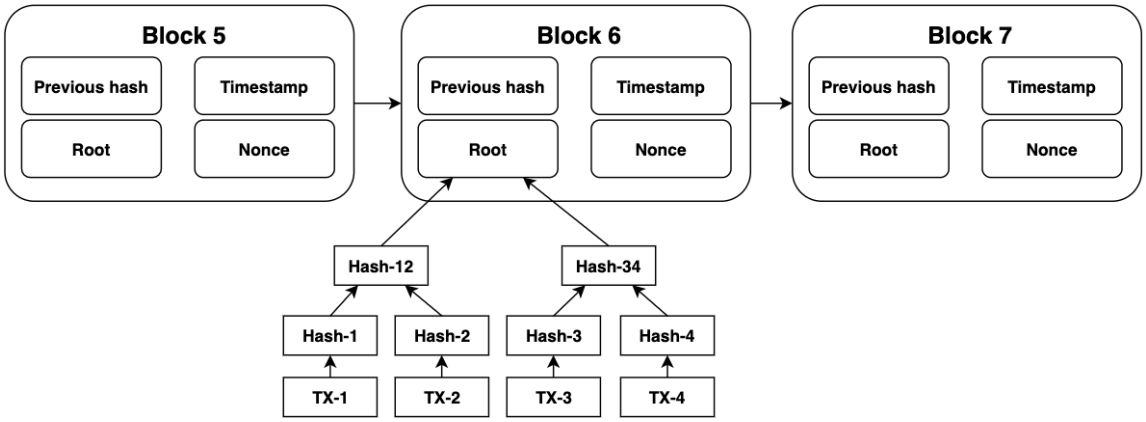

Figure 2.4: Blockchain Structure

The concept of a Blockchain was first introduced by Satoshi Nakamoto in his famous white paper about the cryptocurrency Bitcoin [32]. On a basic level, a Blockchain is a decentralized, distributed ledger that contains lists of records, called *blocks*. Each block contains a timestamp, a nonce and transaction data represented as a Merkle tree [29]. A Merkle Tree is a binary tree, where the leaf nodes are transactions and the parent nodes are formed by taking the hash of their child nodes (see Figure 2.4),

facilitating easy look-up and validation of transactions. The most important component inside a block is the hash of the previous block, which results in the blocks chained together: When the contents of one block changes, this invalidates all the hashes of the following blocks, which would need to be recomputed. When this is made infeasible, because of the huge amount of chained blocks and the computational effort associated in computing a specific hash, the Blockchain becomes practically *tamper-proof*.

With hashes we mean secure *cryptographic hash functions* $h(x)$ which have additional mathematical properties, from which we want to highlight the following three:

1. **collision resistant**: it is infeasible to find x and y, s.t. $h(x) = h(y)$
2. **irreversible**: given only $h(x)$, it is infeasible to find x
3. **puzzle-friendly**: given a value k, for any n-bit value y it is infeasible to find x so that $h(x||k) = y$ in time significantly less than 2^n (not a property of standard cryptographic hash functions but added here nonetheless for completeness, as it is needed in Section 2.2.2)

Collision resistance is important, because otherwise an attacker would be able to change a block in such a way that the result would still hash to the same value and he would not need to recompute the hashes of the following blocks. The relevance of irreversibility and puzzle-friendliness will become clear after we introduced *Proof-of-Work* algorithms in Section 2.2.2.

Blockchains can be classified into two categories:

- **public blockchains**, where anyone can join and participate in the network, and
- **permissioned blockchains**, where participants are fixed and determined upfront

Blockchains are managed in a decentralized manner, within a peer-to-peer-network, where every node holds an entire copy of the Blockchain. As a result, Blockchains offer three central properties: Data stored on the Blockchain is immutable through the immutability of the chain state. Data is stored in a fault-tolerant, persistent and auditable manner, because of the replication to all clients in the network. And since every client has access to the whole copy of the Blockchain, every transaction can be transparently verified by all parties.

2.2.2 Consensus Protocols

Because a whole copy of the Blockchain is stored on each participating system, this raises the problem that all nodes need a way to agree on the contents of the transaction log. In traditional distributed systems, typically consensus protocols would be used to come to an agreement between the nodes. Consensus protocols have the property that given a set of parties and their initial input, the protocol terminates after a finite amount of rounds and all parties agree on the same value, which was the input for a majority of the parties.

Examples of such algorithms are Paxos [25] and Raft [34], see *Database System Concepts*, Chapter 23.8 (p. 1150ff) on *Consensus in Distributed Systems*. The problem with those algorithms is that they assume a fixed number of participants and are not working under the *Byzantine adversary model* [26], where each node can arbitrarily deviate from the protocol. Especially in public blockchains, so called *Sybil Attacks* [12] are possible, where an attacker creates a large number of participants to join the network and form a majority of dishonest parties, which can control the outcome of the consensus

protocol and compromise the tamper-resistancy of the Blockchain system. Therefore, different kinds of consensus protocols are needed, which can be divided broadly into the following three categories:

- **Proof-of-Work**: The power to make the decision on what is agreed is not anymore bound to the majority of nodes, but to the majority of the compute power by requiring a *miner* of a block to solve a computationally intensive mathematical puzzle. Now an attacker needs to acquire 51% of the total compute power of the network, which is much more difficult. The proof-of-work algorithm used in Bitcoin is named Hashcash [4] and the principle behind it is that the hash of the block has to have a specific number of trailing zeros to be considered valid. More specifically, the nonce, a number that is part of the block, has to be found so that the hash of the block starts with n zeros, where n can be increased with growing compute power over time to make the puzzle more difficult.
- **Proof-of-Stake and others**: Roughly speaking, nodes holding a larger stake of the currency have a higher probability of adding blocks. There exists also Proof-of-Activity, Proof-of-Capacity and more. These are favored over traditional Proof-of-Work recently, because they do not waste as much energy and are therefore more environmentally friendly.
- **Byzantine consensus**: Those algorithms can only be used in permissioned blockchains, as they do not counter Sybil attacks, but can cope with a certain number of malicious nodes that try to disrupt consensus or reach a wrong agreement. Early work in this area comes from Pease et al. (1989) [35] and Lamport et al. (1982) [26]. Castro and Liskov (1999) [6] formed the foundation for many current Byzantine consensus algorithms used in Blockchains today.

2.2.3 Performance and Usability Challenges

To summarize, altering the contents of a block becomes harder and harder the longer the chain grows, specifically due to the necessary execution of a proof-of-work algorithm for every block, which is a function that can be efficiently verified, but takes a long time to compute. This makes Blockchain an immutable and auditable data store, but also increases the latency of adding new transactional data. All those features make it a distributed system with high Byzantine fault tolerance, but also lead to a lack of performance and scalability. Transaction processing rates are way beyond those of traditional database systems.

Consensus is regarded as one of the biggest bottlenecks in Blockchain performance. Two approaches for improvement are *Sharding* [46], where mining is distributed for more parallelism, and *Off-chain transaction processing*, where a trusted system executes some part of the transaction outside of the Blockchain without the overhead of consensus and only later updates the Blockchain.

For further adoption of Blockchain outside the world of cryptocurrency, for general data sharing use cases, it also misses a well-standardized query interface and other abstractions that are provided by regular DBMS. [11] provides further information on the Blockchain technology while also elaborating on what it takes to bring Blockchain to the data processing domain. BlockchainDB [16] is a database systems that leverages Blockchain as its underlying storage layer. Our work can be considered a follow-up work of BlockchainDB, presented next.

2.2.4 BlockchainDB

BlockchainDB [16] is a database that leverages Blockchain as its underlying storage layer to profit from its security guarantees and to enable auditable data sharing. It implements a database layer on top to overcome the major obstacles of Blockchain technologies. It provides a way to create shared tables with a simple key-value query interface including different consistency levels and improves its limited performance and scalability by utilizing sharding. That means, data is not replicated to all peers to not have high overhead of Blockchain consensus, but instead data is partitioned and only replicated to a limited number of peers. Because now, not every peer has a complete view of the data, reads or writes to remote peers become necessary and therefore an additional off-chain verification procedure, so that a client can ensure that the correct value for a GET request was returned or the PUT request was actually processed by the storage layer and not dropped by the remote peer. Evaluation has shown that BlockchainDB is able to provide two-orders of magnitude higher throughput than native Blockchains.

2.2.5 TrustDBle

TrustDBle [17] is the successor of BlockchainDB. It wants to enable secure data sharing between multiple parties by utilizing Blockchain for auditable storage. It implements components on top for verifiable processing and data sovereignty. Data sovereignty means that data providers can revoke access to data at any time. In that context, the paper introduces the concept of *data contracts*, which allow to dictate by whom the data can be used and which kind of queries can be run on the data. Verifiable processing means that the DBMS is able to prove that a transaction was executed in an ACID-compliant way. BlockchainDB only provided simple PUT and GET requests. Extending this interface to allow for complex SQL transactions and building a secure and verifiable transaction engine is the main concern of TrustDBLe at the time of this writing. It would have been possible to implement the transaction processing logic inside the Blockchain via smart contracts, but this would not solve the inherent scalability and performance problems. Still, we need a way for clients to be able to verify that each component of the transaction processing related to fulfilling the ACID properties works correctly. The solution is to put those critical components into a TEE, which is tamper-proof by the means of hardware isolation and can verify its integrity to a remote peer. One such a component is the lock manager, which is responsible for verifiable isolation.

2.3 Intel SGX

Next, we introduce TEEs, specifically Intel SGX, because as we have seen, they are an essential component in ensuring the security of our system. As we make use of Intel SGX in implementing our lock manager, it is important to know its security guarantees and shortcomings to be able to understand the reasons behind our design decisions. This section will cover the basics, we will provide further details in later sections as needed.

Intel Software Guard Extensions (Intel SGX) is an Intel architecture extensions that comes with a set of new CPU instructions and memory access changes to add support for TEE within Intel processors. The underlying concepts were introduced in papers by Intel in 2013 [28] [3] and from the 6th generation of processors onwards became widely available in 2015. It was designed to solve the problem of

secure remote computation, i.e., executing software on an untrusted remote computer with some confidentiality and integrity guarantees.

Intel SGX leverages trusted hardware on the remote computer to enable secure remote computation. One can say that there exist three core concepts behind Intel SGX that form the pillars of Trusted Computing in general, which will be outlined in the following sections: *Enclaves* (Section 2.3.1), *Attestation* (Section 2.3.2) and *Sealing* (Section 2.3.3).

2.3.1 Isolation via Secure Enclaves

Applications written for Intel SGX are generally split into two parts, an untrusted part and a trusted part called *enclave*, which is a protected environment for all security-critical operations and data. The confidentiality and integrity of the data and computation performed inside an enclave is shielded from attacks coming from malicious software on the same computer, even from the system software. Execution flow can only enter the enclave via special CPU instructions. SGX does this by setting aside a portion of memory inside the Processor Reserved Memory (PRM) and protecting it from all non-enclave memory accesses, including the OS and hypervisor. The associated memory is encrypted and only gets decrypted when inside the CPU.

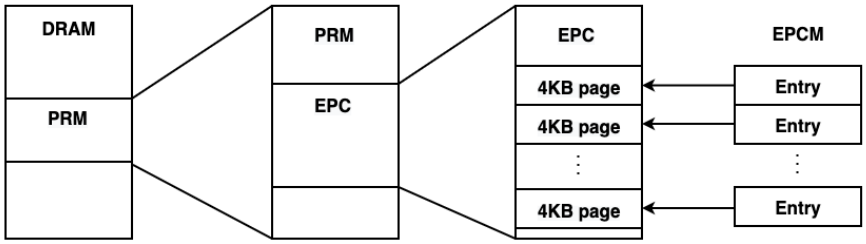

Figure 2.5: Intel SGX Memory Layout [10]

Each enclave gets assigned Enclave Page Cache (EPC) pages and the CPU tracks additional security-related information about each EPC page within the Enclave Page Cache Metadata (EPCM), all inside the PRM (see Figure 2.5). This needs to be done, because the system software is responsible for the allocation of EPC pages. However, the system software itself is untrusted, therefore SGX processors needs a way to check, if the system software correctly handles the page allocation.

For example, it needs to store the identifier for a specific enclave inside the EPCM entry for the associated EPC page, to ensure that an EPC page cannot be mapped to more than one enclave, as this way of sharing memory between enclaves would enable leakage of information to a potentially malicious enclave.

Also, access rights (*Read*, *Write* or *Execute*) of an EPC page are stored in its corresponding EPCM entry, so that the processor can detect, when the system software has tampered with the access rights.

Another piece of information that is stored inside the EPCM for security reasons is the virtual address of the EPC page. This is to counter Address Translation Attacks.

Imagine a scenario, where we have code that performs access control (see Figure 2.6). When the access control passes, e.g. when the right token was provided, we execute some portion of code that

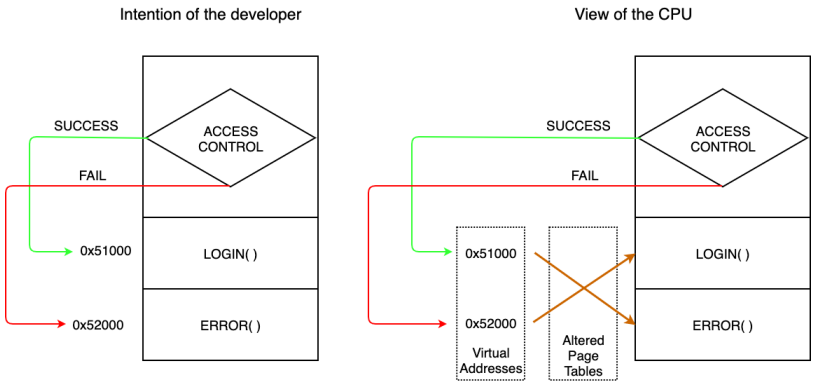

Figure 2.6: A simple Address Translation Attack [10]

resembles a `LOGIN()` function and is located within the page with the virtual address of `0x51000`. Else, an error is displayed using the `ERROR()` code at virtual address `0x52000`, where no access is granted. But as one can see in the right half of Figure 2.6, the system software can modify the page table at will and therefore is able to change the mapping between virtual and physical address, switching the pointers to the `LOGIN()` and `ERROR()` page. That means, when the access control fails, the attacker still gets logged in.

But with Intel SGX, the intended virtual address of an EPC page is stored in its EPCM entry, so the CPU can check, if the virtual address from which a request was mapped to the physical address of the EPC page, is actually the correct virtual address or if the mapping inside the page table was manipulated. A wider range of attacks are discussed in a later section about side-channel attacks on Intel SGX (see Section 2.3.5).

An important question arises that when non-enclave software cannot directly access the PRM, which is very important for the isolation guarantees of enclaves, how can system software load the initial code and data into a newly created enclave? The initial code is indeed loaded in by untrusted system software. During the initialization, the system software asks the CPU to copy the data from untrusted memory into EPC pages and assigns those pages to the specific enclave. When this process is finished, the enclave is marked as initialized. Now, software can execute inside that enclave, but the system software can no longer add new EPC pages, so the enclave becomes tamper-proof and truly isolated.

As the finalization of the initial loading step, a *measurement* of the enclave is computed, which is the hash over the enclave's code and its initial data, taking into account the order of the placement of the pages and their security properties. This functions as the identity of the enclave, since any change in the code or data results in an entirely different measurement. This fact will enable a remote party to perform software attestation (see Section 2.3.2).

2.3.2 Establishing Trust via Remote Attestation

Using remote attestation, a user that is communicating with an enclave can ensure that it was setup properly within a secure container hosted by trusted hardware and that the code executing inside that enclave is unmodified and as intended by the user. Any unauthorized changes to the software on the remote computer can be immediately recognized. This works by letting the trusted hardware generate

a certificate that proves that the software was unaltered via signing its *measurement* introduced in the previous section. After the user has performed remote attestation and therefore has established trust in the correct workings of the enclave, she can proceed to provision his secrets to the enclave.

The proposed remote attestation schemes slightly differ between various architectures, e.g. for Intel SGX [21], RISC-V [38] or ARM [1]. Here, we will only introduce a simple generic version that combines remote attestation with a key agreement protocol to highlight the general principle.

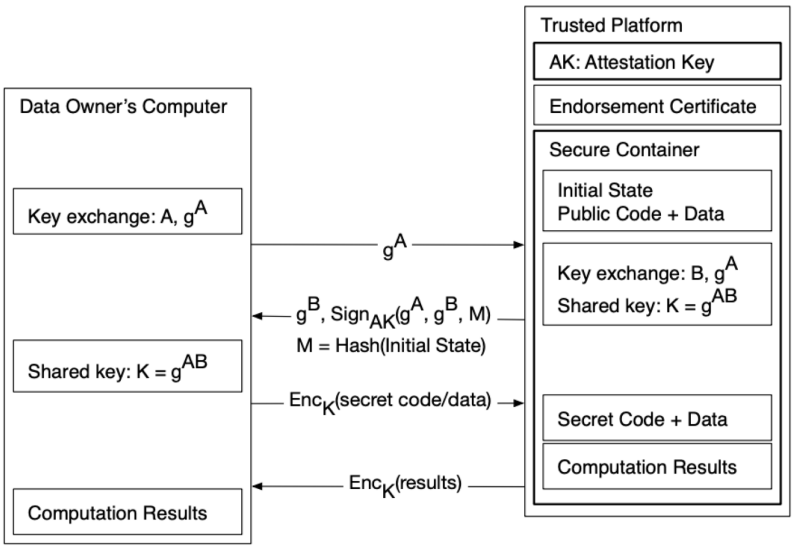

Figure 2.7: Remote Attestation combined with Diffie-Hellman Key Exchange (DKE) [10]

In Figure 2.7, we see a Diffie-Hellman Key Exchange (DKE) to compute a shared key and establish a secure channel between the user and the trusted platform for secret provisioning. The only modification to the standard DKE is that the response of the trusted platform also contains the signed measurement hash M. More specifically, when the remote party receives g^A from the user, it generates g^B in software and asks the trusted hardware to sign (g^A, g^B, M). After the user has verified the signature, it serves as authentication of the trusted platform, which prevents Man-in-the-middle attacks that stem from lack of authentication in key exchange protocols. If the verified measurement of the trusted platform does not match the user's expectations, she denies further communication and does not provision her secrets to the enclave.

The chain of trust is rooted at a signing key by the hardware manufacturer, acting as a Certificate Authority (CA). The hardware manufacturer has stored a unique *attestation key* inside the hardware in tamper-proof storage during manufacturing, which is used to produce the signatures. The manufacturer also has generated an *endorsement certificate*, which can be used to validate the associated attestation key.

Note that this design requires that the user has to trust the hardware manufacturer.

2.3.3 Securing Data via Sealing

Sealing refers to securely storing data outside of the enclave, "sealed" from access by anyone but the enclave itself. Sealed storage is an important concept, because we need a way to persist data securely

outside of the enclave. On the one hand, this is due to the limited memory of 128 MB set aside for EPC pages inside the PRM. In consequence, we can run out of space, and Intel SGX needs a secure way to evict pages to DRAM and load them back later on. On the other hand, all secrets provisioned to an enclave get lost when the enclave is destroyed. Here, we need sealed storage as well in order to maintain this secret data across sessions.

For storing data in unprotected storage, three things are taken care of by Intel SGX's sealing mechanism: Confidentiality via encryption, integrity via Message Authentication Code (MAC) and freshness via nonces. The derivation of the *sealing key* used for those cryptographic primitives is possible with two different policies: We can bind the key to the current enclave's identity or to the enclave author. While the former only allows the restoration of the sealed secrets by the exact same enclave, the later enables sharing information via sealed storage between different enclaves of the same author (see the Intel SGX SDK Developer Reference [20]).

2.3.4 Applications

Intel SGX is used in cases where sensitive data should be processed privately on a remote computer and/or when a remote computation should be executed securely without the underlying system to be able to intervene with its integrity.

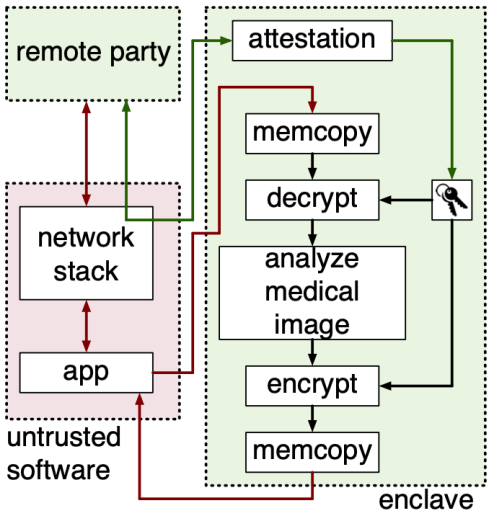

Figure 2.8: Example for an SGX application

A concrete example is shown in Figure 2.8, taken from [10]: A cloud service performs some form of image processing with confidential medical images. To protect data privacy, the user encrypts the input images and provisions the decryption keys securely inside the enclave, after having performed remote attestation. The enclave contains the code for encrypting and decrypting the images and the image processing algorithm. Code for receiving sent images and storing them should be implemented outside the enclave, as it is not relevant for security and privacy. Subsequently, the enclave can send back encrypted results to the user.

Real world use cases include privacy-preserving machine learning [18] and data analytics [48]. More widely known software that makes use of Intel SGX are Signal for private contact discovery [27] or

in the Hyperledger Sawtooth Blockchain framework for its scalable and efficient Proof of Elapsed Time (PoET) consensus algorithm [1].

There are also databases that are using Intel SGX. EnclaveDB [36] is a secure in-memory database, which guarantees confidentiality and integrity by putting security-critical components into the TEE. The untrusted part hosts public data. The sensitive data is stored inside the enclave, together with a query processing engine and natively compiled stored procedures. Clients connecting to EnclaveDB create a secure channel with the enclave and generate a shared session key. They also need to authenticate the enclave via remote attestation. Parameters sent by the client are encrypted using authenticated encryption, where the authenticated data portion contains data to prevent replay attacks, like a nonce. Results are also returned encrypted by the enclave. EnclaveDB's biggest contribution is an efficient protocol that ensures the integrity and freshness of the database log, which is necessary for database discovery, and due to its huge size, needs to be stored outside the enclave.

2.3.5 Side-Channel Attacks

Unfortunately, Intel SGX is vulnerable to a range of well-known side-channel attacks, that can deplete the confidentiality guarantees of the data to varying extend. It is in the hand of the developer to build counter measurements against those type of attacks into the code. In the following section, some basic classes of side-channel attacks that also apply to Intel SGX are outlined. For a more thorough discussion of the topic refer to [33].

Controlled-Channel Attacks make use of the fact that the untrusted OS has full control over the platform, e.g., memory management and can modify page tables at will. One way of exploiting this is to induce artificial page faults by setting the "present" bit in page table entries to 0 in order to leak information about which page was accessed by the enclave [45]. Subsequent research showed that even bits of cryptographic keys can be extracted in this way [39].

Cache Attacks exploit the cache-hierarchy system, where portions of the code that were accessed more recently are higher in the cache-hierarchy for lower latency. The time difference between a memory access causing a *cache miss* compared to a *cache hit* can be easily tracked, allowing an attacker to infer, which cache lines got accessed. One common methodology for this is the PRIME+PROBE approach. In the *priming phase*, the attacker fills up the cache with arbitrary data. In the *probing phase*, the attacker then runs the algorithm and afterwards measures the access times for the data written in the priming phase, from which he can infer if a cache hit or miss occurred. This knowledge is enough to even extract AES keys [30].

Speculative Execution Attacks are based on an optimization technique of modern processors, where processors speculatively execute instructions. When a branch occurs, they make a prediction about which branch is likely to be chosen and then follow that branch. Of course, the CPU will rollback the instructions once it becomes clear that the wrong branch was taken, but this still leaves intermediate traces in the caches, which can be exploited. This is the underlying concept of the Spectre security vulnerability discovered in 2017 [24]. Researchers have shown that speculative execution attacks are also applicable to Intel SGX enclaves [7].

[1] `https://medium.com/coinmonks/a-processor-at-the-heart-of-hyperledger-sawtooth%2Deng-763900f204f2`

Of course, there are some attempts in the research community to thwart the effect of side-channel attacks, but hardening against them is out-of-scope for this work. The potential leakage of information is not so critical to our application, since the data we process, except of some cryptographic keys, is not confidential. We only want to ensure the integrity of data and computation through the enclave.

Further information about Intel SGX can be found in [10].

3 Design

We envision a system for decentralized data sharing using Blockchain, with which parties can share data across borders of companies and institutions, even if they do not fully trust each other. Blockchain is a great candidate to serve as the underlying storage layer in such a system because of the inherent auditability it provides. However, Blockchains lack certain abstractions that relational DBMS provide, so we want to extend Blockchain with a transaction layer that guarantees ACID-compliant transaction processing. One major component in a transaction layer is concurrency control that ensures concurrent access of transactions while maintaining the integrity of the data. In this work, we focus on a lock-based protocol for concurrency control and the implementation of a corresponding lockmanager.

Concretely, we implement the 2PL protocol that avoids all inconsistencies and ensures serializable transactions. However, under our trust model, all clients of the system can be malicious and arbitrarily deviate from the protocol. The main design challenge is to account for these malicious actors and define defense mechanisms against possible ways to compromise the concurrency control scheme. We define possible attacks in Section 3.1 and defense mechanisms in Section 3.3.

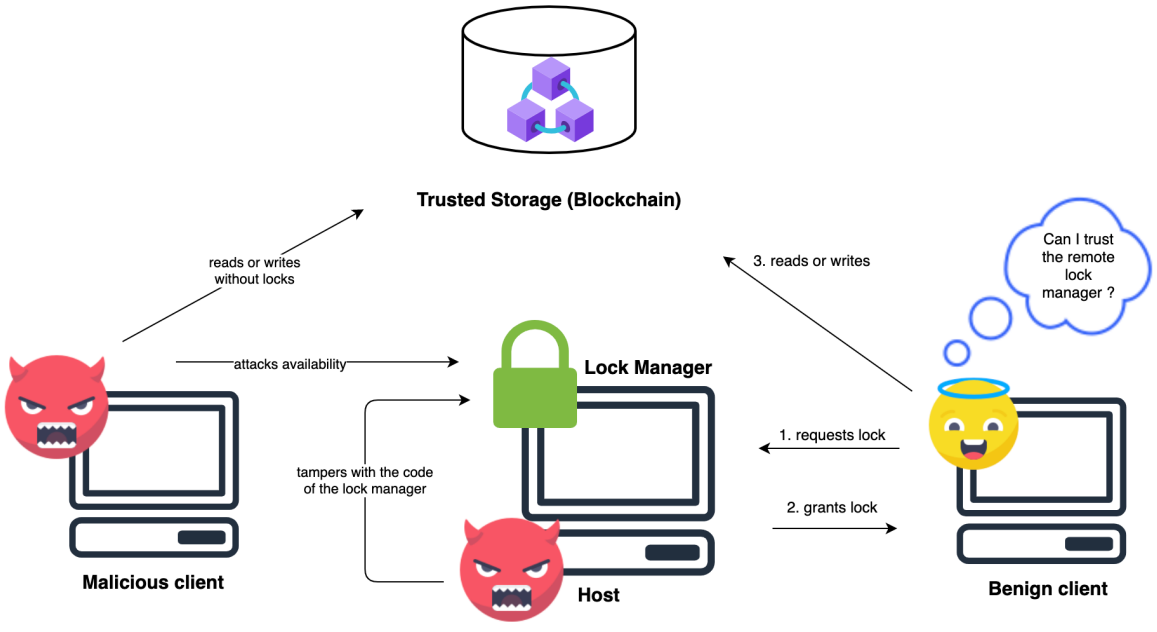

Figure 3.1: Lockmanager Instance with Clients and Storage Layer

In our system, the lockmanager has to be a centralized component residing on a separate node. Peer-wise local locking cannot ensure isolation across clients. Additionally, we do not intend to

implement the lockmanager inside the Blockchain directly. This would not be applicable anymore once we decide to shard the Blockchain for scalability reasons.Then we would have several distinct Blockchain nodes within the system which require a centralized lockmanager component. Additionally, offchain transaction processing is desirable due to performance reasons. The fact that the lockmanager resides on its own node might lead to clients questioning if they can trust the lockmanager's integrity. Namely, we also have to account for another type of malicious actor, the underlying host of the system the lockmanager resides on.

To make sure that the lockmanager cannot be compromised by an underlying malicious system, we deploy it inside a TEE, namely Intel SGX, that guards the lockmanager from manipulation and enables clients to verify its integrity. We discuss attacks by the underlying host of the lockmanager in Section 3.2 and design challenges implementing the lockmanager inside SGX in Section 3.4.

In Figure 3.1, you see a summary of the components of our decentralized data sharing system: A trusted storage in the form of Blockchain that gives us certain security guarantees like auditability and tamperproofness (compare Section 2.2), clients that want to access the storage layer and a centralized lockmanager component. You can also see the two possible types of attackers, malicious clients and a malicious host of the lockmanager, additionally, a benign client, that might seek some form of confirmation that the lockmanager is running correctly and has not been tampered with, if he is not fully trusting its underlying host.

3.1 Client Attack Model

First, we describe the attack model for clients of the lockmanager with respect to the attacks that we assume those clients may attempt to carry out. In general, clients can target either the consistency of the stored data by manipulating the concurrency control scheme that the lockmanager tries to enforce, we refer to those as *integrity attacks*, or the availability of the lockmanager, we refer to those as *availability attacks*.

3.1.1 Integrity Attacks

Bypass-Attack The most straight-forward way for a client to bypass concurrency control, is to issue reads and writes directly to the storage layer without acquiring a lock on the respective data item from the lockmanager beforehand. This is in theory possible, because the storage layer and the lockmanager are two distinct components, residing on different nodes. The lockmanager needs to be able to generate a proof that a certain lock was acquired by a client that can be verified by the storage layer and also not forged by a malicious client. We refer to this proof as a *lock signature*.

Lock Signature Reuse Attack A malicious client is not able to forge a lock signature, but he might be able to reuse an old lock signature later on. For example, he acquires an exclusive lock for writes on a data item and gets a valid signature, but afterwards immediately releases the lock again. From the perspective of the concurrency protocol, the client should not be able to write to that data item after that, but the client still is in possession of the signature. Without a way to invalidate the lock signature, the client can reuse it at a arbitrary point in the future to issue writes whenever he pleases, violating transaction isolation. We refer to approaches to invalidate signatures after certain points as *signature timeout*.

2PL violations Violating 2PL means that resulting schedules are not serializable and anomalies can be provoked. However, as long as the lockmanager can be trusted, it will enforce 2PL and only grant a lock signature if the corresponding request adheres to 2PL.

Node Impersonation Attack A malicious client might also try to impersonate other clients and make requests on their behalf. This is also targeting the integrity of the stored data. Therefore, we assume that a Public Key Infrastructure (PKI) is in place and the lockmanager has knowledge of the public keys of its clients. When all requests are signed with the respective client's secret key, the lockmanager is able to identify the client and impersonation is no longer possible.

Request Integrity and Replay Attack A malicious client might attempt to tamper with requests from other benign clients, e.g. changing the data item to acquire a lock for. However, this is not possible, as the malicious client is not able to forge a valid signature for the request of the benign client. Replays of older requests are also not possible, if each request has a field with the current *timestamp*.

3.1.2 Availability Attacks

A client might also target the availability of the lockmanager, which effectively hinders benign clients to further proceed, as they need to acquire locks from the lockmanager for every read and write request. We do not consider standard Denial-of-Service (DoS) attacks that attempt to overload the network with too many requests. However, we do consider attacks from malicious clients that try to drain the lockmanager from its limited (memory) resources:

Acquiring too many locks When data about the acquired locks has to be securely stored inside the enclave, we have to cope with the enclave's limited memory resources. A malicious client might attempt to acquire a huge number of locks in order to consume excessive amounts of memory. We therefore employ a *lock budget* for each transaction. A transaction is only allowed to acquire any number of locks smaller than its lock budget. This raises the question on how to determine and control this lock budget. More details are provided in Section 3.3 under *Lock and Transaction Budget*.

Creating too many transactions Analogously, we have to limit the number of concurrent transactions a client is allowed to create. We do not consider how to determine a proper lock- and transaction budget beforehand.

Holding locks indefinitely A malicious client could attempt to acquire locks and never release them, this time, to target availability, namely, to avoid other clients being able to acquire the locks for themselves. This issue is resolved with a form of *lock timeout* as well.

3.2 Lockmanager Attack Model

Benign clients might also distrust the node that is hosting the lockmanager itself. A malicious actor with administrator privileges would be able to modify the lockmanager application at his will, giving him full control over the concurrency control of the database system. The actor would then have the ability to violate the integrity of the data by making incorrect locking decisions or target the availability of the lockmanager. Clients want a way to ensure that they are communicating with an authentic lockmanager that is enforcing correct concurrency control. This is an instance of the secure remote computation problem. We cannot guarantee availability, however, since this is impossible for a

centralized lockmanager, whose underlying system is completely controlled by an attacker. What we do ensure is that clients will always communicate with an honest lockmanager and that clients can get proof that the code of the lockmanager has not been tampered with and is guarded even against system-level attacks, through Intel SGX.

3.3 Defense Mechanisms

Our goal is to implement Byzantine locking. Therefore we have to deal with the aforementioned attacks, both from the clients as well from the host of the lockmanager itself. Here, we summarize our attack countermeasures.

Attack	Attacker	Countermeasure
Bypass Attack	Client	Lock Signatures
Lock Signature Reuse	Client	Signature Timeout
2PL violations	Client	Checks inside the enclave
Node Impersonation	Client	PKI, network security
Request Integrity and Replay Attack	Client	Timestamp and Signatures
Acquiring too many locks	Client	Lock Budget
Creating too many transactions	Client	Transaction Budget
Holding locks indefinitely	Client	Lock Timeout
Tampering with lockmanager	Host	Enclave isolation and remote attestation

Table 3.1: Overview of Attacks and Countermeasures

3.3.1 Enclave Isolation

To protect the lockmanager from a compromised host system, we need to deploy the security sensitive data and operations of the lockmanager inside a TEE, namely Intel SGX, which protects all code inside the enclave even from system-level adversaries (Section 2.3.1). It is now impossible to tamper with the logic of the lockmanager. It can enforce the concurrency protocol, including 2PL, correctly.

3.3.2 Remote Attestation

Clients can use remote attestation (Section 2.3.2) to convince themselves about the authenticity of the lockmanager. Within this process, client and lockmanager can securely handle out session keys.

3.3.3 Node Authentication

These session keys can be used by a client to transfer its public key to the enclave. The public key should be signed by a trusted authority, so that the enclave can determine if the client is legitimate. Storing public keys of clients consumes the limited memory inside the enclave. However, for all signatures, we use signature algorithms that are known to have smaller key sizes, e.g. algorithms

based on Elliptic curves [14]. Additionally, public keys of inactive nodes can be safely evicted out of the enclave through Intel SGX's sealing mechanism (Section 2.3.3).

3.3.4 Lock and Transaction Budget

To thwart availability attacks from clients, the lockmanager keeps track of a per-transaction lock budget and a per-node transaction count budget that limit the amount of concurrent transactions and locks that can be hold. This comes down to two simple counters that the enclave has to maintain, but also requires two additional steps. First, nodes have to register at the lockmanager. The process for this is the following: The client will perform Remote Attestation with the enclave. Within the process, they exchange session keys and the client securely transfers his public key to the enclave, which is signed by a trusted authority. The enclave has the public key of a trusted authority hard-coded inside itself to verify that signature and avoid unwanted clients. Then, the enclave determines a transaction budget for the client and maintains a counter for the currently active transactions for that client, making sure it does not surpass the transaction budget. Additionally, each client needs to register individual transactions before making requests. The transaction will get assigned a lock budget, that is the upper limit of locks the transaction is allowed to acquire during its life time. We do not specify how to determine both budgets, but note that the lock budget could be derived using *cardinality estimation*, which is a technique used in query optimizers to estimate how many rows a certain query yields [15]. This estimate summed up over all queries inside the transaction might be used to determine an upper bound on the locks the transaction needs to acquire. Another approach would be to determine the lock budget statically, based on predefined transaction types.

3.3.5 Lock Signatures

Lock signatures are there to counter *bypass attacks* (Section 3.1.1): To avoid that clients directly access the storage layer without acquiring locks beforehand, the lockmanager signs each lock that he issues to clients. We again do this by making use of Elliptic curve cryptography due to smaller key sizes. The secret key used by the enclave can be persisted over several sessions via sealing (Section 2.3.3). The storage layer has to verify those signatures and neglect requests without valid signatures. In case of a Blockchain, this can be implemented using smart contracts.

At this point we want to note that, although we did not implement this, it is possible to amortize the considerable cost of signing locks. One possibility to do this is at the client side, by changing the API and enabling clients to request a set of locks with only one request, which corresponds also to only a single signature. Another possibility would be to do it at the server side, by generating a Merkle tree (which was already introduced in Section 2.2.1) for a batch of locks and only signing the root of the tree. Then, we do not have to sign the locks individually, but need to send more messages, namely the intermediate nodes of the tree that are needed to verify that the given signed root is correct and contains the hash of the lock that was acquired (see Figure 3.2 for a visualization of the Merkle Tree and the necessary hashes that the lockmanager needs to return, alongside the lock that was acquired and the signature of the root). This idea is introduced in *Basil* [42], which is part of related work in Section 6.2.

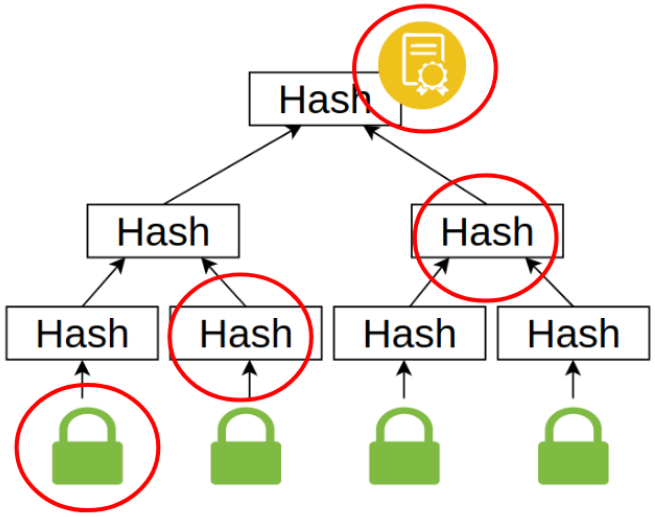

Figure 3.2: Amortizing the cost of Lock Signatures using Merkle Trees

3.3.6 Lock Timeout

Inside the lock table, the lockmanager stores a time stamp for every lock and transaction that currently owns it. A timeout is predetermined and each owner of a lock, whose associated timestamp is older than the timeout, can be deleted. This avoids clients holding locks too long and e.g. when having an exclusive lock, preventing all other clients from modifying that value. A problem hereby is, that the enclave cannot directly get a time stamp of its own, since this requires operating system calls, which are also untrusted in our threat model. Due to the same reason, an enclave can also not have a trusted counter. There exist no sense of time inside the enclave without special hardware. However, assuming that in a permissioned setting there is an honest majority of clients, the enclave can derive an accurate enough time stamp, when time stamps are sent from clients with their requests to the enclave, which is conveniently already done to avoid replays of requests. One can argue that this is not effective when the rate of incoming transactions is very low, but in this case timing out locks is arguably also not necessary.

3.3.7 Signature Timeout

We refer to signature timeout as the mechanism to avoid *Lock Signature Reuse Attacks* (see Section 3.1.1). We stress that reusing a lock signature for reads is not an attack, as a malicious client that can perform reads at arbitrary points by reusing an old signature does not affect the integrity of the data for other clients. Therefore, we can only focus on signature reuse for write requests. Also reuse of signatures for write requests are only a problem, when in-between the two usages other clients read or write to that value. Since writes are always exclusive, we know that this client has to be a malicious client, since he released the lock, therefore allowing other clients to issue reads or writes to the corresponding data item, and then reuses the signature of the already released lock, which is forbidden for benign clients. Notice however, that repeated use of the same lock signature for writes is permissible when the client has not released that lock yet, since that doesn't allow in-between reads or writes by other clients, keeping isolation.

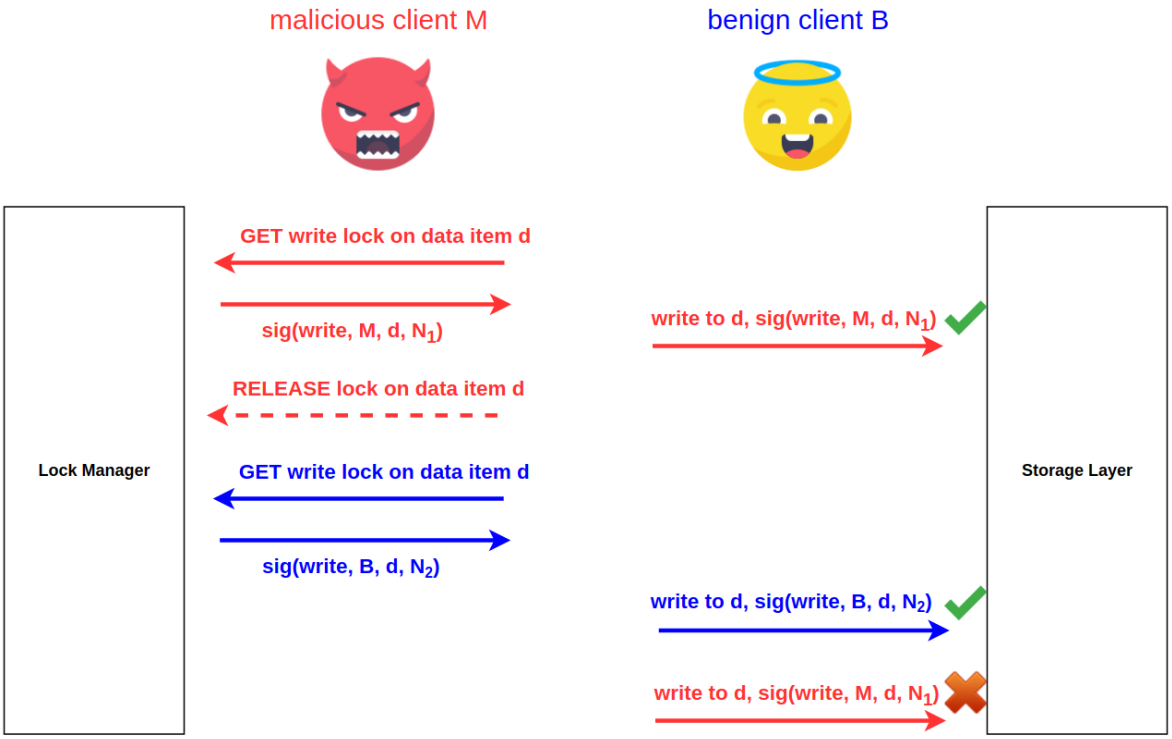

Figure 3.3: Signature Reuse

Now, to solve the problem, we add a *fencing token* N to each signature. The value for that token comes from a counter that the enclave maintains for each data object. That counter gets incremented whenever a client releases an exclusive lock for that data object. In other words, this ensures that the token will always be bigger than before, after an exclusive lock was released. The only thing the storage layer has to do now is to remember the largest fencing token it has seen for each request to a data item and frame signatures as invalid, whose token is not larger or equal to that value. This is avoiding the attack, as can be seen in Figure 3.3. After the malicious client M has released its write lock on the data item d, the internal counter that the lockmanager maintains for d, will be incremented. Therefore, $N_2 > N_1$ and the storage layer will refuse the write request of M, where she reuses an old signature, after having seen the request of benign client B with the new token N_2. We stress again, that this defense mechanism makes assumptions about the storage layer though, namely that the storage layer keeps track of the highest fencing token for each data item.

3.4 Intel SGX Design Considerations

When designing applications with Intel SGX, we have to divide our code in a secure and insecure part. The insecure part is outside of the enclave, while the secure part is protected by the enclave and contains sensitive data or security-relevant code logic that needs the extra protection by the TEE. Most importantly, due to security considerations, the amount of code we put into the TEE should be as small as possible, to make it easy to test and ensure that there exist no software vulnerabilities. Intel SGX

actually enforces a strict memory limit of 128 MB, so our solution should be able to comply with those tight memory restrictions. Also, the performance overhead that comes with utilizing hardware-based security solutions like Intel SGX should be taken into consideration.

3.4.1 Enclave Design

To decide which part of our application we need to put inside the enclave, we have to identify the resources that we need to protect, the data structures that maintain those resources as well as the code logic that deals with them.

In the case of our lockmanager, the resources that we need to protect, i.e., cannot store in untrusted memory as a system-level adversary could modify them, are the following:

- **Node Table**: As mentioned under *Node Authentication* in Section 3.3, we need to store public keys for each node together with a transaction budget. This information is stored inside the Node Table. Nodes are identified by a node ID.
- **Transaction Table**: The transaction table maintains the lock budget for each transaction, identified by a transaction ID (see *Lock and Transaction Budget* in Section 3.3) together with other information like which locks the transaction currently holds.
- **Lock Table**: The lock table stores for each data item, identified by a row ID, the corresponding lock data structure, that keeps track of the transactions that currently hold the lock, a.s.o.

We also need to define interface functions. Those are the functions of the enclave that the untrusted application can call. Due to security reasons, the untrusted application can only transition into the enclave code via these interface functions, called *ECALLS*. Since the input parameters to those *ECALLS* can be modified by the untrusted application, the enclave does extensive checks which add overhead for every transition from the untrusted to the trusted part, making those so called *enclave switches* much more expensive than system calls [43]. For completeness, there are also *OCALLS*, calls from the enclave to the untrusted part, which also have to be defined, e.g., so that the enclave can access operating system functionality, which are only available from the untrusted part. The return values of those calls however are not guaranteed to be correct in our security model. We also do not make use of *OCALLS* in our design.

In our case, conceptually, the necessary *ECALLS* are:

- **RegisterNode()**,performing Remote Attestation, assigning a node its ID and sending the public key to the enclave
- **RegisterTransaction(TS, sig)**, starting a transaction and getting assigned a transaction ID and lock budget for the duration of that transaction
- **Lock(NID, TXID, RID, mode, TS, sig)**, acquiring a lock as node NID for transaction $TXID$ on data item RID in either shared mode for reads or exclusive mode for writes
- **Unlock(NID, TXID, RID, TS, sig)**, releasing a lock as node NID for transaction TXID on data item RID

The requests have to contain a timestamp, for once to avoid replay attacks and second, because the enclave needs timestamps of incoming requests to get a notion of time for enforcing the *lock timeout*, as described in Section 3.3 under *Lock Timeout*. Note, that the enclave cannot get a trusted timestamp otherwise via *OCALLS*, because the timestamp would stem from the untrusted operating system, which could modify the returned timestamps to avoid locks to time out.

Each request also gets signed. The signature sig is created with the secret key of the corresponding node NID to avoid *Node Impersonation Attacks* (Section 3.1.1). More details follow in Chapter 4.

3.4.2 Memory Limitations

Another huge design consideration with Intel SGX is its limited memory size of 128 MB. To highlight its importance, we redo an experiment from *ShieldStore* [23] (Section 6.1), where the authors measure memory access latencies (see Figure 3.4).

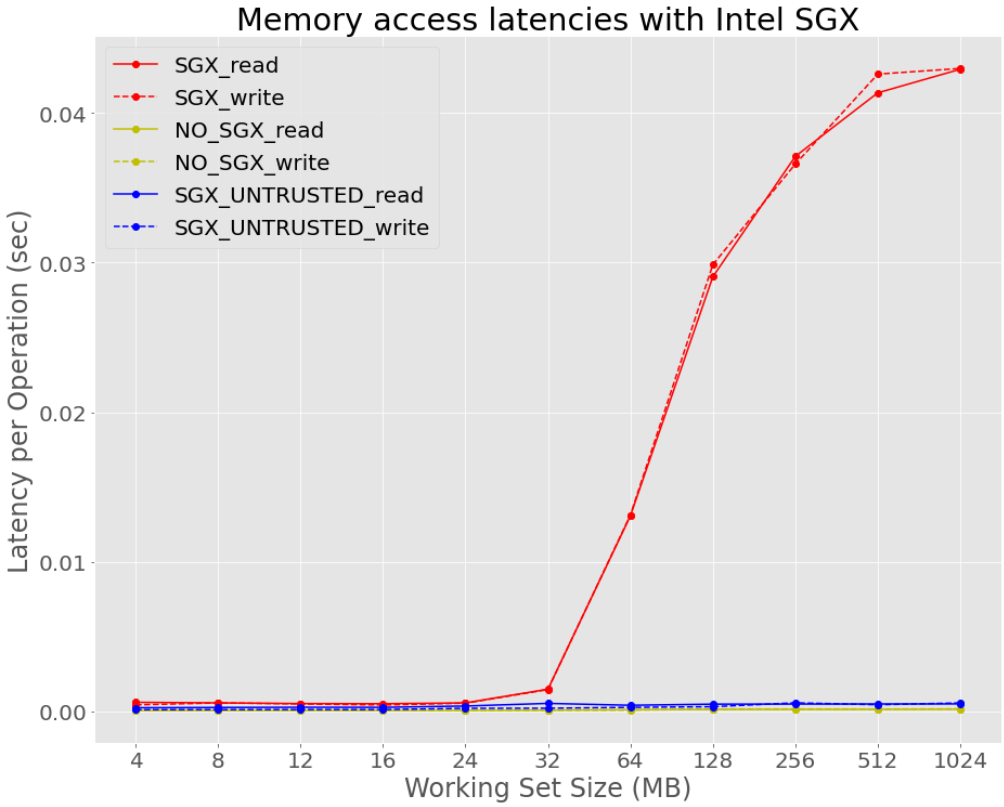

Figure 3.4: Memory Access Latencies

We measure memory access latencies in seconds for reads and writes with growing working set sizes in MB. Once for normal reads and writes without SGX (yellow), then inside SGX (red) and lastly reads and writes from inside SGX, but on memory that is located outside of the enclave (blue).

The most striking feature of the resulting plot is that suddenly, around 32 MB, access latencies are rising significantly for SGX reads and writes. We already mentioned the reason for this, the memory limit of 128 MB. At least for the Linux SDK, the Windows SDK does not support this, costly paging

will set in after this limit is reached, when the enclave will evict pages to untrusted memory, allowing the enclave to handle memory larger than 128 MB. Paging is already an expensive operation in, for example, the classical operating system context. But we have an additional overhead here, because the enclave needs to encrypt and integrity protect pages before it can securely evict them to untrusted memory. Also note that the performance degradation associated with the paging mechanism sets in way before a working set size of 128 MB. This is because other things like the code itself already take away memory inside the enclave. Interestingly, we can see that reads from inside the enclave on untrusted memory are not much slower than normal reads and writes. We want to use this insight later on for a modified version of our lockmanager.

From this initial experiment we derive the design requirement that we want to be memory-efficient and if possible find a solution to circumvent that we reach the memory limit and prevent costly paging from draining performance.

3.4.3 Multi-threading Support

It is very hard to design Intel SGX for high throughput workloads. We defer most optimizations regarding performance to future work. We will highlight some possible directions in Chapter 7. One of the performance optimizations we do consider, however, is to employ multi-threading inside the enclave. A couple of peculiarities about SGX threading have to be noted though:

- SGX does not support dynamic threading. Threads cannot be created or destroyed flexibly, but the number of threads has to be determined before starting the enclave and does not change.
- Threads are mapped to logical processors, so the number of concurrent threads is limited to the number of logical CPUs of the machine
- SGX hardware has no thread synchronization primitives, it relies on the thread synchronization primitives provided by the operating system

We aim to make our design enable utilizing multiple threads and reach a performance gain through multi-threading. How we exactly achieve this is described in Section 4.3.2.

3.4.4 Lockmanager Architectures

Leading over to the implementation section, this is how we want to implement our lockmanager:

1. **Insecure Lockmanager**: First, we start with an implementation without Intel SGX, mainly to be able to benchmark the overhead of Intel SGX and our defense mechanisms.
2. **Demand Paging Lockmanager**: Then we come up with a solution where we put everything, including lock tables a.s.o, inside the enclave and rely on Intel SGX's costly paging mechanism, when we reach the memory limit.
3. **Out-Of-Enclave Lockmanager**: Here, we store the lock table outside of the enclave. The lock table is the part of the lockmanager that takes up most space and is growing dynamically depending on the workload. By not putting the table into the enclave, we hope to avoid paging setting in. The experiment done in Section 3.4.2 on *Memory Limitations* shows that there is no significant overhead when accessing untrusted memory from the enclave. A consequence is

that the enclave now operates on untrusted memory when executing lock and unlock requests, which is a problem when an attacker can tamper with the contents of the lock table at his will. The solution to this problem is inspired by Shieldstore's approach [23]. We need to implement some form of *Integrity Verification* (see Section 4.3.1), which is a mechanism that maintains metadata inside the enclave that enables the enclave to determine that it accesses untampered data from the tables in untrusted memory. When an attacker would make changes inside the lock table, the enclave would be able to detect it using the stored metadata.

4 Implementation

In this section, we will describe the implementation of three different versions of our lock manager. Our goal with the implementations is to mainly have a way to evaluate the overhead that comes with Intel SGX and the memory efficiency of the out-of-enclave approach. We will start with the simplest form, the *Insecure Lockmanager*, which is a lock manager that implements pessimistic concurrency control via 2PL and exposes a gRPC [13] interface. Then, we follow up with the *Demand Paging Lockmanager*, which is the same implementation, but within Intel SGX. Lastly, the *Out-Of-Enclave Lockmanager* is a modification of the Demand Paging Lockmanager, that stores tables outside of the enclave.

From the defense mechanisms in Section 3.3, we implement only the lock and transaction budgets, as well as lock signatures. Lock signatures are the most cost-intensive operations among the defenses, requiring to compute signatures for every successful lock request, so they are the only relevant defense to implement with respect to the overhead we want to evaluate.

4.1 Insecure Lockmanager

4.1.1 Main Components

The three basic components of the basic, insecure lock manager implementation are a gRPC client, a gRPC server and a lock manager class that works with the following abstractions:

- **Lock Structs**: Maintain a set of transaction IDs of the transactions that hold the lock. One can request shared or exclusive access to the associated data item via that lock, as well as upgrading the lock from shared to exclusive and releasing it
- **Transaction Structs**: Have a unique ID, track the lock budget, are either in a growing or shrinking phase according to 2PL and keep track of the rows the transaction has a lock on
- **Transaction Table**: Maps transaction IDs to transaction structs
- **Lock Table**: Maps row IDs identifying data items in the storage layer to lock structs

4.1.2 Lock Table Design

The underlying data structure for the lock table, and also for the transaction table, is a hash table data structure. The main design decision that has to be made with hash tables is which hashing scheme to use, especially how to handle collisions. All of these schemes offer a trade-off between table size and additional overhead in finding and inserting keys into the table.

In the literature, it is mainly differentiated between static and dynamic hashing schemes. Static hashing schemes can be used when the number of elements that are supposed to be stored in the table are known upfront, because one slot for each element is allocated and we can determine an upper bound of elements through our transaction and lock budget. Secondly, the keys have to be unique, and transaction and row IDs are unique. However, we thought of static schemes to be just too inflexible for such dynamic content as locks in a database system, and in practice, dynamic schemes get mostly implemented as well.

We decided for simple *Chained Hashing* for collision avoidance, i.e., we maintain a linked list on entries, also called *buckets*, in the lock table, where values are stored, whose keys map to the same entry in the hash table.

4.1.3 Multithreading and Job Queue

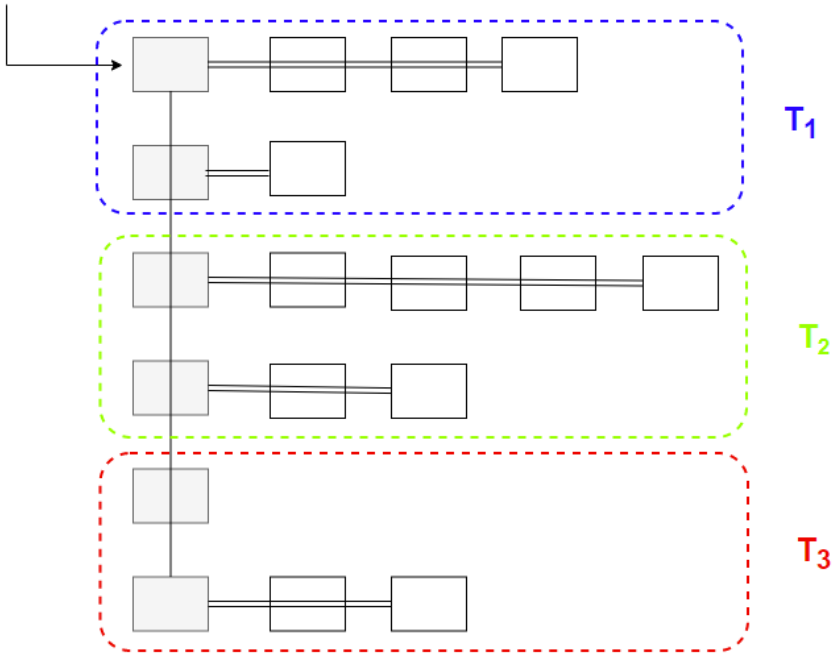

Figure 4.1: Multi-threading for the Hash Table

To enable multi-threading, we partition the lock table by the number of available threads. Each thread only deals with requests for locks that will be placed inside the partition it is responsible for. This is done by maintaining a job queue for each thread. The job queue contains job structs that describe the operation the thread needs to do, e.g. acquire or release a lock, together with the necessary

parameters. For efficiency reasons, we could also make use of a buffer, where we preallocate those job structs on the heap, to avoid costly memory allocations being in our hot path. The idea behind this implementation decision of partitioning the hash table between the worker threads is to avoid thread synchronization, as this will be a a costly operation in the enclave design later on, refer to Section 3.4.3 on *Multi-Threading inside the enclave*.

For a visualization, Figure 4.1 shows the hash table with chained hashing to resolve collisions. The grey boxes represent the head entries of the linked lists or buckets, while the white boxes are the other entries in that bucket. We see three different worker threads T_1, T_2 and T_3, differentiated by color, and each gets assigned its own partition of the hash table. E.g., worker T_1 gets assigned the first two buckets, a.s.o.

4.2 Demand Paging Lockmanager

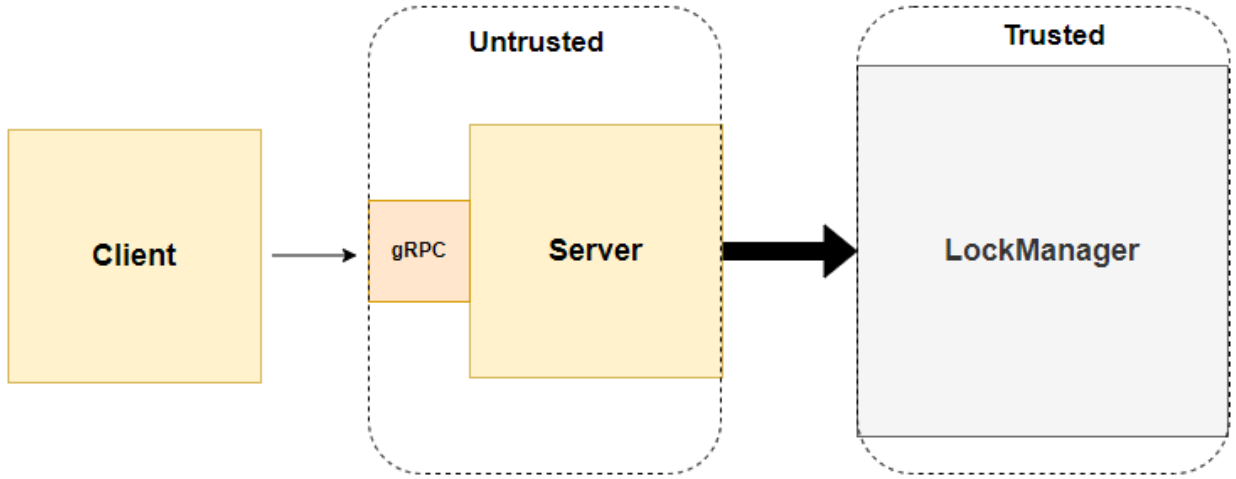

Figure 4.2: Partitioning the Lockmanager into trusted and untrusted Part

Now, we go on to the implementation of the lock manager inside the enclave, which is based on the lock manager implementation discussed previously. Here, we have to partition the application in an untrusted and trusted part. All of the lock manager components, including lock and transaction table, are stored inside the enclave. The gRPC-Server communicates with the lock manager inside the enclave via ECALLS, see Section 3.4.1. It will mainly utilize the enclave_send_job() ECALL to push a job into the corresponding worker queue inside the enclave to send lock requests to the enclave.

Another major change is the introduction of lock signatures, for which we use the SGX cryptography library, that provides us with an implementation of Elliptic Curve Digital Signature Algorithm (ECDSA) for signing locks. Because they are based on elliptic curves, signatures and public/private keys are just a point on that curve and not as many bits are necessary to represent them as compared to for example signatures based on RSA, which is very good for the memory-efficiency that we strive for. We also implement sealing of the keys used for signing so that they are preserved in main memory even when the lock manager has to restart (compare Section 2.3.3 on sealing in Intel SGX).

Notice that once the memory limit of the enclave is reached, costly paging will set in, as demonstrated in Section 3.4.2. This is why we designed another architecture for the lock manager, where the lock and transaction tables are stored outside of the enclave, described next.

4.3 Out-Of-Enclave Lockmanager

The transaction table and of course foremost the lock table are the two components of the lock manager which consume the most amount of memory and where the memory can grow basically indefinitely based on the client requests. We therefore build an alternative architecture, where the transaction and lock table are stored outside the enclave, in untrusted memory. The enclave can access the tables via pointers that are given to it via an initial ECALL. Notice that we had to implement our own hash table struct for that purpose, because using classes, e.g. from the C++ standard library, does not work with ECALLs. The enclave is able to read and write to the tables via the pointer, but is not able to allocate more memory in the untrusted region, therefore we allocate empty locks and transactions in the untrusted part before we make the call to the enclave.

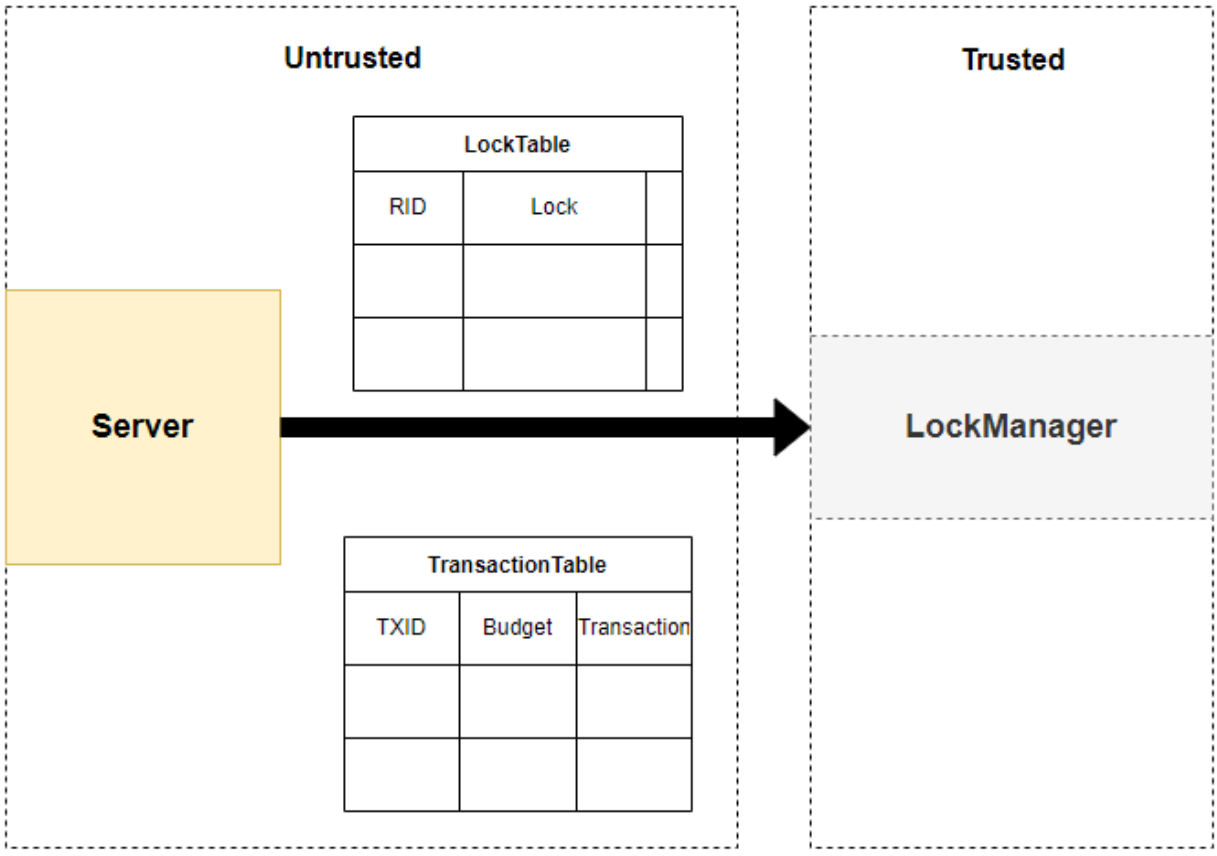

Figure 4.3: Putting the Lock and Transaction Table in unprotected Memory

Storing the tables outside of the enclave, raises a security problem, however. According to our attack model, the contents of the table can be modified by an adversary at any time. This has two major consequences:

1. We have to create a secure copy before we want to make a change. That is, we have to copy at least the lock we want to work on into trusted memory and make our operation, e.g. checking if the lock is not exclusive and add a new owner to the lock, on that trusted copy. After that, we can copy the changes back into untrusted memory.
2. When reading from the table, we have to have a way to ensure that its content has not been tampered with. We refer to those approaches as *Integrity Verification*.

Note that both considerations trade additional performance overhead for more memory inside the enclave and an unlimited size for the lock and transaction table in untrusted memory, making lock and transaction budgets unnecessary.

4.3.1 Integrity Verification

To verify the integrity of the tables in untrusted memory, we aim to compute some sort of digest that we can store inside the enclave. Such a digest can be a cryptographic hash function, which we have already introduced in Section 2.2.1 on Blockchain block structure. We now want to analyze, which possible attacks our verification mechanism has to defend against.

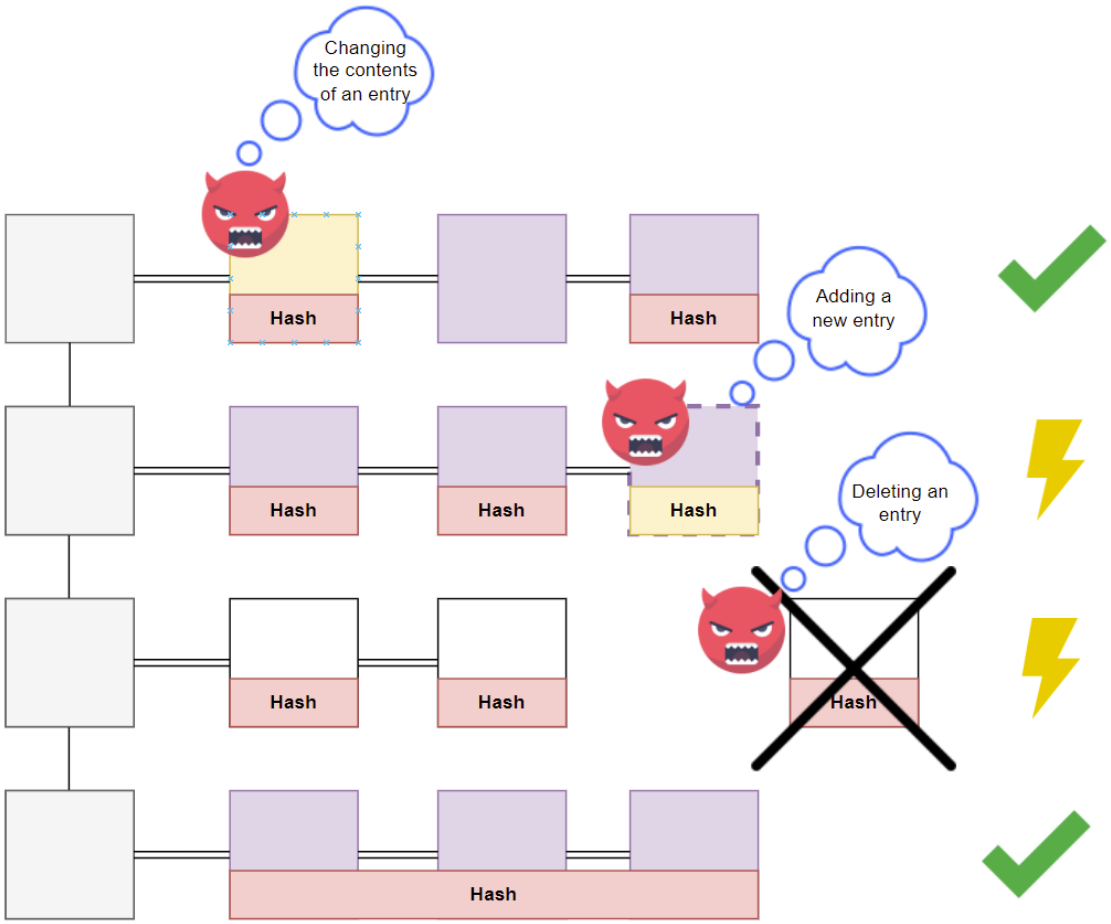

Figure 4.4: Possible Attacks on Integrity Verification

We start out with a simple approach, where we hash each entry in the hash table, ignoring the fact that storing hashes per element would anyway add too much of a memory overhead. An attacker cannot change an individual entry anymore, because this would change the hash. When we recompute the hash on the copy of the untrusted memory, this will yield a different hash then the one we have stored inside the enclave and we are able to detect the change. However, attacks that this does not defend against, is adding a new entry, an attacker can compute the hash by himself, and we also don't detect deletes of entries, which is visualized in Figure 4.4 in the second and third row.

Therefore, we decided to compute a hash over each bucket, making the number of hashes fixed, as the size of the hash table, i.e. the number of buckets, is also fixed. It is easy to see that having a hash over the whole bucket defends against all the attacks above, however, we now have to copy in the whole bucket into trusted memory to operate on it in a secure environment and to recompute the hash. To optimize this, we do not copy the bucket, but serialize it into an array of integers instead, which takes up much less memory. We also only allocate the memory for that array only once and reuse it for subsequent requests.

We want to note that another possibility is to make use of digital signatures instead of hashes. This has the advantage that the hashes do not have to be stored for comparison inside the enclave anymore, but only the secret key used to generate the signatures. An attacker cannot forge a valid signature for a fake entry and cannot modify the entries, as this will invalidate its signature. Therefore, we only need to verify the signature in untrusted memory. However, we decided against this, because the storage we save is not that critical, because it is bound to the fixed number of buckets in the table and we later confirm that the memory consumption of that is negligible, even within the enclave. Finally, signing and verifying the signature of an entry is much more labor intensive than hashing.

4.3.2 Multi-threading Extension

We did not implement multi-threading in the out-of-enclave implementation. However, it can be easily extended to operate with multiple threads the same way as the previous implementations do. As threads always operate on separate buckets, there are no conflicts when computing and updating the integrity hashes. Each thread just needs its own array inside the enclave to serialize the bucket from untrusted memory into. The memory overhead should be negligible, as for example for four threads we only have to store four serialized buckets, while the lock table that resides outside of the enclave contains 10000 buckets in our default setting.

In the next section, we can show in our experiments that the ouf-of-enclave approach drastically reduces the memory requirements inside the enclave. And because of the lack of paging overhead when permanently staying within the EPC limit, it is even faster than the demand paging approach.

5 Evaluation

In this chapter we want to evaluate the implementation of our lockmanager architectures presented in the previous chapters. Specifically, the results we want to show are the following:

1. There is a large performance overhead of using trusted hardware compared to the insecure variant of our lockmanager (Section 5.2).
2. We can achieve an increased throughput with multiple threads, in the insecure variation as well as in the SGX implementation (Section 5.3).
3. There is a high memory consumption inside the enclave, when we store the lock table in trusted memory, but we reduce memory consumption in the Out-of-Enclave Lockmanager (Section 5.4).
4. When the memory limits of the enclave is reached, the performance degrades, but with the Out-of-Enclave Lockmanager, we never reach the memory limits, because the lock table is stored in untrusted memory. Moreover, the overhead of the necessary integrity verification procedure on every lock table access is smaller than the paging overhead (Section 5.5).

5.1 Experiment Design

To reiterate, we implemented three lockmanager architectures:

- Insecure Lockmanager, without Intel SGX (Section 4.1)
- Demand Paging Lockmanager, completely within Intel SGX (Section 4.2)
- Out-of-enclave Lockmanager, within Intel SGX, but the lock table that consumes most memory is stored outside of the enclave (Section 4.3)

The only component that is growing dynamically in memory depending on the workload, is the lock table. If more and more locks are acquired, the memory consumption will eventually reach the memory limit of the EPC and paging will occur, which should lead to a drop in performance. The paging overhead should be measurable in the Demand Paging Lockmanager, but should not be present in the out-of-enclave approach, because here we store the lock table outside of the enclave. Accessing unprotected memory per se does not have higher access latencies, as shown in Figure 3.4, but the necessary integrity verification does, which should be lower than the paging overhead. We design a simple experiment that is aimed to cause higher and higher memory consumption to gradually reach and surpass the EPC limit of the enclave and provoke paging.

The idea is to measure the time it takes to acquire an increasing number of locks. We do not send the requests from the gRPC client, but make direct calls to the lockmanager instance, as prior

measurements have shown that gRPC contributes to over 97% of the total runtime. This would not only cause experiments to last very long, but would also overshadow other bottlenecks, i.e., paging. Acquiring more locks sequentially consumes more and more memory, as memory needs to be allocated for the lock objects stored inside the lock table. At a certain number of locks, the EPC of the enclave will be full. We also do measurements of the memory consumption within the enclave later on, to show that this is indeed the case and to determine how many locks are needed to fill up the cache.

Lock requests are issued in shared mode for reads by a single transaction. Then we start another transaction that acquires the same locks in shared mode as well, so that potentially locks on EPC pages that previously got evicted need to be loaded into the enclave again.The resulting run times in nanoseconds are then used to compute the throughput in locks per second.

We also conduct the same experiment with multiple threads in the Insecure Lockmanager and the Demand Paging Lockmanager by starting multiple worker threads inside the enclave that operate separately on requests targeting their dedicated portion of the lock table. Note that this is only leading to concurrency inside of the enclave part. We are not evaluating concurrent requests in general, i.e., a scenario where multiple clients send requests to the lockmanager concurrently. This will be deferred for future work.

5.2 Intel SGX Overhead

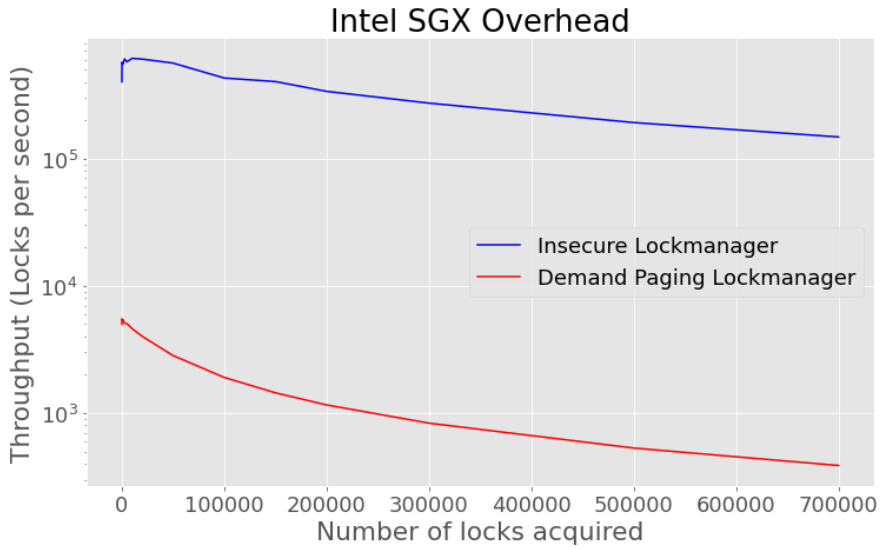

Figure 5.1: Throughput Degradation with Intel SGX

We execute the experiment described in Section 5.1 on both the Insecure Lockmanager and the Demand Paging Lockmanager, acquire a growing number of locks, up to 700.000 locks, on the x-axis, measure the time it takes for both transactions to acquire the respective number of locks sequentially and derive from this data the throughput in locks per second, on the y-axis (see Figure 5.1). Note that the y-axis has logarithmic scale. It becomes apparent that the throughput of the Demand Paging Lockmanager is much lower. Because, apart from the fact that one is implemented with Intel SGX,

the implementations are the same, this difference in throughput has to be attributed to Intel SGX's performance overhead.

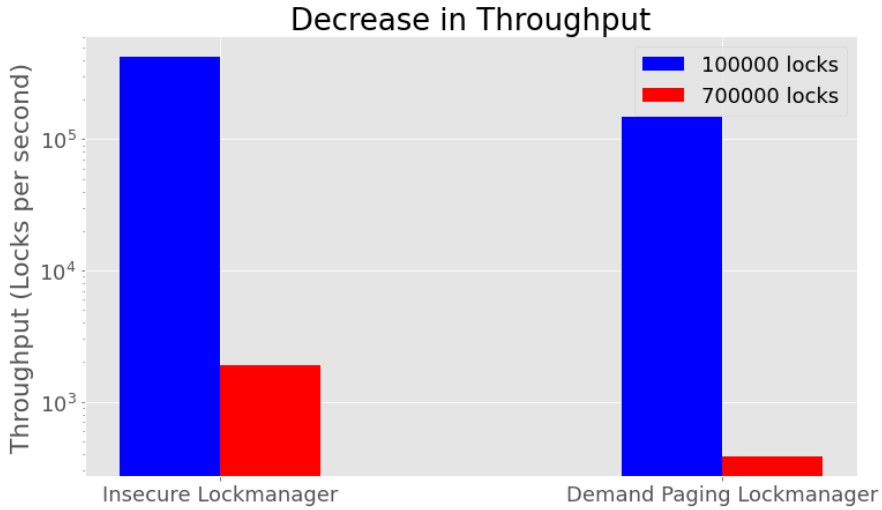

Figure 5.2: Degradation of Throughput with an increasing number of Locks

We note that for both implementations, the throughput is degrading with a growing number of locks, even in the Insecure Lockmanager without Intel SGX. We identified the reason for this to be the growing number of collisions inside the hash table. As the size of the hash table is fixed to 10000, collisions will definitely occur when we acquire more than 10000 locks. Because we use *Chained Hashing* for collision resolving, this leads to longer and longer linked lists or buckets inside the table that have to be traversed when looking for an item. This leads to a degrading throughput with a higher number of locks. However, as can be seen in Figure 5.2, the throughput from 100.000 to 700.000 locks is decreasing much stronger in the Demand Paging Lockmanager. So there has to be another factor that influences the throughput for a higher number of locks that is not present in the Insecure Lockmanager, without Intel SGX. Later, in Section 5.5, we will identify this to be the paging overhead, setting in when the memory limit of the enclave is reached.

5.3 Performance Gains via Multithreading

We now want to show that we can improve the throughput by adding additional worker threads inside the lockmanager that operate on the lock table, as described in Section 4.1.3. We execute the same experiment, with a fixed number of 100.000 locks, but with 1, 2, 4 and 8 threads for the Insecure Lockmanager and Demand Paging Lockmanager respectively. We can see the results in Figure 5.3. The throughput is increased by adding more worker threads for the Insecure Lockmanager until 4 threads, but degrading a little again with 8 threads. The throughput is getting better until a certain number of threads where each single thread is not working to capacity and the performance gain by additional workers gets outweighted by the additional synchronization overhead. The point where the throughput decreases again is already reached with 4 threads in the Demand Paging Lockmanager. We can infer a higher overhead in thread synchronization inside Intel SGX from this. And indeed, because thread synchronization primitives use operating system functionality, it requires a costly OCALL, see the Intel Software Guard Extensions Developer Guide on *Calls outside the enclave* [19].

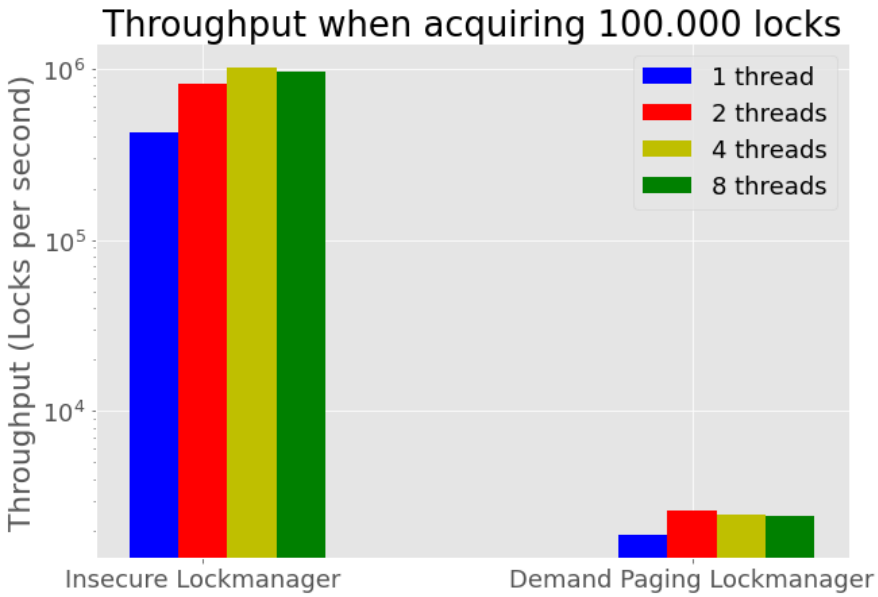

Figure 5.3: Increase in Throughput by adding more Workers

5.4 Memory Consumption inside the Enclave

We implemented the Out-of-Enclave Lockmanager, where the lock table that is consuming the most amount of memory is stored outside of the enclave. We want to measure the stack and heap consumption to proof that the memory consumption inside the enclave of the Demand Paging Lockmanager is growing linearly with the number of locks and is surpassing the EPC size at some point. On the contrary, the memory consumption of the enclave in the Out-of-Enclave Lockmanager should be constant and should not surpass the EPC size, because the dynamically growing lock table is no longer stored inside the enclave memory.

In Figure 5.4, we measured peak heap and stack usage inside the enclave for both the Demand Paging Lockmanager and the Out-of-Enclave Lockmanager when acquiring varying number of locks using a tool called *SGX emmt* [20]. The black dashed horizontal line is set to 32 MB and represents the point that paging started in the previous experiment on memory access latencies in Figure 3.4. So we assume paging to set in around that time as well, potentially also a little earlier, since our code base etc. is much larger, therefore even less memory will be available inside the enclave. In the plot, we can see that peak stack sizes are constant for both architectures and also the peak heap size for the Out-of-Enclave Lockmanager is about constant, regardless of the number of locks. The peak heap size in the demand paging approach, however, is increasing heavily, reaching the 32 MB border at around 150.000 locks. This clearly visualizes that with the lock table stored inside that enclave, the peak heap size is growing linearly, as more and more locks need to be stored inside the enclave. When the lock table is stored outside of the enclave, this growth does not affect the memory consumption inside the enclave and it stays way below any EPC limit or paging threshold.

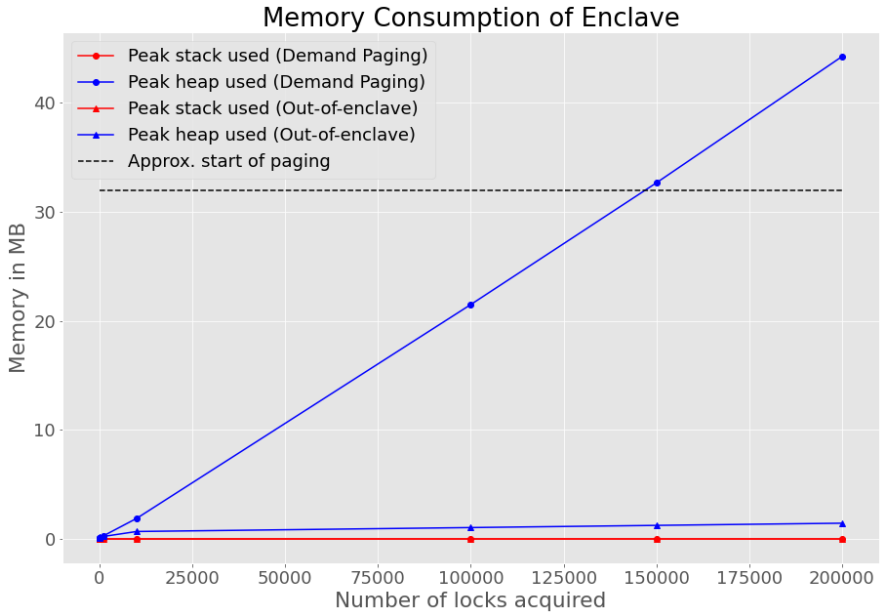

Figure 5.4: Memory Consumption for the Demand Paging Lockmanager and Out-of-Enclave Lockmanager

5.5 Paging Overhead

We have seen that the throughput in the Intel SGX implementation is degrading stronger than in the implementation without Intel SGX (Figure 5.2), which we attributed to the paging overhead. We have also seen that the Demand Paging Lockmanager surpasses the memory limit of the enclave at a certain number of locks (Figure 5.4). Therefore paging will definitely set in. Our goal is to show that the overhead through paging has a noticeable influence on the performance. Therefore, we measure the throughput with an increasing number of locks of the Demand Paging Lockmanager and the Out-of-Enclave Lockmanager, which does not experience any overhead from paging. However, in the Out-of-Enclave Lockmanager, the integrity verification on every access on the lock table adds overhead. That is why we also measured the throughput for a modified Out-of-Enclave Lockmanager, where we are not doing integrity verification, to be able to see the performance difference.

Now when comparing the throughput of both the Out-of-Enclave Lockmanager and the Demand Paging Lockmanager in Figure 5.5, we can clearly see the overhead induced by paging. The throughput in the Demand Paging Lockmanager is decreasing drastically, while in the Out-of-Enclave Lockmanager it is only decreasing lightly due to a growing number of collisions in the hash table. By avoiding the overhead of paging, we achieved a much higher throughput in the Out-of-Enclave Lockmanager. When comparing its throughput with the modified version without integrity verification, we can see that the throughput is not impacted much. We can infer that the integrity verification only adds a marginal overhead.

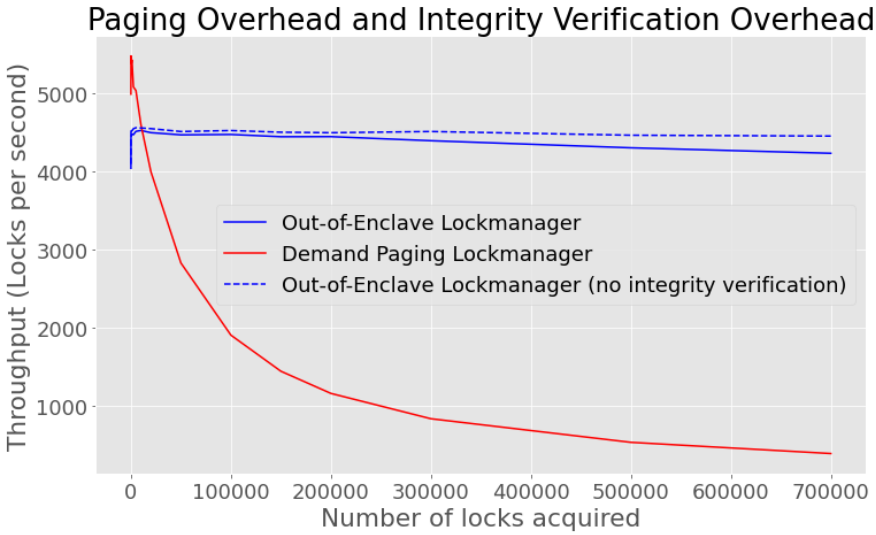

Figure 5.5: Rising Runtime due to Paging

6 Related Work

In this section, we present two related works. First, we look at ShieldStore [23], which is an efficient key-value store inside Intel SGX. We present it, because it has many similarities to our approach from a implementation perspective and we took huge inspiration from it, especially for our out-of-enclave design. Second, we introduce Basil [42], which follows the same goal of achieving byzantine fault-tolerant concurrency control, but promotes an opposite approach, namely an optimistic concurrency control scheme based on timestamps and multiple versions.

6.1 ShieldStore: Shielded In-Memory Key-value Storage with SGX

ShieldStore [23] is an in-memory key-value store designed for the Intel SGX platform, so it is very similar in principle to our lock manager, since its internal tables for locks and active transactions can be viewed as a simple key-value store as well. Assuming that the data definetely will surpass the memory limit of SGX, the main data structures are placed in unprotected memory, keys and values encrypted and integrity-protected with only cryptographic keys and meta-data to verify integrity stored inside the enclave.

Being a key-value store, their interface is naturally different from our lockmanager. They offer reads and writes of string values to string keys. However, their architecture served as an inspiration for our out-of-enclave lock manager implementation in Section 4.3, so there is a lot of overlap with their design, which we will summarize first:

- ShieldStore uses *Chained Hashing* for its hash table implementation, i.e., collisions on the same key are stored in a linked list, called a *bucket*.
- They adopt multi-threading by assigning a thread to a set of buckets, which is then responsible for all requests on a key inside that bucket.
- The hash table is initialized and maintained in untrusted memory. The enclave gets a pointer to that hash table via an initial ECALL and just maintains metadata for integrity verification in trusted memory. On every read request, the content of the corresponding bucket has to be verified as well as before every write request, but additionally, after a write request, the metadata has to be updated accordingly.

The main difference comes from the way the integrity verification is performed. ShieldStore, as a general key-value store, also considers the confidentiality of the data, while this is not relevant to our lock manager. Therefore, they also have to encrypt the data stored inside the hash table. While for our lock manager, it suffices to compute a hash over the whole bucket, ShieldStore has to employ authenticated encryption, which has a higher performance overhead, to compute a MAC for each data

entry and subsequently store the hash over all MACs in a bucket inside the enclave. Additionally, a specific key inside a bucket cannot be searched without decrypting all data entries.

ShieldStore on the other hand implements a couple of additional optimizations that we did not consider in our work:

- To avoid having to decrypt the whole bucket when reading a key, they include a key hint, one byte of the hash of the key value as a field in the data entry, lowering down the number of data entries that have to be encrypted to find a key in exchange for leaking a small amount of information
- It implements a custom heap allocator, since per default, there exist the default heap allocator in the untrusted memory region and one within the enclave to allocate heap memory within the trusted region. That means allocating untrusted memory from within the enclave requires a costly exit from the enclave via an OCALL. To avoid this overhead, a custom memory allocator running inside the enclave is added, which is able to allocate untrusted memory. It does this by utilizing a pre-allocated memory pool.
- ShieldStore considers persistency support for its data

In their evaluation, they showed that ShieldStore has 7-10 times higher throughput than the baseline and 21-26 times higher throughput, when the number of threads is increased to 4. The baseline hereby is the naive solution, where everything is put inside the enclave and the enclave's internal paging mechanism has to be used to accomodate for the fact that not all enclave data fits into the memory limit of 128 MB. They also compare that to an insecure baseline without Intel SGX and report to be 17.7 and 39.8 times slower with 1 thread and 4 threads, respectively. These numbers are very similar to the results of our evaluation.

6.2 Basil: Breaking up BFT with ACID transactions

Basil [42] also wants to achieve Byzantine fault tolerant concurrency control like we do with our locking-based approach, i.e., secure sharing of data with ACID guarantees despite malicious actors that can arbitrarily deviate from the protocol and want to violate data integrity. It is however also very different from our work, as it achieves this goal with a complimentary approach. It does not make use of trusted execution environments and it does not follow a locking-based protocol, rather an optimistic concurrency protocol based on timestamps, as introduced in Section 2.1.5.

Two important notions are introduced at the beginning of the paper: *Byzantine Isolation* ensures that benign client always observe a state of the database that could have been only produced by benign clients, i.e., a state that is ACID compliant, and *Byzantine Independence* means that malicious clients cannot determine whether a transaction commits or aborts. It is important to avoid that a Byzantine actor is able to abort all transactions systematically. Byzantine independence therefore requires a leaderless scheme as a malicious leader can abort transactions freely.

More concretely, they implement a variant of Multiversion Timestamp Ordering (MVTSO) that fulfills both properties, an optimistic concurrency control protocol that is particularly efficient under the assumption of a mostly conflict and tamper free transaction execution. They claim that locking-based protocols are not that well suited as they pose the inherent problem of Byzantine lock holders preventing the progress of other transactions, and indeed, we had difficulties with exactly those points

in the design of our lockmanager. We introduced lock budgets and timeouts, but it is difficult to determine those numbers upfront.

MVTSO works like the following: Each transaction T gets assigned a unique timestamp ts_T, which determines its order for serialization. Writes of T are multi-versioned, each creates a new version of the data objects they work on, tagged with the timestamp ts_T, while reads always return the version of the object with the highest timestamp smaller than ts_T and update their *read timestamp* RTS to ts_T. The key to preserving serializability in this way is to ensure that reads are not missing writes from a preceeding transaction. Therefore, writes from a transaction with ts_t smaller than the RTS of an object they attempt to read have to be aborted.

Under byzantine actors, one can identify two main attacks on this protocol: Byzantine clients can manipulate the timestamps, specifically, they can choose arbitrarily high timestamps and make reads to a lot of data objects. This sets the RTS of those data objects to a very high value and makes subsequent transactions with lower timestamps abort. Or byzantine clients could write to a large number of data objects and never commit the corresponding transaction, blocking transactions that depend on those writes. The first attack is dealt with by having replicas accept operations only when their timestamp does not differ from their own local clock more than a certain fixed amount. The second attack is circumvented by having clients buffering writes locally until the transaction has finished execution and only making them visible in a later stage of the protocol, avoiding dependencies between transactions. More specifically, the transaction processing is divided into three phases:

1. **Execution Phase**, where clients execute transactions individually, sending reads directly to remote replicas but buffering writes locally
2. **Prepare Phase**, where shards vote on if the transaction can commit or needs to abort because it would violate serializability
3. **Writeback Phase**, where the client collects the votes, cryptographically protected by certificates, determines the final decision and forwards it to the replicas, which can subsequently apply the buffered writes on their data

Basil claims to improve the throughput over traditional Byzantine Fault-tolerant systems by four to five times and being only four times slower than TAPIR [47], which is a distributed database, that is not byzantine fault-tolerant. However, we cannot compare their results with ours, as we did not measure the throughput in transactions per second.

7 Conclusion and Future Work

This chapter concludes the contributions of this thesis. Then, it presents further research challenges for future work.

7.1 Conclusion

We designed a Byzantine-Fault Tolerant Lockmanager in the context of a Blockchain based decentralized data sharing system and described in detail its different attack vectors and possible countermeasures, differentiating between clients that target availability and integrity and the underlying host system of the lockmanager, targeting integrity. Afterwards, we implemented the lockmanager within a trusted execution environment and analyzed its performance. The paging overhead that occured due to the limited amount of enclave memory turned out to be a major bottleneck. So we developed an optimization to better cope with Intel SGX's tight memory restrictions, by putting the lock table, the portion of the lock manager that consumes the most memory and is dynamically growing depending on the workload, outside of the enclave. This entails an additional integrity verification mechanism that ensures that data read from the table has not been tampered with. In the evaluation, we showed that this optimization indeed avoids the paging overhead and shows to be two to eight times faster than our initial design.

7.2 Future Work

Although we managed to decrease the performance overhead of paging, there is still a huge gap between the implementation with and without SGX. Therefore, for future work, the goal should be to further optimize the SGX implementation to be closer to the insecure version. One particular bottleneck are enclave switches, i.e., whenever control is transferred between the trusted and untrusted part of the application, triggered via ECALLS and OCALLS, which are more than 50 times more expensive than system calls. We can infer that this is specifically problematic in high-throughput workloads like in our lockmanager. One proposed alternative are so called *switchless calls* that avoid this transition in and out of the enclave, by making use of shared memory between the trusted and untrusted part and use worker threads to execute function calls asynchronously [43]. We need to investigate this and further possibilities to optimize Intel SGX for high throughput.

Furthermore, we want to adapt our lockmanager architecture: Right now, we follow a centralized approach, where we have a single lockmanager residing on a single node that accepts lock requests from all clients. This does not scale well, especially when we would decide to employ sharding in the underlying trusted storage layer. As already mentioned in the background section (Section 2.2),

sharding is a reasonable option to scale Blockchain solutions. A centralized lockmanager quickly becomes a bottleneck since all requests need to be processed there. It is also a single point of failure. If that node failed, than the whole concurrency control would be lost and transactions of the whole system would need to come to a halt. Consequently, the lockmanager should be distributed over several nodes, where each node manages a local lockmanager that is responsible for the locks of a specific shard of the data. The main component that needs to be modified for the distributed lockmanager is the deadlock handling component, introduced in Section 2.1.4. Deadlock prevention techniques work analogously, but deadlock detection techniques via wait-for-graphs are challenging, because a local cycle-free graph does not imply a global cycle-free graph. Solutions exist, but are complicated and expensive [41]. We need to evaluate different approaches and benchmark the lockmanager under a realistic workload, e.g., YCSB [9].

Bibliography

[1] Tigist Abera et al. “C-FLAT: control-flow attestation for embedded systems software”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 743–754.

[2] Rakesh Agrawal, Michael J Carey, and Larry W McVoy. “The performance of alternative strategies for dealing with deadlocks in database management systems”. In: *IEEE Transactions on Software Engineering* 12 (1987), pp. 1348–1363.

[3] Ittai Anati et al. “Innovative technology for CPU based attestation and sealing”. In: *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*. Vol. 13. Citeseer. 2013, p. 7.

[4] Adam Back et al. “Hashcash-a denial of service counter-measure”. In: (2002).

[5] Hal Berenson et al. “A critique of ANSI SQL isolation levels”. In: *ACM SIGMOD Record* 24.2 (1995), pp. 1–10.

[6] Miguel Castro, Barbara Liskov, et al. “Practical byzantine fault tolerance”. In: *OSDI*. Vol. 99. 1999. 1999, pp. 173–186.

[7] Guoxing Chen et al. “Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution”. In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, pp. 142–157.

[8] *Concurrency Control Theory*. `https://15445.courses.cs.cmu.edu/fall2019/schedule.html`. Accessed: 2021-10-30.

[9] Brian F Cooper et al. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154.

[10] Victor Costan, Ilia Lebedev, Srinivas Devadas, et al. *Secure processors part I: background, taxonomy for secure enclaves and Intel SGX architecture*. Now Foundations and Trends, 2017.

[11] Tien Tuan Anh Dinh et al. “Untangling Blockchain: A Data Processing View of Blockchain Systems”. In: *IEEE Transactions on Knowledge and Data Engineering* 30.7 (2018), pp. 1366–1385. DOI: `10.1109/TKDE.2017.2781227`.

[12] John R Douceur. “The sybil attack”. In: *International workshop on peer-to-peer systems*. Springer. 2002, pp. 251–260.

[13] *gRPC*. `https://grpc.io/`. Accessed: 2021-12-29.

[14] Mike Hamburg. “Fast and compact elliptic-curve cryptography.” In: *IACR Cryptol. ePrint Arch.* 2012 (2012), p. 309.

[15] Hazar Harmouch and Felix Naumann. “Cardinality estimation: An experimental survey”. In: *Proceedings of the VLDB Endowment* 11.4 (2017), pp. 499–512.

[16] Muhammad El-Hindi et al. “BlockchainDB: A shared database on blockchains”. In: *Proceedings of the VLDB Endowment* 12.11 (2019), pp. 1597–1609.

[17] Muhammad El-Hindi et al. “TrustDBle: Towards trustable shared databases”. In: *Third International Symposium on Foundations and Applications of Blockchain*. 2020.

[18] Tyler Hunt et al. “Chiron: Privacy-preserving machine learning as a service”. In: *arXiv preprint arXiv:1803.05961* (2018).

[19] *Intel SGX Developer Guide*. `https://download.01.org/intel-sgx/linux-2.1.3/docs/Intel_SGX_Developer_Guide.pdf`. Accessed: 2021-12-27.

[20] *Intel SGX SDK*. `https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf`. Accessed: 2021-10-30.

[21] Simon Johnson et al. “Intel software guard extensions: EPID provisioning and attestation services”. In: *White Paper* 1.1-10 (2016), p. 119.

[22] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC press, 2020.

[23] Taehoon Kim et al. “Shieldstore: Shielded in-memory key-value storage with sgx”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–15.

[24] Paul Kocher et al. “Spectre attacks: Exploiting speculative execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1–19.

[25] Leslie Lamport. “The part-time parliament”. In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 277–317.

[26] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine generals problem”. In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 203–226.

[27] Moxie Marlinspike. “Technology preview: Private contact discovery for Signal”. In: *Retrieved April* 29 (2017), p. 2018.

[28] Frank McKeen et al. “Innovative instructions and software model for isolated execution.” In: *Hasp@ isca* 10.1 (2013).

[29] Ralph C Merkle. “A digital signature based on a conventional encryption function”. In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

[30] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. “Cachezoom: How SGX amplifies the power of cache attacks”. In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2017, pp. 69–90.

[31] Chandrasekaran Mohan et al. “ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging”. In: *ACM Transactions on Database Systems (TODS)* 17.1 (1992), pp. 94–162.

[32] Satoshi Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: *Decentralized Business Review* (2008), p. 21260.

[33] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. “A survey of published attacks on Intel SGX”. In: *arXiv preprint arXiv:2006.13598* (2020).

[34] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319.

[35] Marshall Pease, Robert Shostak, and Leslie Lamport. “Reaching agreement in the presence of faults”. In: *Journal of the ACM (JACM)* 27.2 (1980), pp. 228–234.

[36] Christian Priebe, Kapil Vaswani, and Manuel Costa. “EnclaveDB: A secure database using SGX”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 264–278.

[37] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. “Trusted execution environment: what it is, and what it is not”. In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. IEEE. 2015, pp. 57–64.

[38] Carlton Shepherd, Konstantinos Markantonakis, and Georges-Axel Jaloyan. “LIRA-V: Lightweight Remote Attestation for Constrained RISC-V Devices”. In: *arXiv preprint arXiv:2102.08804* (2021).

[39] Shweta Shinde et al. “Preventing page faults from telling your secrets”. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 2016, pp. 317–328.

[40] Abraham Silberschatz. *Database System Concepts, 7th edition*. McGraw-Hill Education, 2019.

[41] Mukesh Singhal. “Deadlock detection in distributed systems”. In: *Computer* 22.11 (1989), pp. 37–48.

[42] Florian Suri-Payer et al. “Basil: Breaking up BFT with ACID (transactions)”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 1–17.

[43] Hongliang Tian et al. “Switchless calls made practical in Intel SGX”. In: *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. 2018, pp. 22–27.

[44] David Übler, Johannes Götzfried, and Tilo Müller. “Secure remote computation using Intel SGX”. In: *SICHERHEIT 2018* (2018).

[45] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-channel attacks: Deterministic side channels for untrusted operating systems”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 640–656.

[46] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. “Rapidchain: Scaling blockchain via full sharding”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 931–948.

[47] Irene Zhang et al. “Building consistent transactions with inconsistent replication”. In: *ACM Transactions on Computer Systems (TOCS)* 35.4 (2018), pp. 1–37.

[48] Wenting Zheng et al. “Opaque: An oblivious and encrypted distributed analytics platform”. In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 283–298.