Provenance Control and Auditability in Autonomous Driving



Sérgio Paulo de Sousa Moreira Neto
Mestrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2024

Provenance Control and Auditability in Autonomous Driving

Sérgio Paulo de Sousa Moreira Neto

Dissertação realizada no âmbito do Mestrado em
Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2024

Orientador
Bernardo Portela, Professor Auxiliar, Faculdade de Ciências da Universidade do Porto

Coorientador
Manuel Barbosa, Professor Associado, Faculdade de Ciências da Universidade do Porto

Acknowledgements

Em primeiro lugar, quero agradecer ao Professor Bernardo Portela por toda a ajuda, disponibilidade e paciência. Sem o seu acompanhamento e inúmeros conselhos, eu não teria conseguido terminar esta dissertação.

Quero também expressar a minha profunda gratidão à minha mãe e ao meu irmão, pelo apoio incondicional e pelos sacrifícios que fizeram para que eu pudesse seguir o percurso académico na área que eu adoro. Em especial quero agradecer ao meu irmão, Luís Neto, por todos os concelhos e ajuda ao longo do meu percurso, por me incentivar a querer aprender sempre mais e por me encorajar a explorar novas áreas de conhecimento.

Agradeço também à Mariana Lopes, por toda a paciência durante a realização desta dissertação, nos momentos mais difíceis a sua presença e apoio constantes ajudaram-me a chegar até aqui.

Por fim quero agradecer a todos os meus amigos e colegas, por todo o apoio e por me acompanharem ao longo do meu percurso académico. Em especial quero agradecer ao Eduardo Santos pela sua ajuda durante a realização deste trabalho.

Resumo

Num mundo onde cidades inteligentes com extensas redes de sensores recolhem dados relevantes para melhorar a gestão urbana e otimizar o fluxo de trânsito, a comunicação entre veículos e outras entidades (*Vehicle-to-Everything (V2X)*) é fundamental. A capacidade de manter uma visão global da cidade replicada entre veículos autónomos apresenta desafios semelhantes aos dos sistemas distribuídos, onde a necessidade de uma visão consistente pode comprometer a interatividade e a eficiência do sistema.

Uma possível solução para este desafio é o uso de Conflict-Free Replicated Data Types (CRDTs), estruturas de dados abstratas projetadas para replicação. Enquanto os sistemas distribuídos tradicionais com consistência forte exigem sincronização rigorosa entre réplicas, os CRDTs sacrificam essa consistência forte para obter disponibilidade contínua, assegurando consistência eventual. Isto permite que as réplicas operem de forma independente, mesmo sem estarem completamente sincronizadas, o que pode ser uma abordagem promissora para a comunicação entre veículos num cenário V2X.

Apesar da utilidade dos CRDTs na replicação de dados distribuídos, ainda é escassa na literatura uma abordagem formal para garantir a segurança destas estruturas, sendo as soluções existentes projetadas individualmente para cada construção. Nesta dissertação, propomos uma abordagem para habilitar segurança de CRDTs através do uso de Trusted Execution Environments (TEEs), viabilizado pelo uso de hardware confiável (Intel SGX), que permite transformar construções gerais de CRDTs em versões seguras de forma simples. Além disso, aproveitamos funcionalidades como a Atestação Remota para assegurar que entidades que participam no sistema, como veículos, não se desvião do protocolo e funcionamento pretendido.

Para validar a abordagem proposta, desenvolvemos uma aplicação que simula um cenário no qual a infraestrutura da cidade comunica com veículos que contêm hardware confiável a bordo. Estes veículos utilizam CRDTs como, contadores e conjuntos, para guardar dados sincronizados, que traduzam uma visão da cidade. Os nossos testes envolvem a simulação de vários carros que, circulam pela cidade, enquanto recebem dados (relacionados com o seu ambiente) arbitrários da infraestrutura. Na nossa conclusão, discutimos os resultados dos testes e realçamos a limitação de uma característica específica de hardware confiável.

Palavras-chave: CRDT, Segurança de Computadores, Intel SGX, TEE, Sistemas Distribuídos, V2X.

Abstract

In a world where smart cities rely on extensive sensor networks to collect relevant data for improving urban management and optimizing traffic flow, communication between vehicles and other entities (Vehicle-to-Everything, V2X) is crucial. The ability to maintain a global, replicated view of the city across autonomous vehicles presents challenges similar to those found in distributed systems, where the need for consistent views can compromise system interactivity and efficiency.

A potential solution to this challenge is the use of Conflict-Free Replicated Data Types (CRDTs), abstract data structures designed for replication. While traditional distributed systems with strong consistency require strict synchronization between replicas, CRDTs sacrifice strong consistency to achieve continuous availability, ensuring eventual consistency. This allows replicas to operate independently, even without being fully synchronized, making CRDTs a promising approach for communication between vehicles in a V2X scenario.

Despite the usefulness of CRDTs in distributed data replication, the literature still lacks a formal approach to ensure the security of these structures, with existing solutions being designed individually for each construction. In this dissertation, we propose an approach to enable the security of CRDTs through the use of Trusted Execution Environments (TEEs), made possible by trusted hardware (Intel SGX), which allows general CRDT constructions to be transformed into secure versions with ease. Additionally, we leverage features such as Remote Attestation to ensure that entities participating in the system, such as vehicles, adhere to the intended protocol and behaviour.

To validate the proposed approach, we developed an application that simulates a scenario in which city infrastructure communicates with vehicles equipped with trusted hardware. These vehicles use CRDTs, such as counters and sets, to store synchronized data that represents a view of the city. Our tests involve simulating multiple cars moving through the city while receiving arbitrary data from the infrastructure related to their surroundings. In our conclusion, we discuss the test results and highlight the limitation of a specific feature of the trusted hardware.

Keywords: CRDT, Security, Intel SGX, TEE, Distributed Systems, V2X.

Contents

Acknowledgements i

Resumo iii

Abstract v

Contents viii

List of Tables ix

List of Figures xi

Listings xiii

Acronyms xv

1 Introduction 1

2 Background 5

2.1 Security 5

2.1.1 CIA Triad 5

2.1.2 Security Mechanisms 6

2.1.3 Cryptography 7

2.1.4 Transport Layer Security (TLS) 10

2.2 CAP Theorem 10

2.3 Conflict-Free Replicated Data Types 12

2.3.1 Synchronisation Models . . . 13
2.3.2 Examples . . . 16
2.3.3 CRDT Security . . . 18
2.4 Trusted Execution Environments . . . 22
2.4.1 ARM TrustZone . . . 23
2.4.2 Intel Software Guard Extension (Intel SGX) . . . 24

3 Design of Hardware-enabled CRDTs 29
3.1 Use Case . . . 29
3.2 Adversary and System Model . . . 30
3.2.1 Standard Functionality . . . 32
3.2.2 Communication Protocol Design . . . 34

4 Implementation and Experimental Discussion 37
4.1 Implementation . . . 37
4.1.1 Gramine . . . 37
4.1.2 Communication Protocols Construction and Used Structures . . . 39
4.2 Experimental Setup . . . 42
4.2.1 Correctness Test . . . 43
4.2.2 Performance Test . . . 44
4.3 Discussion . . . 45

5 Conclusion 47
5.1 Future Work . . . 48

Bibliography 49

List of Tables

4.1 Communications Common Fields 40

4.2 RA+TLS Communication Overhead 44

4.3 Protocol Time Execution Distribution 45

List of Figures

2.1 CIA Triad 6

2.2 Symmetric Encryption and Decryption 8

2.3 CAP Theorem Venn diagram 11

2.4 State-Based Synchronisation 14

2.5 Operation-Based Synchronisation 15

2.6 Secure CRDT Protocol High-Level View 19

2.7 MPC CRDTs System View 21

2.8 MPC CRDTs Merge and Propagate 22

2.9 ARM TrustZone Division Concept 23

2.10 Intel SGX Overview 24

2.11 Intel SGX Attestation Flow 25

3.1 System Overview 31

3.2 Use Case Illustration 33

3.3 Proposed Communication Protocol 35

4.1 Integration of Intel SGX and Intel SGX with Gramine 38

4.2 Gramine EPID Remote Attestation 39

4.3 RA+TLS+Protocol Overhead 44

4.4 Remote Attestation Network Analysis 46

Listings

4.1 Infrastructure Data Structures 41
4.2 vehicle Data Structures 41

Acronyms

AES Advanced Encryption Standard

CA Certificate Authority

CRDT Conflict-Free Replicated Data Type

EC Eventual Consistency

Gcounter Grow-only Counter

MAC Message Authentication Codes

MPC Multi Party Computation

PKI Public-Key Infrastructure

PN-Counter Positive-Negative Counter

SEC Strong Eventual Consistency

SHA Secure Hash Algorithm

TEE Trusted Execution Environment

TLS Transport Layer Security

Chapter 1

Introduction

As time goes by, the reality of having smart cities with extensive sensor networks collecting valuable data for enhancing urban management, including optimizing traffic flow, becomes an increasingly desirable reality [26]. Vehicle communication [2, 21, 58] is typically categorized into three types: Vehicle-to-Infrastructure (V2I), Vehicle-to-Vehicle (V2V), and Vehicle-to-Everything (V2X). V2I involves vehicles communicating with infrastructure elements like traffic lights and road signs, while V2V focuses on direct communication between vehicles themselves. The category we are particularly interested in is V2X, which encompasses both V2I and V2V.

The V2X setting allows us to explore a hierarchical permission's scenario: vehicles communicate in real-time with each other, which may expose them to potential attacks, and with city infrastructure, which can act as a trust anchor. Additionally, enabling autonomous vehicles to receive and locally store real-time data from their surroundings would provide them with an understanding of the current city environment, which in turn would support faster route optimization, enhance traffic management, improve safety, and enable efficient communication for coordinated traffic flow.

This scenario resembles that of a replicated distributed database, where geographically separated servers hold copies of shared data to provide better service for users. For example, high availability ensures that data is accessible from multiple locations with reduced latency, improving data access speed by allowing users to retrieve data from the nearest geographical replica. However, maintaining strong consistency in such systems requires establishing a global total order of updates, ensuring that all users or components view the data in the same order. This pursuit of consistency can impact availability, performance, and scalability, as replicas spend considerable time synchronizing. For example, imagine a distributed database system used for coordinating online reservations for a popular concert, where multiple users are trying to book tickets simultaneously from different locations. To maintain strong consistency, the system must ensure that all users see the same sequence of updates regarding seat availability.

In our scenario, vehicles maintain a real-time version of the city's environment, which is constantly updated by infrastructure and other vehicles. For instance, updates may include

changes such as road closures, accidents, or traffic congestion. These messages can be sent by the city infrastructure and then disseminated among vehicles. Due to the constant influx of such updates, vehicles must prioritize high availability, as strong consistency methods are not suitable for such a dynamic and rapidly changing environment.

Eventual consistency is an alternative to strong consistency in distributed systems. It acknowledges that instead of guaranteeing immediate consistency across all replicas, data may temporarily diverge and then converge to a consistent state over time. Eventual consistent systems allow updates to occur locally in the replicas without the need to synchronize immediately. Instead, they ensure that as updates propagate throughout the system and eventually reach all replicas, they will converge to the same state. However, there is still a challenge. Since updates to the data do not follow a global order, conflicts from concurrent updates can occur, and it can cause data inconsistencies which then must be resolved. For example, Amazon's highly available key-value store system, Dynamo, provides an "*always-on*" experience, but at the cost of inconsistencies under certain failure scenarios [20].

Hence, we turn to the use of Conflict-Free Replicated Data Types (CRDTs) to address the aforementioned challenge. CRDTs [46, 51] are data structures designed for being replicated. They offer strong eventual consistency guarantees, ensuring that replicas who have received the same set of updates will have the same state. CRDTs have special mathematical properties that allow synchronisation without causing conflicts and guarantee that all replicas converge to the same state when all updates are delivered to all replicas. CRDTs are already deployed in some systems, for example, collaborative text editors [45] and distributed NoSQL databases [11, 12].

In the domain of intelligent driving and traffic management, addressing security concerns is crucial due to the role these systems play in controlling the car they are installed in. As these systems depend more on real-time data from multiple sources, they are increasingly exposed to a range of security threats. For instance, if a car's control system relies on data from an online routing application, and this application is fed with malicious information, it could enable malicious actors to manipulate the vehicle. By introducing false congestion reports, attackers could steer the car onto a specific route of their choosing. Unfortunately, these situations are already happening. Google Maps and Waze can be deceived by (e.g. Sybil attacks) fake congestions [49, 53], thus influencing benign user routing. Given that we are considering a scenario where cars store various types of data to reflect the city environment and assist onboard self-driving algorithms in decision-making, ensuring the security of this data, both in storage and during propagation, is crucial.

Indeed, extending CRDT to ensure privacy and authenticity is not a trivial task, as security is still a developing field of study [7, 44]. In most of the literature on CRDTs, the goal is to explore different constructions that can provide different behaviours [3, 51, 52]. Moreover, ensuring data protection in systems using CRDTs presents challenges. These challenges arise from the need to perform computations on encrypted data at the replica's end, which can potentially lead to decrypted data that no longer matches the original information. *Pregiça et al.* [46] suggest the use of techniques like homomorphic encryption or using specific hardware support, such as Intel

SGX and ARM TrustZone.

Barbosa et al. [7] provide the first contribution to this field by presenting a cryptographic treatment of CRDTs. They make use of standard deterministic encryption for sets and registers CRDTs, which works seamlessly in these constructions, and also, homomorphic encryption schemes for counter CRDTs. In their study, they emphasize the need for individualized security adaptations for each CRDT, and their experiments demonstrate a significant performance impact, especially in secure constructs that employ more complex cryptographic techniques (e.g. homomorphic encryption).

To overcome this challenge, *Portela et al.*'s [44] approach, makes use of secure Multi Party Computation (MPC) to seamlessly lift general-purpose implementations of CRDTs into their secure variants without the need for individually tailored solutions. In their work, they take a replica's functionality and assign it to a group of three *MPC* participants. Because different machines need to communicate with each other to handle replica operations, the number of operations per second that can be done will decrease. These works are the closest related formal treatment of secure CRDT we identified during the writing of this dissertation.

To address the issues raised by these works, including the need for individual effort for each secure CRDT construction and the performance impact of techniques like encryption and MPC, we suggest leveraging Trusted Execution Environments (TEEs).

TEEs, enabled by Trusted Hardware, such as Intel SGX, offer a secure and isolated environment within a computer system, ensuring both code execution and data processing remain confidential and untampered. TEEs safeguard against external threats, preventing unauthorized access and changes, while also allowing for efficient code execution.

We will present a practical application for the following use case: (*i*) vehicles accessing real-time information about their surroundings, such as road conditions and nearby vehicles; (*ii*) we want to store the number of cars within a specific location (e.g. streets, in the whole city or parts of it, points of interest), accident reports, status of traffic signals, road closures and general traffic incident reports (e.g. stalled vehicles or debris on the road); (*iii*) make this data accessible to autonomous driving algorithms to optimize routes and manage traffic efficiently.

Our goal is to explore the feasibility of secure CRDT solutions, under the following assumptions:

- Every car can be equipped with a TEE solution;
- TEEs possess the capability of remote attestation, allowing external parties to efficiently verify the correctness and trustworthiness of the code running within them;
- City infrastructure interacts with CRDTs deployed in the vehicles by transmitting states, executing queries, and performing updates as needed.

The subsequent sections of this document are structured in the following manner. In Chapter 2,

we provide the foundational knowledge essential for understanding this thesis. This includes concepts such as, CRDTs, core security principles encompassing confidentiality and integrity, the security mechanisms enabling them, fundamental principles in cryptography and finally we address TEEs. Additionally, in this chapter, we give a detailed overview of the works related with CRDT security that we found in the literature. Chapter 3 introduces the adversary model, presents the design of the proposed system, and provides insights into the practical setting we considered. It also offers an explanation of how the various components of our system communicate and function. We follow by presenting Chapter 4, where we give details about the implementation of the system described in the previous chapter and also provide a comprehensive exploration of our experimental setup, the conducted tests, and their corresponding results. We conclude this chapter with a thorough discussion of the acquired data. In Chapter 5, we provide a summary of the objectives, and the practical application proposed in this work. We reflect on the final achievements, analyse the obtained results, and provide the rationale behind them. The chapter concludes by discussing potential approaches and areas for improvement in future work.

Chapter 2

Background

In this chapter, we will present all the fundamental concepts necessary to understand this work. We will start with fundamental concepts and mechanisms in both systems security, and information security. We follow by giving an overview of the fundamental issue in distributed systems, known as the CAP theorem. Additionally, we will introduce Conflict-Free Replicated Data Types (CRDTs), a very popular object of distributed data structures, what challenges they address, how they function and the existing related work that studies their secure constructions. Finally, we address Trusted Execution Environments (TEEs), presenting current solutions along with their features and limitations.

2.1 Security

In this section, we will present background knowledge about systems and information security. To navigate this field, we rely on the work of *Aumasson* [5] and *Stallings* [54]. In the following sections, we will draw upon the expertise of these authors to better describe security concepts that, we believe, are appropriate to mention for a better understanding of this work.

2.1.1 CIA Triad

In systems and information security there are three main key objectives [29], often referred to as the *CIA triad*, as illustrated in Figure 2.1:

- Confidentiality:
  - Data confidentiality, guarantee that private information is not disclosed to unauthorised parties/individuals;
  - Privacy, guarantee that individuals know what information related to them may be collected, by who and to whom it may be disclosed. Make sure that these individuals can choose what information related to them is collected and stored.

- Integrity:
  - Data Integrity, ensure that data is not modified in any way by anyone or anything that is not authorised to do so;
  - System Integrity, guarantee that a system will perform its intended function, without any deliberate or inadvertent unauthorised manipulation.
- Availability, ensure swift system functionality without denying access to authorised users.



Figure 2.1: Confidentiality, integrity, and availability. Taken from [41].

Although these objectives are well established, additional concepts are commonly proposed in order to address rapidly changing technology and business requirements. We will present two additional concepts that align with the objectives of this work:

- Authenticity, verifying that users are who they claim to be, and that information being communicated to the system came from a trusted source;
- Accountability, detect and prevent intrusions and have the ability to trace a security breach to the responsible entity. This supports non-repudiation, when a message is sent, the sender can verify that it was received by the intended receiver, and the receiver can be assured that the message originated from the expected sender.

These concepts are enabled by the application of specific security mechanisms. In the following section, we will address some of these technologies, considering their relevance for this thesis.

2.1.2 Security Mechanisms

There is an extensive selection of cryptographic mechanisms that enable achieving the aforementioned objectives. We will briefly mention the appropriate mechanisms for each objective and how they help.

Confidentiality requires the protection of data, allowing access to those who are authorised and disallowing those who are not. The main tool for confidentiality is encryption. Encryption involves transforming information using an encryption key, ensuring that the information remains confidential and can only be deciphered using a corresponding secret key, known as the decryption key. This decryption key can either be different (asymmetric) from or identical (symmetric) to the encryption key. Advanced Encryption Standard (AES) is the common adoption currently, it ciphers blocks of 128 bits and uses a secret key of 128, 192 or 256 bits. AES uses the same key for encrypting and decrypting.

Integrity refers to the property that information as not been altered in an unauthorised way. Using hashing tools such as Secure Hash Algorithm (SHA) or Message Authentication Codes (MAC) guarantee data integrity. Hashing functions take a long input and return a short output, referred to as *hash vale* or *hash digest*. Hashing a value S will always return the same fixed size value $|S^{'}|$, thus ensuring that two different values, S and S_2, will not produce the same hash value. Altering a single bit within the message will result in a distinct hash value, so we can readily identify data tampering by comparing the provided hash value with the output of the hash function applied to the corresponding data input.

Availability is primarily achieved through redundancy and duplication of data, systems, and infrastructure.

Authenticity refers to the capability of verifying the legitimacy of statements issued by an entity. The main tools to ensure authenticity are Digital Signatures or Message Authentication Codes. MACs protect a message's authenticity and integrity by encrypting the message along with a secret key, resulting in a unique message tag $T = MAC(K, M)$ where T is the tag, K is the secret key and M is the message. Let us consider the following: if Trudy and Bob share a key K and Bob sends a message M, along with its authentication tag T, to Trudy, Trudy can compute $MAC(K, M)$ and compare the result to T. Because only Bob and Trudy have the secret key, Trudy can not only validate the message integrity, but also verify that it was sent by Bob. Digital signatures work similarly, instead of a secret key, a key pair is used: a private key to decrypt and a public key to encrypt. The private key is kept secret, only known to its owner and the public key is public, everyone can use the public key to encrypt messages and send them to the respective owner, later the owner can use its private key to decrypt the received messages. A common proceeding is to use a hash function like SHA to process the original message and obtain the hash value, if the hash value is encrypted with the sender's private key it serves as signature, providing integrity and authenticity.

2.1.3 Cryptography

In this section, we will describe core cryptography concepts that are of relevance to the problem at hands. We will start by introducing symmetric and asymmetric encryption. Following that, we will discuss the algorithms utilised for generating public-key pairs. Finally, we will conclude this section by exploring Public-key Infrastructures (PKI) and the role of X.509 Certificates.

Symmetric and Asymmetric Encryption

In symmetric encryption the key used to encrypt is the same key used to decrypt as we can see in Figure 2.2.

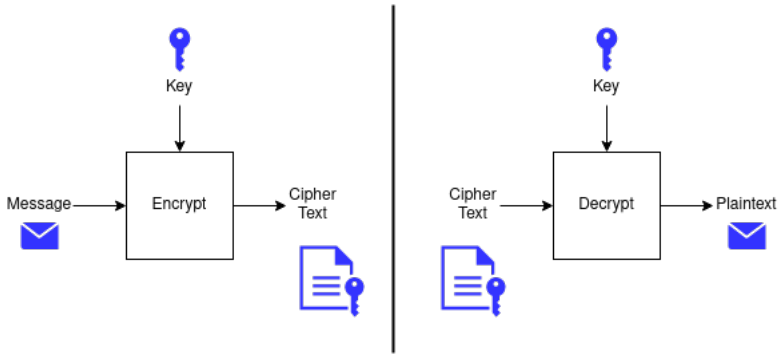


Figure 2.2: Symmetric Encryption and Decryption. Adapted from [5].

The difference between symmetric and asymmetric encryption is that asymmetric encryption uses two different keys, one to encrypt and one to decrypt. The key to encrypt is available to everyone and is referred to as the Public-key, on the other hand, the second key is only available to the key-pair owner and is referred to as the Private-key. The Private-key is used to decrypt data encrypted by the public-key and can also be used to compute the public-key.

Homomorphic Encryption

Homomorphic encryption is a type of encryption scheme that allows operations to be performed on encrypted data without decrypting it first. In other words, it enables computations to be carried out on data while it remains in an encrypted state. This is achieved through a special property of homomorphic encryption that preserves the relationship between the plaintext and ciphertext during operations. For example, an additively homomorphic scheme would allow performing addition operations on encrypted data, and when decrypting, the result would reflect the additions that were made to the data while it was encrypted.

Homomorphic encryption provides robust security measures for safeguarding data privacy. However, it may incur substantial performance costs due to the computational overhead associated with conducting computations on encrypted data.

Public-Key Infrastructure

Let us consider the following scenario, we have two subjects, Bob and Trudy, both have their own private and public keys, if Bob wants to send a secure email to Trudy, Trudy has to know Bob's public key and Bob has to know Trudy's public key (to understand a possible response). Let us say they keep their private keys safe and send the respective public keys to each other,

Bob then uses Trudy's public key to encrypt the message and sends it to her. After, Trudy uses her private key to decrypt the email and read its contents.

This example raises the question, how does Bob know with certainty that the public-key he used to encrypt the email actually belongs to Trudy? There is a possibility that a malicious individual could have intercepted the communication and substituted Trudy's key with a different key.

The concept of Public-Key Infrastructures (PKIs) aims to help address this problem. PKIs are defined as a set of hardware, software, people, policies, and procedures [8] needed to manage digital certificates based on public-key cryptography. The main objective is to enable secure acquisition of public-keys. A trusted party, or Certificate Authority (CA) issues a validated certificate, signed with its private-key, which guarantees that the public-key does indeed belong to the appropriate individual.

With the use of a PKI the previous example would be as follows:

1. Bob and Trudy agree to use the CA to verify their identities;
2. Trudy requests the CA for a certificate;
3. The CA verifies Trudy's identity, computes the hash value of the information that will be in her certificate, and signs it with its private-key, note that the CA public-key is itself in the certificate and Trudy can later validate the signature. The CA finishes by returning the certificate to Trudy;
4. When Bob receives Trudy's certificate, he can verify its authenticity by using the public key of the CA.
5. Bob uses Trudy's public-key to encrypt the email and sends it to her;
6. Trudy uses her private-key to decrypt the email from Bob.

X.509 Certificates

An *X.509 Certificate* is a standardised digital document structure that defines a link between a public-key and an identity. This document is paired with a signature of a trusted party, the previously mentioned CA. The CA signature guarantees that the link between a public-key and identity is indeed valid.

The general format of a *X.509 certificate* and the information it contains is the following:

- Version Number: the standard version;
- Serial Number: a unique integer number assigned by the CA;
- Signature Algorithm ID: the algorithm used to sign the certificate;

- Issuer: name of the CA that issued the certificate;
- Validity Period;
- Subject: identifier of the entity to which the certificate was issued to;
- Subject Public Key: the public key of the subject;
- Extensions: additional optional information and certification revocation lists;
- Signature: the digital signature by the issuing CA for the certificate.

The CA signs the certificates with its private key, if a user receives a certificate signed by a CA then the corresponding subject's public-key is guaranteed to be of the respective subject.

2.1.4 Transport Layer Security (TLS)

Transport Layer Security (TLS) is a robust communication protocol that establishes a secure connection between two entities and also authenticates them. It guarantees confidentiality, authentication, and integrity. TLS combines both symmetric and asymmetric encryption techniques. Initially, it utilises a key pair to securely exchange a session key. This session key is then employed to encrypt and decrypt the data in transit. Once the session concludes, the session key is securely discarded.

TLS can be divided in two main protocols, *record protocol* and the *handshake protocol*. The *record protocol* determines how to transmit data, and the *handshake* determines what data to transmit. The *handshake* will result in the two parties agreeing on which encryption methods will be used, and the exchanging cryptographic keys.

Data exchanged in a TLS communication is transmitted through *TLS records* that serve as data packets. The TLS record protocol acts as a transport protocol, initially, the TLS record protocol carries data exchanged during the handshake phase, but once the handshake is complete, application data is fragmented into chunks and transmitted within *TLS records*.

2.2 CAP Theorem

When designing and deploying a distributed system there are usually three well known core systematic requirements [22]:

- Consistency, the read data is always the same for all clients;
- Availability, the system responds to all requests it receives, even if some replicas are offline;
- Partition Tolerance, the system must be robust to node failures and message losses.

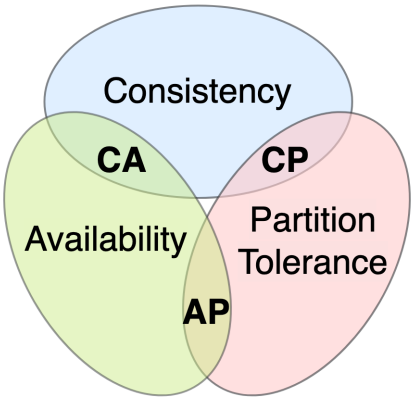


Figure 2.3: CAP Theorem Venn diagram. Taken from [31].

According to Brewer [10], it is impossible to have all the three requirements at the same time.

Usually, high availability is the main focus [20], which requires an effort to reach a balance between availability and consistency. The challenge in balancing consistency and availability in distributed systems revolves around finding the right trade-off. Strong consistency ensures that all replicas have the same data at all times, but this can lead to increased latency and reduced availability due to synchronization. Eventual consistency prioritizes availability and responsiveness but allows temporary data divergence, requiring mechanisms to resolve conflicts in order to converge to a consistent state over time. Achieving the right balance depends on the specific system's requirements and use cases.

Considering our scenario, where the replica's state will be changing constantly, a need for responsiveness is paramount, vehicles should be able to quickly propagate their state and also receive multiple propagates and, updates or queries, from other vehicles and city infrastructures. To use strong consistency in our system is not feasible because replicas must support a high number of requests.

Hence, the reason to opt for an eventual consistency approach such as CRDTs. Additionally, CRDTs provide a Strong Eventual Consistency (SEC) model, which guarantees that replicas who receive the same set of updates will have the same state, assuring that conflicts or inconsistencies will be resolved by design. *Shapiro et al.* [51] show that a replica with SEC guarantees is always available for any reads and writes and that it provides an extreme form of fault tolerance, up to $n - 1$ simultaneous crashes, all of this while not requiring consensus mechanisms to solve occurring conflicts.

2.3 Conflict-Free Replicated Data Types

In distributed systems, data replication is often necessary to enhance data availability, fault tolerance, and load distribution to achieve better user experiences (e.g. low latency and high availability). Some systems, such as those in the banking sector, need to be strongly consistent to ensure that all transactions are processed correctly and that all account balances are accurate. Achieving strong consistency involves maintaining a global total order of updates [38], ensuring that all replicas process updates in the same order and maintain identical states. As discussed in Section 2.2, achieving strong consistency reduces availability and imposes negative, performance and scalability, impacts, while eventual consistency offers better availability, performance, and scalability but may result in conflicts and data inconsistencies in replicated systems.

CRDTs are data structures designed for being replicated, that facilitate consistent and synchronized data sharing in distributed systems. They can handle concurrent updates from multiple sources without requiring centralised coordination or consensus algorithms. In other words, CRDTs are designed to allow multiple users or nodes to independently update a shared piece of data in a distributed system while resolving possible conflicts. CRDTs guarantee that replicas who received the same set of updates reach the same state, this is achieved by design, since CRDTs have well a defined interface and mathematical properties that ensure conflict-free merging of updates, even in the presence of network partitions or communication delays.

In systems where temporary data inconsistencies can happen, CRDTs stand out, offering high availability, performance, and scalability even in the face of faults and network disconnections. Unlike some other eventual consistent systems where inconsistencies can arise in certain failure scenarios [20], CRDTs are designed to ensure conflict resolution, guaranteeing that when replicas receive the same set of updates, they converge to the same state [51].

The common interface for CRDT interaction exposes the following functions:

- *Query()*, to retrieve data from the CRDT;
- *Update()*, to modify or update data in the CRDT;
- *Merge()*, to combine data from different replicas into the local state (resulting in a new synchronized state);
- *Propagate()*, to transmit updates to other replicas in the system.

An easy to reason about CRDT, is a replicated counter, since its increment and decrement operations are commutative, receiving these updates in any particular order will not cause conflicts. However, one must provide, a communication architecture that ensures no duplicate updates will be delivered, since increments and decrements are not idempotent. We will see detailed examples of how this type of CRDTs work in Section 2.3.2 below.

The abstractions of a CRDT were first presented by *Pregiça et al.* in [45] and later formalised

by *Shapiro et al.* in [51, 52]. In the latter a new consistency model, that builds on Eventual Consistency (EC), was proposed, Strong Eventual Consistency.

Informally, EC implies that replicas will synchronize to the same state when all updates cease. We adopt the definition of EC, that captures this intuition, from *Shapiro et al.* [51]:

Definition 2.3.1 (Eventual Consistency)**:**

Eventual delivery: An update delivered at some correct replica is eventually delivered to all correct replicas.

Convergence: Correct replicas that have delivered the same updates eventually reach equivalent state.

Termination: All method executions terminate.

Eventual consistency provides a form of convergence where replicas may apply updates immediately, but conflicts can arise later. To achieve full consistency across replicas, additional mechanisms like rollbacks may be necessary to resolve these conflicts [55]. On the other hand, strong eventual consistency (SEC) ensures that all replicas attain consistency once all updates are distributed, and it handles conflicts locally. SEC has been formally verified by *Gomes et al.* [25].

Definition 2.3.2 (Strong Eventual Consistency [51])**:**

An object is Strongly Eventually Consistent if it is Eventually Consistent and has Strong Convergence;

Strong Convergence: Replicas that have delivered the same updates have equivalent state (updates can be in any order).

CRDTs usually handle updates in two phases: (*i*) the local phase, where the replica receives the operation and performs some process; (*ii*) the *downstream* phase, where the update is asynchronously transmitted to all replicas. CRDTs converge if all updates reach all replicas. Therefore, ensuring that updates reliably reach all replicas is crucial. This aspect is referred to as the synchronization model [46], that we will present below.

2.3.1 Synchronisation Models

CRDTs have mathematical properties that allow synchronisation without causing conflicts and guarantee that all replicas converge to the same state when all updates are delivered to all replicas. In the literature, three synchronization models enable this: **State-based**, **Operation-based**, and **Delta-based**.

State-based

In the state-based synchronisation model, CRDTs synchronise by sending their entire state to a peer replica. CRDTs designed for state-based replication use a merge function to integrate the

received state with their local state.

It is proved in [51] that assuming eventual delivery and termination, replicas of a CRDT are guaranteed to converge if:

i.) The sates of the CRDT form a semilattice [19] ordered by $\leq$.

ii.) Merging two states $s1$, $s2$ produces the *Least Upper Bound* of the two states, i.e., $merge(s1, s2) = s1 \sqcup s2$. The merge function must be:

- Idempotent, applying the function various times does not change the result;
- Commutative, order of merges do not change the result;
- Associative, merge may contain several updates and their order does not affect the result, i.e, $m(u1, u2, u3) = m(u2, u1, u3)$.

iii.) The updates to the replica state result in an inflation, i.e., $s1 \leq update(s1)$

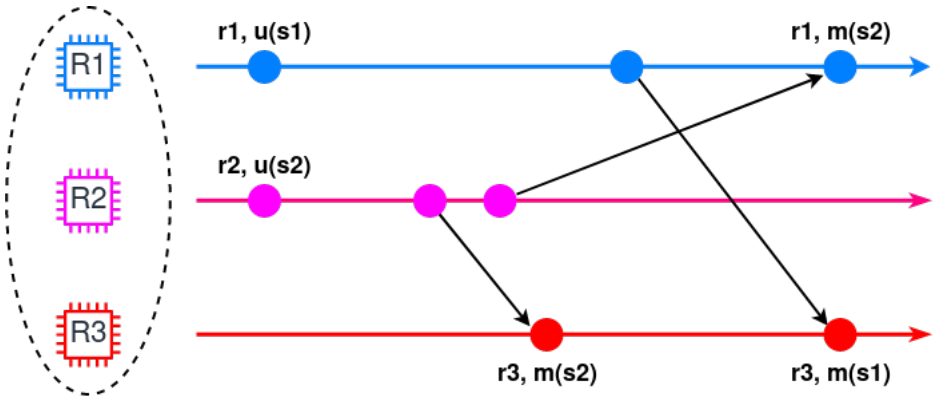


Figure 2.4: State-Based replication. Adapted from [52].

Figure 2.4 is an example of a state-based CRDT, where $\boldsymbol{r_i}$ is the replica identification, $\boldsymbol{u(s_i)}$ is the update to the respective state and $\boldsymbol{m(s_i)}$ represents the merge of the given state.

One of the main disadvantages of state-based CRDTs is the potential for high communication overhead. Since state-based CRDTs share their entire state during synchronisation, even if only a small part of the state has changed, the entire state needs to be communicated. This can lead to increased network traffic and higher latency, especially when dealing with substantial amounts of data.

Operation-based

Unlike state-based CRDTs, operation-based CRDTs synchronise by only sending the respective update operation to all peer replicas. They also dispose of the merge function and instead split the update operation into a *prepare-update* and *effect-update* functions [51].

When an update occurs the prepare-update function is invoked and executes the respective update in the replica where the update is submitted, the effect-update is invoked immediately after and executes the update at all replicas [51].

In order for the operation-based CRDTs to converge some requirements are required:

- Reliable multicast communication subsystem;
- If updates can reach in any order, the operations must be commutative, else if causal order exists, concurrent updates must commute;
- When duplicate or more deliveries of the same update occur, the operation must be idempotent.

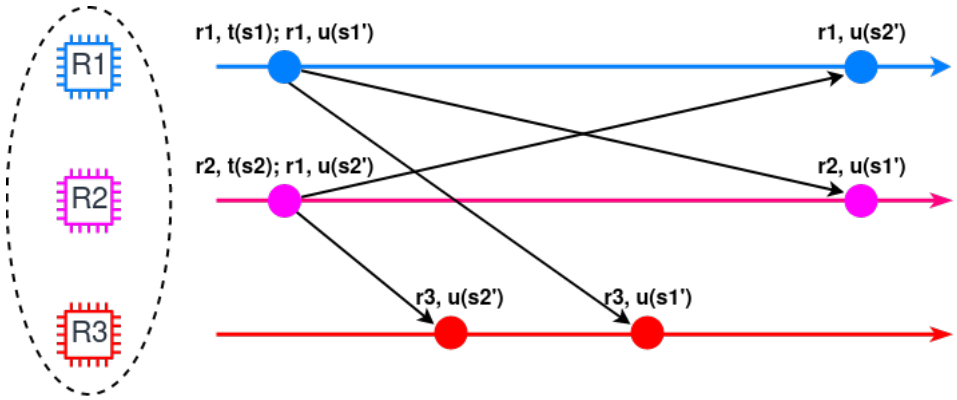


Figure 2.5: Operation-Based replication. Adapted from [51].

Figure 2.4 is an example of an operation-based CRDT, r_i is the replica identification, $t(s_i)$ is the *prepare-update* function and u'_i is the *effect-update* function, s_i represents the data in the respective local replica and s'_i represents the data to send to other replicas.

Operation-based CRDTs bring a communication efficiency improvement since communicating small update messages is less bandwidth consuming than propagating the replicas entire state, which can possibly be a large volume of data. Of course, this comes at the cost of maintaining an appropriate communication subsystem that can provide the needed requirements.

Delta-based

Given the scope of two synchronisation models discussed above, we can observe that propagating the replica entire state when a small update occurs is very inefficient since the replica already knows the majority of the information. Another observation is that, when a numerous amount of updates change the same state, it is more efficient to propagate the entirety of the state instead of propagating all the updates individually.

Delta-based CRDTs [3, 4] tackle these issues by combining both synchronisation models. Instead of propagating single operations or the CRDT entire state, *Delta-mutators* are propagated,

which include the changes that have been made to a replica since the last communication. This way, only the differences in state between the two CRDTs are communicated.

2.3.2 Examples

Shapiro et al. in [52] provide a portfolio of basic CRDT designs with state-based and operation-based specifications. In this section we will present three CRDT constructions: *State-based increment-only Counter (Gcounter)*, *State-based (PN-Counter)* and *Operation-based counter*. Nonetheless, we refer the reader to the literature [52] for a better understanding of all the existing basic CRDTs (counter, registers, sets, graphs, lists) and for more elaborated CRDTs such as JSON documents [35] or ordered lists [42, 45].

State-based Increment-Only Counter (Gcounter)

Specification 1 State-based increment-only counter. Taken from [52].

1: payload integer$[n]$ P ▷ One entry per replica
2: initial $[0, 0, ..., 0]$
3: update *increment()*
4: let $g = myID()$ ▷ g: source replica
5: $P[g] := P[g] + 1$
6: query *value()* : integer v
7: let $v = \sum_{i=0}^{n} P[i]$
8: compare (X, Y) : boolean b
9: let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i])$
10: merge (X, Y) : payload Z
11: let $\forall i \in [0, n-1] : Z.P[i] = max(X.P[i], Y.P[i])$

The Grow-only Counter (Gcounter) CRDT defined in Specification 1 describes the behaviour of an increment-only counter. Each replica is identified by $n \in [0, TotalReplicas - 1]$, and its local state is defined by a vector (with size $TotalReplicas - 1$) of integers, payload P.

The increment update only changes the respective replica r_{id} field in the array with the help of a function *myID()* that returns the local replica's identifier, e.g, for a replica r_{id}, whose ID is given by $id = myID() = 1$, the increment update will result in $r_1.increment() = r_1.P[1] + 1$.

For consulting the counter value, we only need to compute the sum of all the values in the array P. When propagating, the receiving replica utilizes the $compare(X, Y)$ function within its *merge* process to compare each value in its local replica array, denoted as X, with the values in the propagated array Y from the other replica. Finally, the local replica selects the maximum value for each entry in the array.

State-based Positive-Negative Counter (PN-Counter)

Specification 2 State-based positive-negative counter. Taken from [52].

1: payload integer[n] P, integer[n] N ▷ One entry per replica
2: initial $[0, 0, ..., 0]$, initial $[0, 0, ..., 0]$
3: update *increment()*
4: let $g = myID()$ ▷ g: source replica
5: $P[g] := P[g] + 1$
6: update *decrement()*
7: let $g = myID()$ ▷ g: source replica
8: $N[g] := N[g] + 1$
9: query *value()* : integer v
10: let $v = \sum_{i=0}^{n} P[i] - \sum_{i=0}^{n} N[i]$
11: compare (X, Y) : boolean b
12: let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i] \bigwedge \forall i \in [0, n-1] : X.N[i] \leq Y.N[i])$
13: merge (X, Y) : payload Z
14: let $\forall i \in [0, n-1] : Z.P[i] = max(X.P[i], Y.P[i])$
15: let $\forall i \in [0, n-1] : Z.N[i] = max(X.N[i], Y.N[i])$

The Positive-Negative Counter (PN-Counter) CRDT defined in Specification 2 describes the behaviour of a counter that can increment and decrement. It combines two Gcounters, this means the payload consists of two integer arrays, P for increments and N for decrements which also implies the use of two update functions, *increment()* and *decrement()*, respectively. The counter value is the difference between the two arrays $\sum_{i=0}^{n} P[i] - \sum_{i=0}^{n} N[i]$.

Operation-based Counter

The Operation-based counter CRDT defined in Specification 3 describes the behaviour of a counter with both increment and decrement functionalities. Since both addition and subtraction are commutative and associative, updates can be applied in any order which means that the prepare phase is not needed, as well as the pre-condition. The payload of the object is an integer with an initial value of 0. The query operation simply returns the current number value in the CRDT. Additionally, when updating, the replica is responsible to asynchronously propagate, or *downstream()*, the respective update to other replicas, and upon reaching other replicas the update must be correctly executed. In this case, for the correct functionality of the CRDT and to ensure that replicas will correctly receive updates, a reliable multicast communication subsystem is needed. The multicast subsystem must ensure that all updates are delivered to all replicas correctly, and in case of updates not being idempotent, like increments and decrements, the system also needs to guarantee that updates are delivered exactly once.

Specification 3 Operation-based counter. Taken from [52].

```
1:  payload integer i
2:      initial 0
3:  query value() : integer v
4:      let v = i
5:  update increment()
6:      i := i + 1
7:      downstream()                    ▷ No pre-condition: delivery order is empty
8:  update decrement()
9:      i := i − 1
10:     downstream()                    ▷ No pre-condition: delivery order is empty
```

2.3.3 CRDT Security

Because of their high availability and scalability, CRDTs are widely deployed in several distributed databases systems, e.g. *Riak* [11, 12], *Yahoo!'s PNUTS* [15] and *Cassandra* [37]. However, since CRDTs themselves do not provide security guarantees such as, confidentiality and integrity, malicious replicas could simply observe and tamper with sensible stored data, which under real-world circumstances would not be feasible.

It would be desirable if clients could make use of CRDTs characteristics, even if replicas are located in untrusted environments. By ensuring data confidentiality and integrity, a hostile environment with deployed CRDT replicas would be unable to access or tamper with the data within them.

However, having security guarantees like, confidentiality and integrity, would not only protect information against the untrusted environment, but also against replicas themselves, since operations that would require to change state like, updates or merges, would make the protected data unreadable. This is primarily because security mechanisms like encryption transform data into a secret form, and any tampering with this data will produce a different result upon decryption compared to the original data.

Pregiça et al. [46] comment on this particularity and suggest allowing a limited form of computation, like merging state, over encrypted information. Potential techniques like homomorphic encryption or using specific hardware support, such as Intel SGX and ARM TrustZone could enable this.

Although CRDTs exist for some time [45], the field of CRDT security is relatively recent, and the literature is scarce. However, in the subsection below 2.3.3, we will present an overview of the works conducted by *Barbosa et al.* [7], *Portela et al.* [44] and *Jorge* [34], as these are the ones most closely associated with formal treatments of secure CRDTs that we found in the literature.

Secure Conflict-Free Replicated Data Types

Barbosa et al. [7] proposed the first formal approach to secure CRDTs. In their work they present secure constructions for the following CRDT specifications: register, set, counter and bounded-counter. These constructions are carefully designed to allow computations over encrypted data while preserving the CRDT functionality. They start by proposing the following scenario:

- Clients interact with replica nodes by the means of *update* and *query* functions;
- To ensure data consistency, replicas propagate their state and merge states received from other replicas;
- A security layer is added between clients reading and writing data in order to encrypt data before sending to the replica node and decrypt data after receiving it from the replica node;
- *Propagates* and *Merges* work on top of encoded data.

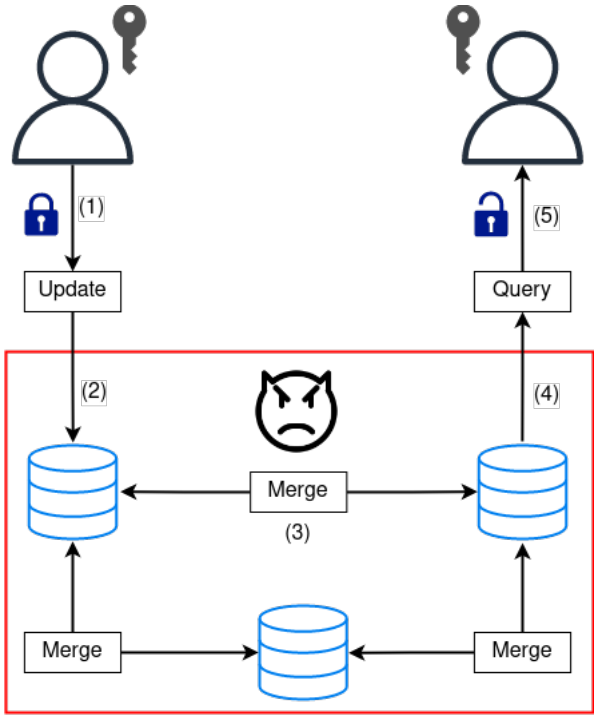


Figure 2.6: High-level view of *Barbosa et al.* proposed system. Adapted from [7].

Figure 2.6 illustrates the proposed scenario. Firstly, we will assume that clients have established encryption keys beforehand in order to be able to encrypt and decrypt data. The *update* function is preceded by an encryption operation (step 1). When receiving the encrypted data, the replica node performs the respective *update* (step 2), followed by possibly multiple *propagate* and *merge* functions (step 3). Step 3 is of particular importance, since it presents the challenge to execute computations over encrypted data, and it is specific to each CRDT construction. Lastly, the *query* function represents a request for the node replica data (step 4) which in return is delivered to the client and must be decrypted (step 5).

In order to make this approach clearer, let us analyse the case of a register CRDT, a data structure that maintains a single value. The *update* function of a register simply replaces the current value with the new one, while the *query* function only returns the value. The *merge* function replaces the register value with the most recent one, only relying on associated metadata (e.g. timestamp to decide what is old and new) to ensure consistency. In the secure register CRDT construction, all the functions work seamlessly with encrypted data because there is no computation over the data. When the client utilises the update function, the new encrypted value straightforwardly replaces the existing encrypted value at the replica. Similarly, when a client queries the replica, it only needs to decrypt the returned value to retrieve the plaintext value. The *merge* function follows the same principles, since it only replaces the old encoded value with the new one.

In the register CRDT construction the cryptographic overhead is minimal, at most, there is one key generation in the initial part of the protocol and then one encryption (in *update* operation) and one decryption (in *query* operation).

The set CRDT construction requires the replica to do comparisons over encrypted data. The set CRDT update function is composed of two operations, one for adding elements to the set and another to remove. In order to make an add or remove, we first need to understand if the element is already in the set, this mandates a comparison to be done between the given element and the elements in the set. Moreover, the *merge* function works by adding to the local replica's set only the unique elements, this again, mandates a comparison to be made. Deterministic encryption can be used as a solution for equality comparisons, since it always produces the same ciphertext for a given plaintext and key, we can assume that comparing encrypted values is as simple as comparing plaintext values.

Like the register, in the set construction the cryptographic overhead is also minimal, assuming that, the only computation over encrypted data is the equality comparison.

Barbosa et al. also propose a counter CRDT construction, where increments and decrements are supported by having two counters per replica, one for increments and one for decrements, similar to the PN-Counter in Specification 2. This counter, however, does two types of computation to the data, arithmetic (increment and decrement) and comparison, as in the merge function in Specification 2. In order to have to deal with only one operation over encrypted data, the proposed secure counter construction includes a per-replica *Lamport clock* [38] in plaintext to establish partial ordering of events. With addition being the only computation over encrypted data present. To achieve addition over encrypted data, *Barbosa et al.* employ an additively homomorphic encryption scheme (as the one described in Section 2.1.3), which results in a higher performance overhead compared to the two previous constructions.

Transforming existing CRDT implementations into secure variants is not straightforward due to the fact that different CRDTs involve distinct computations on their data, and there are limitations on the types of computations that can be performed on encrypted data. Additionally, in the experiment evaluation the authors note a natural trade-off of the secure variant CRDTs,

the performance is proportional to the requirements imposed by the cryptographic scheme [7], meaning that constructions that make use of more complex cryptographic schemes were shown to have a higher latency without any growth in throughput (operations per second). For instance, in *Section 6.1* of *Barbosa et al.*'s [7] paper, specifically in the *Micro Experiments* section, the findings related to the Counter CRDT demonstrate that the secure variant initially operates at 127ops/s with a latency of 57ms. However, it experiences a rapid increase in latency without a corresponding growth in throughput. In contrast, the plaintext variant exhibits a performance increase from 1250 to 3000ops/s with consistently low latency, ranging from 6ms to 33ms.

Secure Multi-Party Computation (SMPC) CRDTs

Portela et al. [44] proposed using Multi Party Computation (MPC) to achieve a secure variant of general-purpose implementations of CRDTs. This represents an alternative approach for achieving secure CRDTs as compared to the previously mentioned work, which relies on standard encryption techniques. *Portela et al.* suggest the use of multiple MPC parties to compute the CRDTs functionality in a privacy-preserving setting. The main idea is to keep the CRDT data private while divided into three distinct parties that can securely compute the desired CRDT function. Figure 2.7 provides an illustration of the proposed system overview. Furthermore, it is emphasised that each of the three parties should be on separate machines. This precaution is essential, as the compromise of a single machine housing all parties could potentially lead to the adversary gaining access to all the data.

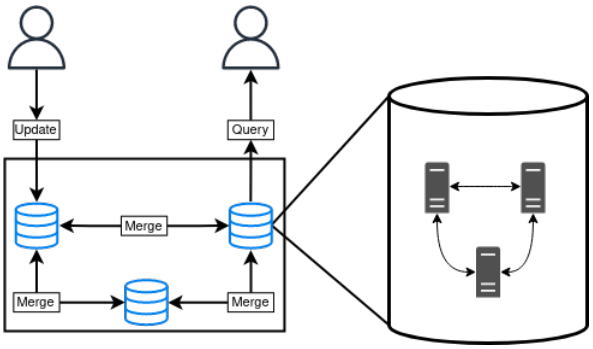


Figure 2.7: System view of MPC CRDTs. Adapted from *Jorge* [34].

The CRDT general operations, like in the previous work, consist of update, query, merge and propagate. Clients arbitrarily choose a replica to connect and perform the update or query operations. In case of an update, clients are responsible to handle the partition of the data and send each partition with the respective metadata (operation identifier, timestamp, or others) to each replica's CRDT party. Additionally, on query operations, the client also as the responsibility to receive the partitioned data from each replica's CRDT party and promptly perform the specific steps needed in order to reconstruct the original state of the data.

Replicas have the responsibility to propagate and merge state with other replicas underlying parties by means of a direct mapping. Figure 2.8 depicts this, for example, party one of replica

A merges or propagates to party one of replica B, party two of replica A merges or propagates to party two of replica C and so on.

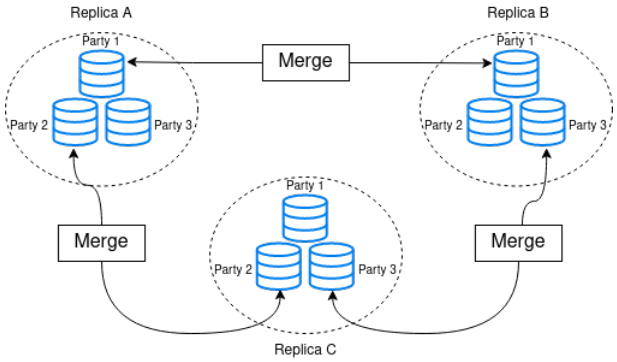


Figure 2.8: Merge and Propagate operations

To conclude, enabling secure CRDT variants presents challenges. In *Barbosa et al.*[7] work we can see that protecting set and register CRDTs can be achieved by using deterministic cryptography which imposes minimum cryptographic overhead, however, with counter CRDTs, which need to use more complex encryption scheme like homomorphic encryption, the cryptographic overhead is significant, and it is noticeable in their experiments. This also emphasizes that there is no one-size-fits-all solution for secure CRDT constructions, and each CRDT requires individual consideration to enable its secure variant. On the other hand, in *Portela et al.*'s [44] work, they manage to achieve secure CRDT constructions by making a replica's functionality be distributed by a group of participants that use Multi Party Computation (MPC). This allows them to do propagates and merges seamlessly, but they have to manage all the interactions between the MPC parties (and parties themselves), which is not easy.

Hence, the need to consider trusted hardware, where replica's code can run inside a protected area and do all necessary operations seamlessly, without sacrificing performance.

The works discussed in this subsection align most closely with the objectives of this thesis. Through our overview of these works, readers now possess a more comprehensive understanding of the challenges at hand, the available solutions to address them and their limitation.

2.4 Trusted Execution Environments

Trusted Execution Environments (TEEs) are a secure and isolated environment within a computer system or microprocessor that allow code execution and data processing with a high level of security and integrity [48]. TEEs are designed to protect sensitive information and ensure that code running within the environment is secure from external threats, including malware and unauthorised access. Any arbitrary piece of information inside this area is not known to the rest of the system. TEEs provide integrity and confidentiality, data inside a TEE cannot be tampered with or accessed.

In this section, we will provide an overview of Trusted Execution Environments, briefly addressing ARM TrustZone and giving special attention to Intel SGX, as it is the technology we have worked with.

2.4.1 ARM TrustZone

ARM TrustZone is a hardware-based security technology designed to provide a secure execution environment on ARM-based microprocessors. The main idea behind TrustZone is to partition the system resources: the Secure partition, named *Secure World*, holds secure memory and containers, and the insecure partition, named *Normal World*, that holds the standard software stack. Figure 2.9 depicts this division and the two separate worlds.

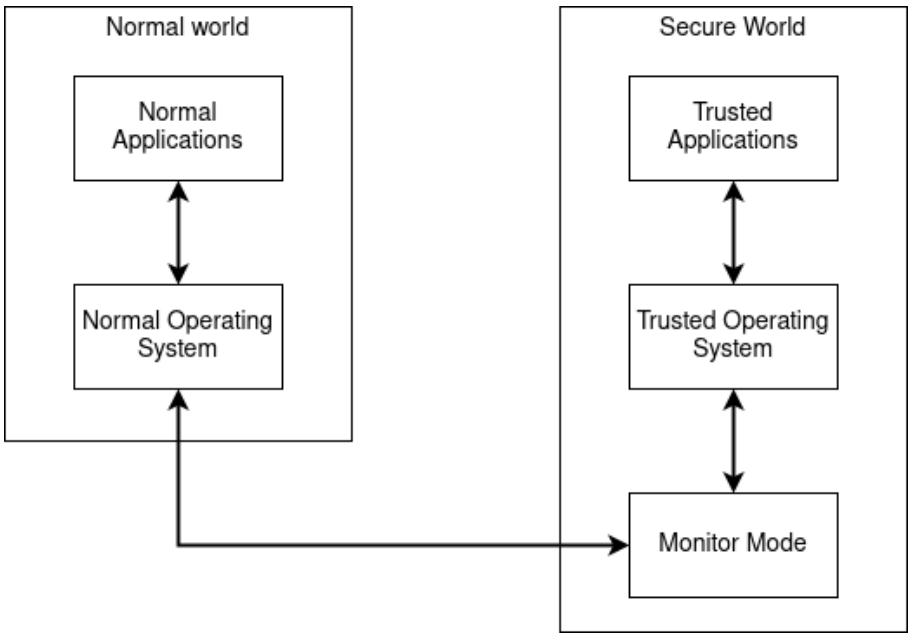


Figure 2.9: ARM TrustZone division concept. Adapted from [43]

In the normal world partition, we typically find standard operating systems like Android or Linux, where everyday tasks are executed. In contrast, the secure world is dedicated to running secure operations, such as encryption tasks and the secure storage of sensitive data, such as private keys. The secure world is totally independent of the normal world, thus guaranteeing that any malicious code or malware in the normal world will not affect any data or operations in the secure world.

Applications running in the normal mode cannot switch between worlds, instead, when some functionality is required from the secure world, they request execution from the secure world through Secure Monitor Calls (SMC). SMC's trigger the Monitor Mode, whose purpose is to manage context switching between worlds in a strict and tightly controlled manner [30, 43].

ARM TrustZone finds extensive use in the smartphone domain. *Brasser et al.*[9] utilize TrustZone to regulate how smart devices, equipped with a wide range of sensors, collect data in

restricted places, such as confidential enterprise environments or federal offices. Other examples are, protecting smartphone sensors data integrity [14, 60] or a client-side game protection tool such as *TZMon* [32] that ensures the confidentiality and integrity of data exchanged between mobile video games and game servers on smartphones. Outside the smartphone domain, for instance, we have *PROTOC* [39], that uses TrustZone to protect drone controls from malicious peripheral control.

As we will see in the next section, unlike Intel SGX, ARM TrustZone does not natively support remote attestation, which is a mechanism that allows a remote party to verify if the secure enclave's code and data are in a trusted and unaltered state before engaging in secure communication or transactions.

2.4.2 Intel Software Guard Extension (Intel SGX)

In this chapter we will refer to the work of *Costan and Devadas* [18] in which they present an extensive detailed view of Intel SGX and fundamental concepts that will help understand the technology. Additionally, Intel's Developer's Manual [16, 17] will serve as a point of reference for a broader understanding, while for more specific functionalities, such as remote attestation, we will turn to the work of *Johnson et al.* [33], *Scarlata et al.* [50] and *Mechalas* [40].

Intel Software Guard Extension (Intel SGX) comprises a set of instructions and mechanisms integrated into Intel architecture processors to enhance memory access security. SGX allows applications, running in the native OS environment, to create isolated containers in memory, referred to as an enclave. An enclave is a trusted area of memory where critical elements of the application, like code and data, benefit from hardware-enhanced confidentiality and integrity protections.

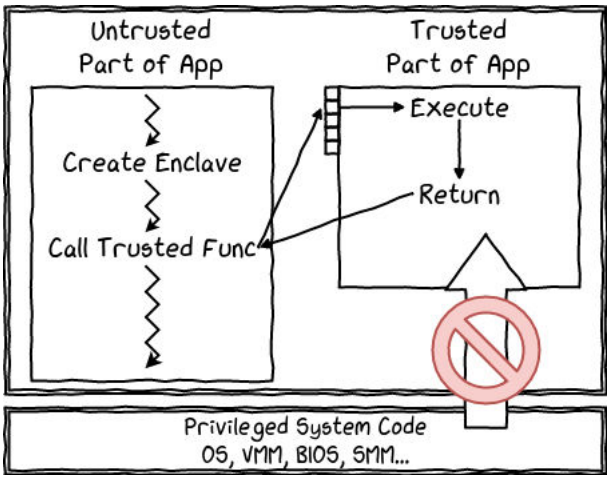


Figure 2.10: Intel SGX Overview. Taken from [1]

Figure 2.10 depicts a summarization of Intel SGX: an application is divided into a trusted and an untrusted part; the application launches the enclave; and finally, the application draws

from a set of pre-defined functions inside the enclave (ECALLs) [16].

Initially, the code and data inside the enclave are loaded by the untrusted application that invokes the enclave creation. This is called the *loading phase*. Once the loading phase is complete, the CPU disables further invocations of the loading method, preventing any further changes to the enclave's state. The enclave creation process is concluded by hashing the enclave's content. The resulting hash value becomes the enclave's *measurement hash* or *MRENCLAVE*. This measurement can then be used to undergo a *software attestation*, which basically tells us if an enclave is running a specific code whose hash value corresponds to a given measurement.

This feature is of particular importance in this work as it enables us to compare the measurements of a potentially untrusted, random car with those of a known trusted car. Based on the outcome of this comparison, we can determine whether to establish communication with the vehicle or not.

Remote Attestation

Intel SGX natively supports remote attestation, which allows a remote entity to verify if a particular piece of code is securely running within an enclave on an Intel SGX enabled platform. If the enclave identity is verified, we can then establish an encrypted communication channel between the two entities.

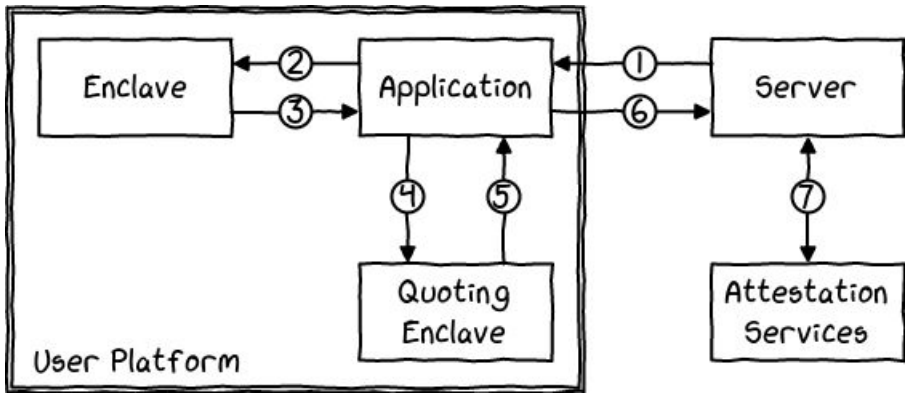


Figure 2.11: Intel SGX Attestation Flow. Taken from [1]

Remote attestation relies on an architectural enclave known as the Quoting Enclave (QE). An architectural enclave, like other enclaves, is a protected and isolated area of memory, but unlike other enclaves the Quoting Enclave is part of the Intel SGX architecture whose primary function is to validate and enhance the *REPORT*, that is locally verifiable, into a QUOTE, which is remotely verifiable.

The *REPORT* is a data structure that contains crucial information about the state and integrity of the enclave, such as measurements of code and data (*MRENCLAVE)*, cryptographic evidence of its authenticity, and details about its configuration.

The *REPORT* serves as a proof of the enclave's trustworthy execution. By signing the *REPORT* with its asymmetric attestation key, known as the *Provisioning Key*, the Quoting Enclave provides a digitally signed *QUOTE* that can be presented to external entities for remote verification.

Currently, Intel SGX supports two types of remote attestation, Enhanced Privacy ID (EPID) [33] attestation and Elliptic Curve Digital Signature Algorithm (ECDSA) [50] attestation.

- Enhanced Privacy ID (EPID) is a form of group signature scheme that enables a platform to sign objects without the need for unique platform identification or the linking of distinct signatures, ensuring user privacy. EPID groups generate signatures using their respective secret keys, and these signatures can only be verified using the public key associated with the group to which they belong. In the case of EPID attestation, the Quoting Enclave utilises EPID to sign enclave attestations. This approach requires internet access since it requires the use of Intel Attestation Services to verify the EPID signature.
- Elliptic Curve Digital Signature Algorithm (ECDSA) enables third-party attestation via the Intel Software Guard Extensions Data Center Attestation Primitives (Intel® SGX DCAP). This attestation method enables end-users to achieve remote attestation without contacting Intel attestation services, instead they are given the tools that allow them to deploy their own attestation service.

Because the hardware we had available did not support the ECDSA attestation flow, we made use of the EPID one. Further details can be found in Section 4.3. Additionally, we refer to Figure 2.11 for the general the attestation flow illustration and will detail how it works if an Enhanced Privacy ID (EPID) flow is used:

1. The application receives an attestation request from a remote entity;
2. The application requests its enclave for an attestation;
3. The enclave generates its *REPORT* and returns the structure to the application;
4. The application returns the *REPORT* from its enclave to the *Quoting Enclave* in order to obtain a *QUOTE* that can be remotely verifiable;
5. In the EPID attestation flow, the *Quoting Enclave* will proceed by doing the following:
   i.) Verifying the *REPORT* and asking for the EPID key to the *Provisioning Enclave*;
   ii.) The *Provisioning Enclave* requests the EPID key, associated with this SGX machine, to the remote *Intel Provisioning Service*;
   iii.) Once the *Quoting Enclave* receives the EPID key, it signs the *REPORT*, resulting in a *QUOTE* that will be delivered back to the application.
6. The application proceeds to send the *QUOTE* to the remote party;

7. Once the remote party receives the application's *QUOTE*, it proceeds to its verification, this can be done by using Intel's Attestation Services (IAS) or a third party that as built their own attestation infrastructure [50]. Once the remote party receives the result of the *QUOTE* validation, it can decide to establish communication or not.

In this work, we leverage Intel's remote attestation feature to ensure that vehicles are running trusted and correct code within their TEE. Subsequently, we can establish a TLS (Section 2.1.4) secure communication channel. This approach ensures the confidentiality and integrity of CRDTs stored within the vehicle's TEE, and additionally, external communication data is also secured through the TLS protocol, guaranteeing in-transit confidentiality and integrity.

All the concepts covered in this chapter serve as the foundational knowledge essential for this thesis. By exploring the topics discussed in this section, the reader is now well-equipped with the necessary understanding and references to delve into the technical aspects explored in the subsequent sections of this work.

Chapter 3

Design of Hardware-enabled CRDTs

In this chapter, we will present in more detail the system's design and practical application. We will start off by presenting the considered use case. Subsequently, we will present an overview of the adversary model we are considering, followed by the system's design, which includes a detailed overview of the system and describes standard functionalities. Additionally, we also give insights as to why we are considering the approach of using Trusted Hardware in this work. We conclude by giving technical details to communications between the two key actors in our system, namely vehicles and the infrastructure.

3.1 Use Case

We will consider an application designed to facilitate intelligent driving in autonomous vehicles. This application relies on the continuous availability of real-time data regarding the vehicle's surroundings and environment, which will be shared by other vehicles and city infrastructure. Its primary goals include to assist route planning, traffic management, and adaptive speed adjustments. In essence, the application aims to help autonomous vehicles to make decisions that enhance the safety and efficiency of the vehicle for both the occupants and other road users, adjusting the vehicle's behaviour accordingly.

To accomplish these goals, the application would require access to a variety of information, including the following examples:

- Traffic signal status;
- Accident reports;
- Road closures;
- Weather conditions (e.g., temperature, precipitation, visibility);
- Road surface quality, including pothole locations;
- Information about construction zones;
- Historical data on traffic flow patterns;
- Available parking spaces in different city areas.

We should note that, if a malicious vehicle would be able to forge or change this type of data and share it with others, it would be able to manipulate other vehicle's behaviour. A real-world example of a similar set is the one of Google Maps and Waze, which can be deceived by (e.g. Sybil attacks) fake congestions [49, 53], this results in benign vehicles route planning being manipulated. Such situations might lead to attackers causing massive traffic congestions willingly. Worse scenarios can be thought of, for example, if a malicious vehicle manipulates traffic signal status (e.g. propagate green light status at all signals) and shares it with other vehicles, this can lead to massive accidents all over a city.

Additionally, we will assume that a city infrastructure will be producing and propagating genuine information to the vehicles. This information will be disseminated by the vehicles to others, which will have the assist of the Trusted Execution Environment (TEE), thus guaranteeing the distinction of possible forged or tampered information.

Three major requirements are needed in order to safely support the considered application: (*i*) replicated data of the city environment within vehicles needs high availability and resistance to disconnects (vehicles and start and stop at will); (*ii*) the information within the vehicle must be protected with confidentiality and integrity guarantees; (*iii*) vehicles must be able to verify if a message's source is trustworthy and reject messages from untrusted sources.

We propose the use of Conflict-Free Replicated Data Types (CRDTs) to achieve (*i*) since they give us high availability and partition tolerance while also providing a strong eventual consistency. Additionally, CRDTs also have particularly good performance, this makes them appropriate to handle a large amount of operations (updates and merges) per second. To achieve (*ii*) and (*iii*) we propose the use of an on-board trusted hardware solution, such as Intel SGX. Intel SGX will be able to allocate the CRDT structures, providing confidentiality and integrity to the information, while hopefully maintaining the good performance of operations per second of the CRDTs. Additionally, by using the Intel SGX remote attestation feature, we will be able to verify if the software/code inside a vehicle is correct, thus, having the capability to decide is the source of a given message is trustworthy.

3.2 Adversary and System Model

As illustrated in Figure 3.1, our system will be composed of three entities: city infrastructure; vehicles; and manufacturer.

The manufacturer plays the role of a Certification Authority (CA) (mentioned in Section 2.1.3), that will authenticate infrastructures. Similarly to a Public-Key Infrastructure (PKI), the code that will run in the vehicles will have information about the CA, that in this case, will be the manufacturer entity. This way, when an infrastructure tries to establish communication with a vehicle, it has to present proof that it is authentic. This allows the vehicle to register infrastructures authenticated by the CA.

The infrastructure will consist of several instances that propagate information from the vehicles. Additionally, the infrastructure will update and merge information in the vehicles that is useful for the vehicle's decision-making algorithm (as described in our use case 3.1 above). The infrastructure will also query information as needed, and it will make sure it is communicating with a trustworthy vehicle. The infrastructure starts communication with a vehicle by requesting attestation evidence, this triggers the behaviour mentioned above, in Section 2.4.2. When completing the attestation process, we can verify that the enclave is running inside a trusted environment and that the code it's running was not tampered with. Additionally, if the infrastructure deems the vehicle trustworthy, secure communication is established using the Transport Layer Security (TLS) protocol. The vehicle can verify the authenticity of the infrastructure by using its locally stored public key, ensuring that the source is legitimate before proceeding with secure communication.

Vehicles will represent our CRDT replicas inside an Intel SGX TEE, which will hold a replicated view of the city. In order for the infrastructure to operate the vehicle's CRDTs, they expose the following interface: Setup(); Query(); Update(); Merge(); and Propagate().

Our objective is to safeguard against threats originating from the network and other vehicles. This entails ensuring that messages cannot be intercepted or altered while in transit, as well as preventing malicious vehicles from tampering with or forging messages. Essentially, we aim to uphold the integrity and authenticity of the propagated messages, verifying that they originate from trusted sources, and that at most, an adversary vehicle can only refuse to propagate states, similar to a Denial-of-Service attack.

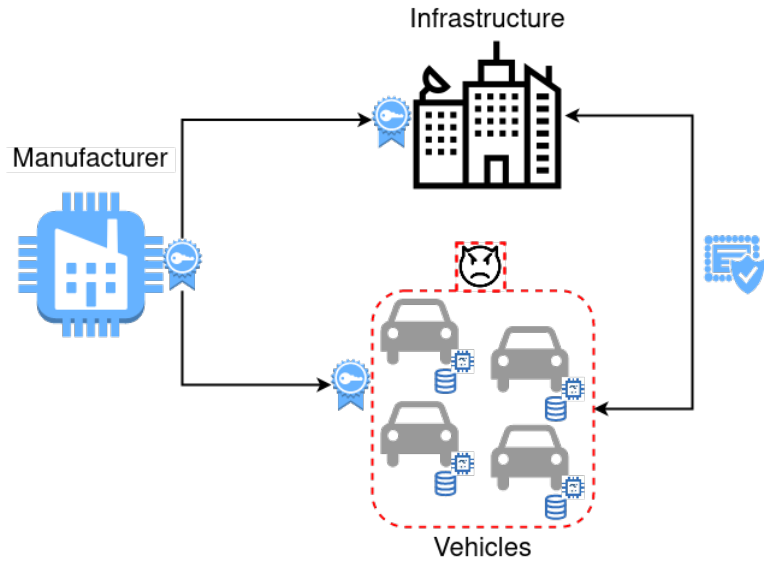


Figure 3.1: High-level system overview

As discussed in Section 2.3.3, previous attempts to achieve secure CRDT variants through software-based solutions have resulted in reduced throughput, leading to CRDT replicas handling fewer operations per second. Furthermore, it is evident that there is no universal solution for

securing CRDT constructions, and each CRDT construction requires a tailored approach to enable its secure variant. Although some techniques like Multi Party Computation can be a solution to achieve seamless secure CRDT constructions, these types of systems are not easy to manage.

Hence, the goal of this thesis. We propose to leverage Trusted Hardware in order to tackle both of these challenges. Assuming that the usage of such hardware technologies can be trusted by the system users, we can use the trusted hardware's TEE to safely execute our applications code while ensuring the confidentiality and integrity of the data it processes. This approach will facilitate the implementation of secure CRDTs without requiring custom constructions. Given the confidentiality and integrity guarantees provided by a TEE for both data and code, we can ensure the security of general CRDT constructions. Additionally, as trusted hardware can perform computations overprotected data seamlessly, the throughput impact on our hardware-based CRDTs should be considerably lower compared to software-based methods.

Since by leveraging trusted hardware, we do not need to design specific secure constructions of CRDTs, we followed the available specifications in [52] for the Grow-only Counter (Gcounter) in Specification 6, Positive-Negative Counter (PN-Counter) in Specification 7 and, the Two-Phase Set (2P-Set) in Specification 13. Additionally, we turned to [6] for the Bounded Counter in Figure 1.

To provide a clearer understanding of our work and demonstrate how we realize the use case outlined in Section 3.1, we will depict what a normal execution of our application would be in the following section. Additionally, we refer to Figure 3.2 which offers a more detailed representation of the environment in which our application operates.

3.2.1 Standard Functionality

We will use CRDTs such as counters to keep track of a number of vehicles in certain areas of the city, and sets to record occurring events such as accidents, closed streets, change in the number of lanes of a street or even social events that can cause a spontaneous increase in traffic circulation.

A new vehicle will start with no CRDT structures or any data relative to its surroundings, even of known trusted infrastructures. The first time an infrastructure interacts with a new vehicle, it starts a *setup phase* which consists of:

- The infrastructure requests attestation evidence and if the vehicle is trustworthy, the infrastructure additionally requests to be registered as trusted;
- In the vehicle, the infrastructure request is only considered valid, if it is signed by the manufacturer's key (which acts as a CA), in order to authenticate the infrastructure. If the infrastructure is authenticated, the vehicle stores a $(\mathsf{Infra}_{\mathsf{ID}}; \mathsf{Infra}_{\mathsf{Pub_key}})$ pair that will allow us to create a secure communication channel, where the infrastructure will propagate, update and query states to vehicles CRDTs.

After the infrastructure successfully registers itself, it proceeds to setup the required CRDT structures. For example, setting up a set would only require the infrastructure to send a $(\mathsf{CrdtType}_{\mathsf{ID}}; \mathsf{CRDT}_{\mathsf{ID}})$ pair, where the $\mathsf{CrdtType}_{\mathsf{ID}}$ represents the identifier of the CRDT type and $\mathsf{CRDT}_{\mathsf{ID}}$ represents the CRDT unique identifier. The vehicle would then handle the request by creating an empty set with the given ID. The ID is needed because vehicles can have CRDTs from multiple infrastructures.

In the case of setting up a counter, the infrastructure would have to send the following $(\mathsf{CrdtType}_{\mathsf{ID}};\ \mathsf{CRDT}_{\mathsf{ID}};\ \mathsf{Replic}_{\mathsf{ID}})$ arguments. This time a new field *Replica ID* must be sent because state-based counters, when updating locally, change only the value in an array whose index is equal to the replica ID, for example, let's consider replica R_1, the update $\mathsf{R}_1.\mathsf{inc}()$ results in $\mathsf{R}_1.\mathsf{P}[1] = \mathsf{R}_1.\mathsf{P}[1] + 1$. The infrastructure is responsible to control whose replica *ID's* it assigns, we achieve this by storing $(\mathsf{CRDT}_{\mathsf{ID}};\ \mathsf{MetaData})$ pairs. Considering the latest example, the infrastructure would have something such as $(\mathsf{Gcounter}_{\mathsf{ID}};\ \mathsf{MetaData})$ where $\mathsf{MetaData.AssignedIDs}[1] = 1$.

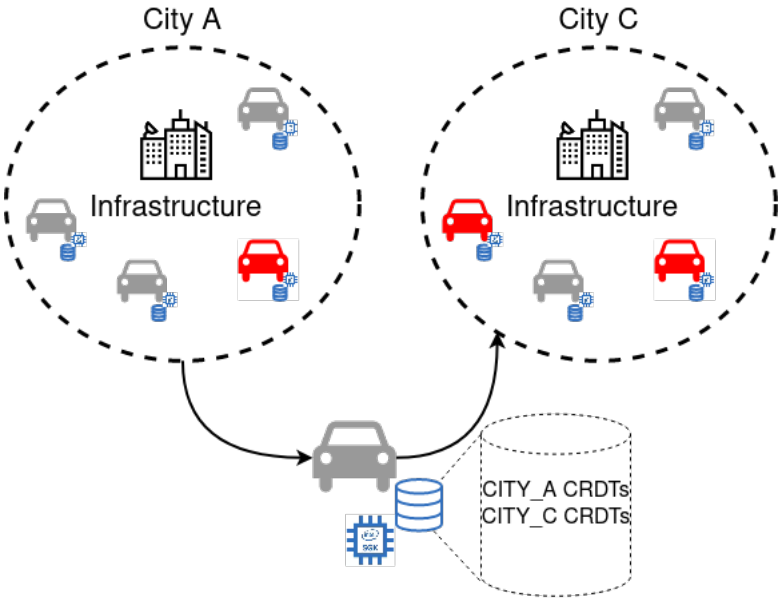


Figure 3.2: Use case environment application

After the infrastructure has registered itself and instantiated the respective CRDTs structures in the vehicle, arbitrary operations (e.g. query, merge, update and propagate) to the respective CRDTs can be made. However, we propose a static communication protocol that does an aggregate of operations. We choose this approach because conducting individual operations, while having to use the EPID attestation flow, was not feasible. Doing this would mean to incur the communication overhead to Intel services, every single operation. This outlines the major setback of using the EPID attestation flow that we will highlight in Chapter 4.

Communication Protocol

Figure 3.3 illustrates the protocol we used for the infrastructure to communicate with the vehicles. It is important to note that, in this initial approach, the communication is facilitated through the infrastructure. Ideally, this protocol could be adapted for other communication settings, such as, vehicle to infrastructure and vehicle to vehicle. We will describe each part and give detail to what functions each step triggers and what they do. The protocol results in the following actions:

(1)(2) The infrastructure starts by requesting a total list of the vehicles CRDTs. The vehicle follows by returning a list of all the CRDTs. The returned list contains pairs like ($\mathsf{CRDT_{ID}}$; Data) where the data is the state of the respective CRDT in bytes.

The infrastructure maps $\mathsf{CRDT_{IDs}}$ to their respective state in a sequence of bytes. This allows the infrastructure to perform a caching role, as we will see below. We will refer to the CRDTs stored in the infrastructure as $\mathsf{LocalCRDT_{ID}}$ and to the map as ($\mathsf{LocalCRDT_{ID}}$; Data).

In this case, for each received pair of ($\mathsf{CRDT_{ID}}$; Data) the infrastructure does the following: if the received $\mathsf{CRDT_{ID}}$ maps to a $\mathsf{LocalCRDT_{ID}}$, it compares their respective data byte sequences. If they are equal, no action is taken. However, if they are different, the infrastructure includes the local pair of ($\mathsf{LocalCRDT_{ID}}$; Data) to be sent in the next message.

(3)(4) The infrastructure sends the built message from the result of the previous step to the vehicle. If any data is sent, the vehicle will then merge the received states locally. This leads to vehicles not only having awareness of the current city's view, but also, the view of other cities, since they will receive information relative to other cities. This happens because vehicles that arrive to, e.g. City C, could come from other cities, e.g. City A. Referring to Figure 3.2, the highlighted vehicle will share the contained City A CRDTs with the infrastructure from City C. City C will then share the City A CRDTs, it received from the highlighted vehicle, with other vehicles within its bounds.

After merging states, the vehicle follows by simply returning a confirmation message if everything goes accordingly.

(5) Upon receiving the vehicle's confirmation, the infrastructure follows by sending an arbitrary amount of updates to the respective CRDTs.

(6) The protocol concludes with the vehicle returning the new state of the updated CRDTs, which the infrastructure will store locally.

3.2.2 Communication Protocol Design

In this section, we will enumerate and provide detailed explanations regarding the construction and functionality of the communication messages used by the infrastructure to interact with vehicles. All communications are started by the infrastructure, and in their body, they share the same four fields:

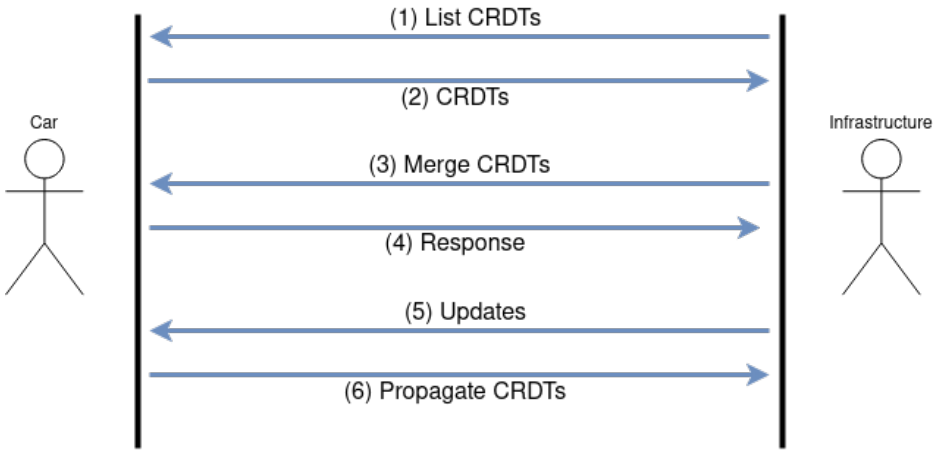


Figure 3.3: Communication protocol design

1. $\mathsf{Operation}_{\mathsf{ID}}$, the operation unique identifier. This is to identify what behaviour to trigger on the vehicle's side;
2. $\mathsf{Infra}_{\mathsf{ID}}$, unique infrastructure identifier;
3. $\mathsf{Infra}_{\mathsf{PubKey}}$, the respective infrastructure Public Key;
4. $\mathsf{Message}_{\mathsf{Sig}}$, as the last field of every communication, we will have a hash value of the message body, which includes all data, signed with the infrastructure's Public Key.

Below we will present communication constructs in the following manner: Function Name, Required Data and Behaviour description.

InfraSetup(): This request will be composed of the fields mentioned in the Section 3.2.2 above, but unlike other requests, the $\mathsf{Message}_{\mathsf{Sig}}$, will be signed with the *Manufacturer's* Key, which acts as a Certificate Authority (CA). This operation results in the vehicle storing the ($\mathsf{Infra}_{\mathsf{ID}}$; $\mathsf{Infra}_{\mathsf{PubKey}}$) pair locally.

SetupCRDT(): Additionally to the fields mentioned in the Section 3.2.2 above, this request will have two more fields:

1. $\mathsf{CRDT}_{\mathsf{ID}}$, a CRDT unique identifier;
2. $\mathsf{CRDT}_{\mathsf{MetaData}}$, additional CRDT metadata (e.g. replica ID).

This request leads to the creation of a CRDT instance within the vehicle's memory, associating it with the received ID and configuring it using the provided metadata. The ($CRDT_{ID}$; $CRDT_{Data}$) pair is stored locally.

Individual CRDT operation requests such as ***Query()***, ***Update()***, ***Merge()*** and ***Propagate()*** will be composed of the common fields described in the Section 3.2.2 above, and an additional $\mathsf{CRDT}_{\mathsf{ID}}$ field. Furthermore, *Update()* and *Merge()* will also contain the following, respectively:

1. *UpdateData*, data respective to the update of the specific CRDT type (e.g. counters will need an arbitrary number to increase or decrease and sets will need an element, this can be a string, to add or remove);

2. $\mathsf{CRDT}_{\mathsf{Data}}$, the complete CRDT state in bytes.

We will now describe the set of operations (illustrated in Figure 3.3) that our proposed protocol implements. Furthermore, it is important to note that in this case, the Remote Attestation and TLS channel only happens in the beginning of the communication. All subsequent operations (except *ListCRDTs()*) happen sequentially after establishing the secure TLS communication channel in order to have less network overhead.

ListCRDTs(): This request uses the fields mentioned in Section 3.2.2, prompting the vehicle to return a list of ($\mathsf{CRDT}_{\mathsf{ID}}$; $\mathsf{CRDT}_{\mathsf{Data}}$) pairs, representing the full state of each CRDT. The infrastructure compares these states with its local versions, and if differences are found, it appends the local ($\mathsf{LocalCRDT}_{\mathsf{ID}}$; $\mathsf{LocalCRDT}_{\mathsf{Data}}$) pairs to the body of the next message, as described below.

MergeCRDTs(): With a secure communication channel established, this request consists of a list of ($\mathsf{LocalCRDT}_{\mathsf{ID}}$; $\mathsf{LocalCRDT}_{\mathsf{Data}}$) pairs. The vehicle parses the request, merges the CRDT states, and responds with a confirmation message, allowing the infrastructure to proceed with the next operation.

UpdateCRDTs(): Like the previous operation, this request will not contain the usual fields, mentioned in the Section 3.2.2 above, instead, the body of the request will contain n individual *update()* operations to the respective CRDTs. This list is arranged by the following fields:

1. $\mathsf{Operation}_{\mathsf{ID}}$, represents the update identifier, e.g. increment, add, transfer (bounded counter), etc...;

2. $\mathsf{CRDT}_{\mathsf{Type}}$, describes the type of CRDT, e.g. counter or set. This is done for sanitization purposes;

3. $\mathsf{CRDT}_{\mathsf{ID}}$, is the CRDT unique identifier;

4. UpdateData, data respective to the update of the specific CRDT type.

In return to this request, the vehicle will again send back a list of ($\mathsf{CRDT}_{\mathsf{ID}}$; $\mathsf{CRDT}_{\mathsf{Data}}$) pairs. For each pair, the infrastructure will update the respective CRDT state in its local memory.

After a detailed exposition of the system's design and the intricate interactions between the infrastructure and vehicles, the ensuing chapter will concentrate on the practical implementation of the system and the respective discussion on the experimental validation results.

Chapter 4

Implementation and Experimental Discussion

In this chapter, we start off by highlighting details of our implementations, including the core tools employed, their significance, and the rationale behind their selection. Additionally, we will give technical details to the construction of our communication protocols and elaborate on the pertinent data structures used and their respective purposes. Subsequently, we will present the results obtained from executing the application described in the previous chapter. We begin by detailing the execution environment in which the tests were conducted. We follow by displaying two test scenarios: a correctness test, where we stress test the application and check for CRDT consistency and, a performance test, where we measure execution times and throughput. Additionally, for both tests, we will provide the gathered measurements, along with analytical observations based on the visualized data. Finally, we conclude by reasoning about the obtained results.

4.1 Implementation

In this section, we will enumerate and provide detailed explanations regarding the construction of communication messages and essential structures in both the infrastructure and the vehicle.

4.1.1 Gramine

Gramine [36, 47], previously referred to as *Graphene* [13, 57], is a *Library OS* designed to enable the execution of unmodified applications within SGX enclaves. Porting applications to SGX is not straightforward due to the introduction of certain constraints that necessitate code adjustments. For instance, system calls within an enclave are prohibited for security reasons. This restriction is in place because the enclave treats the operating system as untrusted, thereby the restriction ensures that system call outcomes can be sanitised by protective code within the enclave before

utilisation. Gramine essentially simplifies the integration of an application into SGX, providing all the necessary tools for a conventional development environment. This approach avoids the limitations imposed when natively using the Intel SGX Software Development Kit, while still preserving the confidentiality and integrity assurances offered by SGX. Figure 4.1 illustrates the differences in the integration of Intel SGX when done conventionally versus when utilising Gramine.

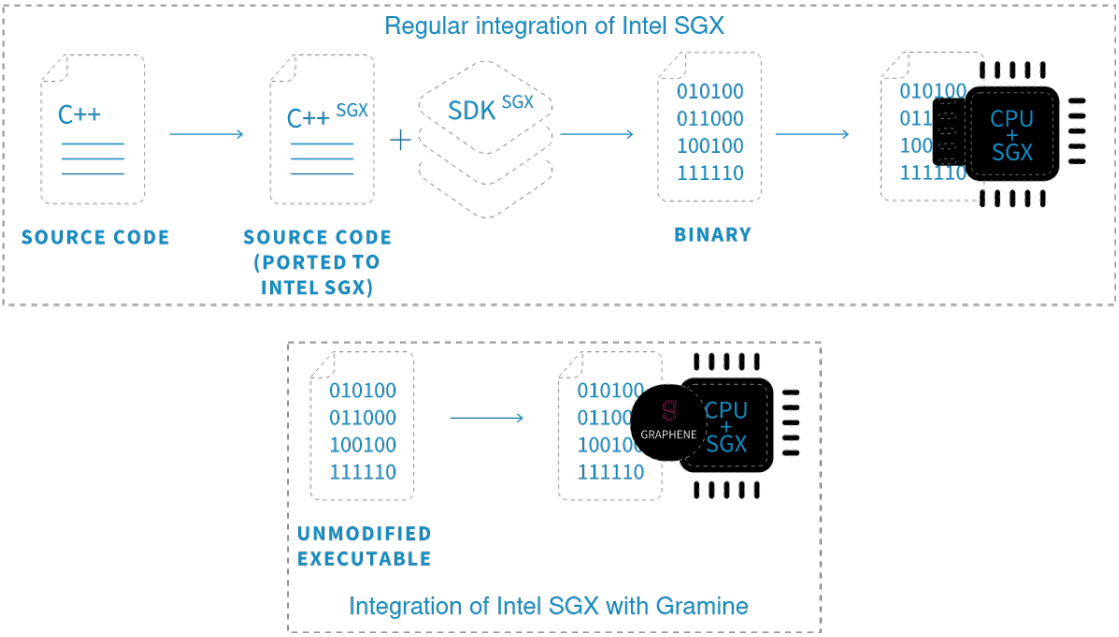


Figure 4.1: Regular integration of Intel SGX and integration of Intel SGX with Gramine. Taken from [47].

The decision to use Gramine was made because it would allow us an easier integration with Intel SGX. Gramine provides a rich documentation [27] on how to use the remote attestation feature, along with an easy-to-read working example that could be easily adapted to our needs [28]. In the following subsection, we will give more specific details regarding how Gramine facilitates the use of the remote attestation functionality.

Remote Attestation

Gramine supports the use of both EPID and DCAP attestation flows provided by Intel SGX. We will be detailing the EPID attestation flow illustrated in Figure 4.2 since, due to hardware constraints, it is the one we made use of (we provide more details in Section 4.3 below).

In the high-level view of the EPID attestation flow illustrated in Figure 4.2 the user application runs inside an SGX enclave on a remote untrusted machine, while the end user runs on a trusted machine. The illustration 4.2 begins with the end user having already requested attestation evidence.

For acquiring a *REPORT*, the enclave opens a special file */dev/attestation/user_report_data*

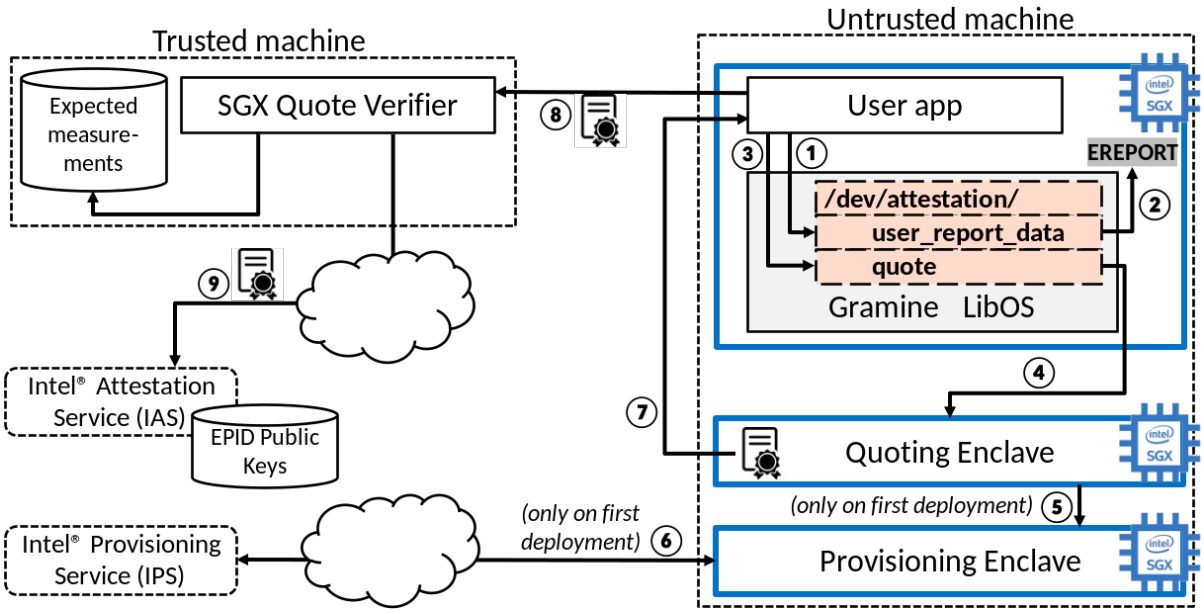


Figure 4.2: Gramine high-level view of the EPID Remote Attestation flow. Taken from [27].

to write the *REPORT* structure **(1)**, after, Gramine calls the *EREPORT* SGX instruction, that in turn returns the *REPORT* structure to the opened file **(2)**.

Similarly, Gramine interacts with the quoting enclave by opening a special file */dev/attestation/quote* for reading a *QUOTE* **(3)** and asks the *Quoting Enclave* for the *QUOTE* structure which will be returned to the opened file for being read and stored in the application's enclave memory **(4)**.

Gramine eases the process of integrating with Intel SGX by taking care of numerous intricate details automatically. This automation not only saves time but also reduces the risk of introducing bugs if managed manually.

Additionally, the remote attestation example, provided by Gramine [28], offers the capability to establish a secure communication channel between the two parties. This is made seamlessly through the integration of the *Mbed TLS* library [56] which not only simplifies various aspects of communication development but also offers valuable cryptographic functionalities, such as, asymmetric (RSA, ECC) encryption algorithms and hashing algorithms like SHA-256.

This example gave us a valuable baseline for our implementation, it achieves Remote Attestation and the creation of a secure channel which we can use to exchange data privately between vehicles and infrastructure.

4.1.2 Communication Protocols Construction and Used Structures

This section's purposed is to give special technical detail about how the communications between infrastructure and vehicle are built, including what data they exchange and how they are

organized. Additionally, we will also detail relevant data structures in vehicles and infrastructures and describe their purpose.

Communications Structure

All communication messages have the following four fields:

Field Name	Data Packet Size
$\mathsf{Operation}_{\mathsf{ID}}$	4 bytes
$\mathsf{Infra}_{\mathsf{ID}}$	16 bytes
$\mathsf{Infra}_{\mathsf{PubKey}}$	$\mathsf{Key}_{\mathsf{Size}}$ bytes
$\mathsf{Message}_{\mathsf{Sig}}$	$\mathsf{Sig}_{\mathsf{Size}}$ bytes

Table 4.1: Communications common fields

Additionally, individual requests are constructed as follows:

- ***InfraSetup()*** and ***ListCRDTs()*** only need the fields above;
- ***Query()***, ***Propagate()*** and, ***SetupCRDT()*** need $\mathsf{CRDT}_{\mathsf{ID}}$ (4 bytes), and ***SetupCRDT()*** needs an additional $\mathsf{CRDT}_{\mathsf{MetaData}}$ (typically 4 bytes);
- ***Update()*** and ***Merge()*** required the $\mathsf{CRDT}_{\mathsf{ID}}$ field and the additional fields below, respectively:
  - $\mathsf{UpdateData}$, if it is a counter the size will be 4 bytes, however, since we store arrays of characters in the sets, this size can vary accordingly;
  - $\mathsf{CRDT}_{\mathsf{Data}}$, the complete CRDT data in bytes.

Requests such as ***MergeCRDTs()*** and ***UpdateCRDTs()*** will not need the common fields presented above, since they are designed to operate inside a TLS secure channel. Their construction is as follows:

- ***MergeCRDTs()***:
  - $\mathsf{TotalAmount}_{\mathsf{CRDTs}} * (\mathsf{CRDT}_{\mathsf{ID}}; \mathsf{CRDT}_{\mathsf{Type}}; \mathsf{CRDT}_{\mathsf{Size}}; \mathsf{CRDT}_{\mathsf{Data}})$.
- ***UpdateCRDTs()***:
  - $\mathsf{Amount}_{\mathsf{Updates}} * (\mathsf{CRDT}_{\mathsf{ID}}; \mathsf{CRDT}_{\mathsf{Type}}; \mathsf{CRDT}_{\mathsf{Size}}; \mathsf{UpdateData})$.

All requests were individually constructed using byte arrays and are handled by both vehicles and infrastructure accordingly. Because of the initial 4 bytes that contain the $\mathsf{Operation}_{\mathsf{ID}}$, we always know how to properly parse received messages. Additionally, all fields that have dynamic sizes, such as $\mathsf{CRDT}_{\mathsf{Data}}$ and $\mathsf{UpdateData}$, are always accompanied by a previous field that holds their size in bytes.

Data Structures

Listing 4.1 contains the data structures used in the infrastructure code. Specifically, we use two HashMap structures, $\text{assigned}_{\text{crdts}}$ and $\text{cached}_{\text{CRDTs}}$, which store Conflict-Free Replicated Data Types (CRDTs) specific to the respective infrastructure and CRDTs from any infrastructure, respectively. In each case, we map a CRDT_{ID} to the respective $\text{CRDT}_{\text{Data}}$. In turn the $\text{CRDT}_{\text{Data}}$ contains: $\text{crdt}_{\text{type}}$ e.g. counter, set, boundedcounter, etc. . . ; $\text{assigned}_{\text{ids}}$ is used to keep track of the replica ID's that the infrastructure assigns when setting up counters, which for now at most will have a maximum size of N_REPLICAS, $\text{crdt}_{\text{byte_size}}$ is the CRDT state size and finally the $\text{crdt}_{\text{byte_data}}$ represent the CRDT state data in bytes. In both $\text{assigned}_{\text{crdts}}$ and $\text{cached}_{\text{CRDTs}}$ we make use of the CRDTData structure as needed.

Listing 4.1: Infrastructure Data Structures

```
typedef struct{
   int crdt_type;
   int assigned_ids[N_REPLICAS];
   size_t crdt_byte_size;
   unsigned char *crdt_byte_data;
} CRDTData;

static GHashTable *assigned_crdts;
static GHashTable *cached_CRDTs;
```

Listing 4.2 contains the data structures used in the vehicle code. All of them are HashMaps. In particular, *infraHashMap* maps $\text{Infrastructure}_{\text{ID}}$ to an $\text{Infrastructure}_{\text{PubKey}}$. On the other hand, the remaining HashMaps, map CRDT_{ID} to the respective $\text{CRDT}_{\text{Data}}$, respectively. We kept different CRDTs in different instances only for convenience, since in some interactions we send the CRDT type, we can then simply go to the respective HashMap and proceed accordingly.

Listing 4.2: vehicle Data Structures

```
static GHashTable *infraHashMap;
static GHashTable *gcounter_crdts;
static GHashTable *pncounter_crdts;
static GHashTable *boundedcounter_crdts;
static GHashTable *twopset_crdts;
```

To save us the time of implementing the HashMap data structures and their functionalities, we made use of the *GLib* library [23, 24]. GLib is a general-purpose utility library in the C. GLib provides data structure handling for C, portability wrappers, and interfaces for such runtime functionality as an event loop, threads, dynamic loading, and an object system.

4.2 Experimental Setup

The experiments that we will display below were conducted on a **local environment** using a single machine with the following hardware specifications: one Intel® Core™ i7-7700HQ processor with 4 total cores, 8 total threads, a base frequency of 2.80GHz, a max frequency of 3.80GHz; and 16 GB of DDR4 RAM memory.

In our experiment, keeping the proposed use case from Section 3.1 in mind, we aim to evaluate the availability and performance of the CRDTs. To achieve this, we will initiate all the functions exposed by their interface, query, update, propagate, and merge, as fast as possible. This experimental approach enables us to assess the feasibility of these structures in a dynamic scenario where real-time information about the vehicles and city surroundings is continuously shared, leading to frequent updates in the vehicle's state.

Each vehicle will be propagating states from all the CRDTs it has instantiated. This includes counter's and set's CRDT states. Furthermore, sets will be holding real location names from Porto city, while counters will be holding arbitrary quantities of other vehicles in separate locations. At the same time, vehicles will also be receiving these types of data, but from other vehicles and multiple infrastructures.

To be able to test this scenario, we created a python script that generates, artificially, all the required input data, which enables the infrastructure to operate over existing vehicles. Particularly, the script starts by generating the needed setup operations and follows by generating artificial inputs that trigger our proposed protocol (described in Section 3.2.1). Additionally, the generated CRDT updates have the following behaviour:

- The Grow-only Counter (Gcounter) update will be a random number to increment;
- The Positive-Negative Counter (PN-Counter) update will have an equal chance of incrementing or decrementing a random number;
- The Bounded-Counter update will have an equal chance of incrementing or decrementing a random number, and additionally will always try to do a transfer operation of a different random number.
- For the Two-Phase Set (2P-Set), the script has a 90% probability of either adding or removing an element. To simplify the process, we utilized the set to store strings representing street names from Porto city. The script accomplishes this by randomly selecting a street name from a predefined list. This approach aims to simulate random events that may occur sporadically within the city.

The communication between the Python script and the infrastructure is made by creating two data streams between them. We use the following command:

```
1    $ socat -d -d pty,raw,echo=0 pty,raw,echo=0
```

The command results in creating two virtual serial terminals (PTYs) and connects them together, allowing data to be transferred bidirectionally between them. We follow by initializing the infrastructure program by using, for example, the following:

```
$ ./runClient.sh < /dev/pts/5
```

The infrastructure program will then be ready for receiving inputs. We follow by starting the Python script with, for example:

```
$ python3 prototype_test.py > /dev/pts/6
```

In this case, writes to */dev/pts/6* by the Python script will be read in */dev/pts/5* by the infrastructure.

The times were collected from the infrastructure view using the `clock_gettime(CLOCK_MONOTONIC,)` function, and every vehicle portrayed in the tests below represents a CRDT replica inside the SGX trusted environment.

4.2.1 Correctness Test

The purpose of this test is to empirically validate the functionality of the Gcounter, PN-Counter, Bounded-Counter, and 2P-Set CRDTs. In order to accomplish this, we deployed three infrastructures which are running the python script and had them interact at the same time with eight different vehicles. The choice of eight vehicles was an attempt to simulate, as closest as we could, a realistic urban scenario with a manageable number of entities that would still allow us to observe meaningful interactions without overwhelming the system.

We ran the test multiple times for periods between 15 minutes and 1 hour. These time intervals were selected to capture the system's performance under both short and sustained conditions. The 15-minute tests allowed us to observe the system's immediate behaviour, while the 1-hour tests helped assess its stability and consistency over longer periods.

At the conclusion of each test, we carried out manual queries on all CRDTs across each vehicle, recording their outputs in separate files corresponding to each vehicle. To ensure their consistency, we performed file comparisons using the *diff* command in the Linux environment. Furthermore, we validated the counter CRDTs values against those generated by the Python scripts and cross-referenced the added elements in the set CRDTs with the elements triggered for addition in the scripts.

During the file equality comparison using the *diff* command, certain differences were identified in the ordering of elements within the sets. However, since the order of elements is irrelevant for Two-Phase Sets (2P-Sets) and only the elements themselves matter, we deemed these differences acceptable. Additionally, we confirmed that the sets contained the same data by exporting the data relevant to the sets, ordering it, and using the *diff* command again. Therefore, the consistency and correctness of the CRDTs was verified.

4.2.2 Performance Test

In this test, our objective was to collect data on connection overhead, execution times, and how many operations the infrastructure can do in a specific amount of time. To accomplish this, we deployed two vehicles and one infrastructure, executing our python script, with the objective to have the application run at full capacity by having it interacting with vehicles non-stop. In this case, we have one infrastructure interacting with two vehicles while the Python script, mentioned in Section 4.2, generates operations in a forever loop. In each iteration, the infrastructure will execute our defined protocol, mentioned in Section 3.2.1, and when it reaches the phase to update the CRDTs states, we are going to do 5 updates, one in each CRDT of the targeted vehicle. We anticipated that the primary limiting factor in serving a vehicle would be the overhead from remote validation via Intel's services, so we opted to update each available CRDT once per protocol execution.

Additionally, we measured the execution times for both the Remote Attestation (RA) with Transport Layer Security (TLS) and our protocol separately. Since RA and TLS occur before our protocol, we recorded the start time at the beginning of the process and again at the start of our protocol. The difference between these two timestamps provided the total execution time for RA+TLS. After completing our protocol and closing the connection with the vehicle, we recorded the final timestamp, giving us the total execution time for our protocol.

The test ran for a duration of two hours.

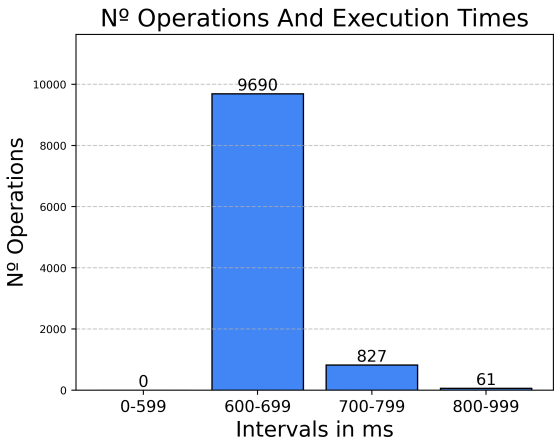


Figure 4.3: RA+TLS Execution Time Distribution

Min	Max	Median	1st Quartile	3rd Quartile
619ms	936ms	663ms	655ms	674ms

Table 4.2: RA+TLS Execution Time Distribution

Table 4.2 illustrates the distribution of communication overhead times, which encompass the Remote Attestation process, the establishment of the TLS secure channel. Additionally, Figure 4.3 presents the frequency of communication operations within 100ms intervals. It is

noteworthy that out of the 10,527 communications analysed, none were completed in less than 600ms. The majority fell within the range of 600ms to 699ms for completion. The throughput *Ops/s* (operations per second), was of approximately 1.46 Ops/s. In 2 hours, the application managed to do 10527 complete interactions (RA+TLS+Protocol).

Min	Max	Median	1st Quartile	3rd Quartile
2.89ms	6.76ms	3.65ms	3.45ms	3.95ms

Table 4.3: Protocol Time Execution Distribution

Finally, in Table 4.3, we present the distribution of individual execution times for the proposed communication protocol.

4.3 Discussion

In all the results, it becomes evident that our performance is significantly constrained by the communication overhead incurred during interactions with the Intel Attestation Services (IASs). Unfortunately, because of the material constrains, we could only make use of the EPID attestation flow, which is the one that requires remote assistance from Intel services. Utilizing DCAP attestation could significantly reduce communication overhead, but it is important to note that this method relies on functionalities that are exclusively accessible in Intel processors from the 8th Generation and beyond.

In contrast to communication times, our protocol demonstrated exceptionally fast computation times. This strongly indicates that executing CRDTs in trusted hardware-based environments has minimal performance impact compared to other software-based solutions. These results strongly indicate that the CRDT aspect performs exceptionally well, and any enhancements to the communication overhead would significantly improve the application's throughput.

We provide additional investigation on the communication overhead by examining the time required exclusively to achieve remote attestation via the EPID scheme. In order to analyse the RA overhead, we made use of *Wireshark* [59], a network protocol analyser tool that allows users to capture, inspect, and analyse network traffic in real-time.

We proceeded to launch our application's infrastructure, and vehicle processes. Following that, we manually executed a standard operation, which is emulated in our test scripts. This operation involved creating the communication channel (RA+TLS) and running our protocol, which includes a set of arbitrary CRDT update operations.

Figure 4.4 displays the results we obtained from executing the process detailed above. We have highlighted the pertinent information and will now provide the respective explanation:

- On the left, we have the all the messages exchanged during the remote attestation process. Message n°41 is the start of the process and message n°83 is the end. The difference

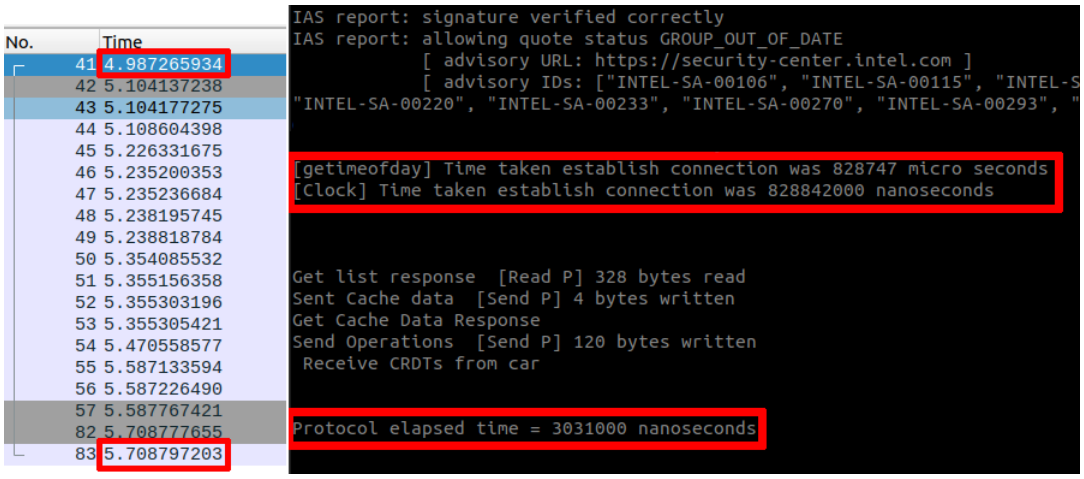


Figure 4.4: Network analysis of the remote attestation procedure and infrastructure's output

between these times represents the individual total overhead of the remote attestation procedure, which in this case was $722ms$;

- On the right, we have our infrastructure's application. The highlighted values above, are the total communication overhead (RA+TLS) (in nanoseconds) viewed from the infrastructure's perspective, while the value below is the execution time of our protocol (in nanoseconds). The values are $828ms$ and $3ms$ respectively.

We consider the following, the total communication overhead measured by the infrastructure, and denoted as T_{Comm}, consists of: (i) the sum of the individual remote attestation procedure's time T_{RA}; (ii) the time spent on all other operations (including the TLS channel) that might occur (during this procedure), noted as $T_{TLS+Ops}$. This comes out as $T_{Comm} = T_{RA} + T_{TLS+Ops}$.

As shown in Figure 4.4 we can conclude that: $T_{Comm} = 828ms$, $T_{RA} = 722ms$. We can then extrapolate that $T_{TLS+Ops} = 106ms$. If we consider the total time of computation to be T_{Comm} plus the time of our protocol execution, $T_{Proto} = 3ms$, then this means, that for all the computation, 86.9% of the time was spent on the Remote Attestation procedure.

$$\text{RA time usage} = \frac{T_{RA} * 100}{T_{Comm} + T_{Proto}}$$

These results suggest that a more lightweight communication protocol, as an alternative to TLS, could significantly enhance performance by reducing the overhead associated with it. Additionally, if the communication mechanism could be optimized to minimize or eliminate the need for the RA handshake, the overall performance would benefit from reduced latency.

Chapter 5

Conclusion

Assuming that the usage of such hardware technologies can be trusted by the system users, we could leverage the use of Trusted Execution Environments (TEEs), such as Intel SGX, to elevate known CRDT constructions to secure variants, where confidentiality and integrity are natively guaranteed, while computations over the confidential data, can be done seamlessly. Additionally, we propose an application that facilitates a practical scenario enabling Vehicle-to-Everything (V2X) communication. In this context, the infrastructure can interact with CRDT replicas hosted within secure hardware on autonomous vehicles. This data interaction has the potential to enhance city traffic management, addressing the traffic challenges prevalent in densely populated urban areas [21].

In the resulting work, we managed to develop a base-line application that can emulate the scenario we proposed. Furthermore, our application supports the general constructions of Grow-only Counter (Gcounter), Positive-Negative Counter (PN-Counter), 2-Phase Set and Bounded Counter Conflict-Free Replicated Data Types (CRDTs), along with all their respective operations. These elements seamlessly benefit from enhanced integrity and confidentiality, ensured by the TEE.

Our experimental results reveal that the EPID-based remote attestation feature had a significant impact on our solution's performance due to the substantial communication overhead incurred during external internet validations. While more efficient remote attestation features like DCAP are available, our hardware did not support it. This attestation process is crucial to our approach, as it verifies whether a vehicle's program is running within a trusted Intel SGX platform. In contrast, we noticed that the latency for various computations (excluding EPID remote attestation) consistently remained low, suggesting significant room for improvement. Furthermore, the rapid execution times for a range of operations on CRDTs indicate that implementing CRDTs in trusted hardware-based environments has minimal performance impact compared to alternative software-based approaches.

5.1 Future Work

As of the time of drafting this dissertation, we believe this to be the first work utilizing a TEE to enable secure CRDTs, and applying it to the same practical scenario we described above, in Section 5. We consider that the developed application is still in initial stages and that several improvements can be done, either to improve performance or to elaborate the practical scenario even further. For example, the major performance constrains, outlined in Section 4.3, could be severely improve. While we may not have accounted for all time-related factors in Section 4.3, the shown results strongly suggest that any improvement on the Attestation procedure would significantly enhance our application's throughput. In a real-world setting, this would allow the infrastructure to accommodate more vehicles per second, thus maintaining the feasibility of our proposal. Improvements on the remote attestation procedure overhead could be:

- Utilizing the DCAP attestation flow, which does not require interaction with external internet services like the Intel Attestation Service (IAS);
- Storing the vehicle's verified QUOTE structure along with a time-to-live (TTL) stamp for future communication while the TTL is valid. This approach would substantially decrease the frequency of IAS consultations.

Considering the V2X scenario, the current implementation could be adapted to support additional communication flows, such as vehicle-to-infrastructure and vehicle-to-vehicle. Furthermore, we could envision a system where vehicles store logs containing detailed records of operations received from other vehicles. Each operation in the record would be associated with the corresponding QUOTE structure from the vehicle that generated the operation, following something such as: $(\mathsf{Vehicle_n}; \mathsf{QUOTE})$. Infrastructures would then have the capability to detect the presence of operations originating from untrusted vehicles within a set of logged operations. This detection could be achieved by having the infrastructure initiate the remote attestation process for each QUOTE associated with a specific operation, using the QUOTE to verify whether a vehicle is executing a particular piece of code within a trusted environment.

Furthermore, expanding the implementation to include other CRDT constructions could enhance the system's capability to store a broader range of city-related data. For instance, Registers or Maps could be employed to manage public transportation schedules, environmental metrics like air quality and noise levels, as well as weather conditions, among other data types.

We believe that the resulting application could serve as a starting point that facilitates the proposed ideas mentioned above.

Bibliography

[1] Alexandre Adamski. Overview of Intel SGX - Part 1, SGX Internals, 2018.

[2] Fadi Al-Turjman and Joel Poncha Lemayian. Intelligence, security, and vehicular sensor networks in internet of things (IoT)-enabled smart-cities: An overview. *Computers & Electrical Engineering*, 87:106776, October 2020. ISSN: 0045-7906. doi:10.1016/j.compeleceng.2020.106776.

[3] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient State-based CRDTs by Delta-Mutation, March 2015. arXiv:1410.2803 [cs].

[4] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173, January 2018. ISSN: 07437315. doi:10.1016/j.jpdc.2017.08.003.

[5] Jean-Philippe Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press, USA, October 2017. ISBN: 978-1-59327-826-7.

[6] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 31–36, 2015. doi:10.1109/SRDS.2015.32.

[7] Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela, and Nuno Preguiça. Secure Conflict-free Replicated Data Types. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking*, ICDCN '21, pages 6–15, New York, NY, USA, January 2021. Association for Computing Machinery. ISBN: 978-1-4503-8933-4. doi:10.1145/3427796.3427831.

[8] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Request for Comments RFC 5280, Internet Engineering Task Force, May 2008. Num Pages: 151. doi:10.17487/RFC5280.

[9] Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. Regulating ARM TrustZone Devices in Restricted Spaces. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications,*

and Services, MobiSys '16, pages 413–425, New York, NY, USA, June 2016. Association for Computing Machinery. ISBN: 978-1-4503-4269-8. doi:10.1145/2906388.2906390.

[10] Eric Brewer. Towards robust distributed systems. In *Brewer's Theorem*, page 7, July 2000. doi:10.1145/343477.343502.

[11] Russell Brown. Bluffer's guide to CDRTs, 2013.

[12] Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. Riak DT map: A composable, convergent replicated dictionary. *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, April 2014. ISSN: 978-1-4503-2716-9. doi:10.1145/2596631.2596633.

[13] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association. ISBN: 978-1-931971-38-6.

[14] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting Data on Smartphones and Tablets from Memory Attacks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 177–189, New York, NY, USA, March 2015. Association for Computing Machinery. ISBN: 978-1-4503-2835-7. doi:10.1145/2694344.2694380.

[15] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, August 2008. ISSN: 2150-8097. doi:10.14778/1454159.1454167.

[16] Intel Corporation. Intel® software guard extensions (intel® sgx) developer reference, August 2023.

[17] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer Manuals, June 2023.

[18] Victor Costan and Srinivas Devadas. Intel SGX Explained, 2016. Publication info: Preprint.

[19] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, April 2002. ISBN: 978-0-521-78451-1. Google-Books-ID: vVVTxeuiyvQC.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating System Principles*, 2007.

[21] Soufiene Djahel, Nafaâ Jabeur, Robert Barrett, and John Murphy. Toward V2I Communication Technology-based Solution for Reducing Road Traffic Congestion in Smart Cities. In *2015 International Symposium on Networks, Computers and Communications (ISNCC)*, May 2015. doi:10.1109/ISNCC.2015.7238584.

[22] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, June 2002. ISSN: 0163-5700. doi:10.1145/564585.564601.

[23] GNOME Foundation. GLib-2.0, 2023.

[24] GNOME Foundation. GNOME / GLib · GitLab, September 2023.

[25] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, October 2017. ISSN: 2475-1421. doi:10.1145/3133933.

[26] Jose Sanchez Gracias, Gregory S. Parnell, Eric Specking, Edward A. Pohl, and Randy Buchanan. Smart Cities–A Structured Literature Review. *Smart Cities*, 6(4):1719–1743, August 2023. ISSN: 2624-6511. Number: 4 Publisher: Multidisciplinary Digital Publishing Institute. doi:10.3390/smartcities6040080.

[27] Gramine Project. Attestation and Secret Provisioning — Gramine documentation, 2023.

[28] Gramine Project. gramine/CI-Examples/ra-tls-mbedtls at master · gramineproject/gramine, 2023.

[29] Barbara Guttman and E. Roback. An Introduction to Computer Security: the NIST Handbook. *NIST*, October 1995. Last Modified: 2018-11-10T10:11-05:00 Publisher: Barbara Guttman, E Roback.

[30] ARM Holdings. ARM Security Technology Building a Secure System using TrustZone Technology, 2009.

[31] JamieMcCarthy. File:CAP Theorem Venn Diagram.png - Wikipedia, December 2022.

[32] Sanghoon Jeon and Huy Kang Kim. TZMon: Improving mobile game security with ARM trustzone. *Computers & Security*, 109:102391, October 2021. ISSN: 0167-4048. doi:10.1016/j.cose.2021.102391.

[33] Simon Johnson, V. Scarlata, Carlos V. Rozas, Ernie Brickell, Francis X. McKeen, and Ernie Brickell. Intel® Software Guard Extensions: EPID Provisioning and Attestation Services, 2016.

[34] Pedro Miguel de Jesus Jorge. Security in Data Aggregation for Eventually Consistent Systems. November 2022. Accepted: 2023-01-30T01:15:44Z.

[35] Martin Kleppmann and Alastair R. Beresford. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, October 2017. ISSN: 1558-2183. Conference Name: IEEE Transactions on Parallel and Distributed Systems. doi:10.1109/TPDS.2017.2697382.

[36] Dmitrii Kuvaiskii, Gaurav Kumar, and Mona Vij. Computation offloading to hardware accelerators in Intel SGX and Gramine Library OS, March 2022. arXiv:2203.01813 [cs]. doi:10.48550/arXiv.2203.01813.

[37] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010. ISSN: 0163-5980. doi:10.1145/1773912.1773922.

[38] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. ISSN: 0001-0782. doi:10.1145/359545.359563.

[39] Renju Liu and Mani Srivastava. PROTC: PROTeCting Drone's Peripherals through ARM TrustZone. In *Proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications*, DroNet '17, pages 1–6, New York, NY, USA, June 2017. Association for Computing Machinery. ISBN: 978-1-4503-4960-4. doi:10.1145/3086439.3086443.

[40] John Mechalas. ECDSA Attestation with Intel Software Guard Extensions Data Center..., 2021.

[41] Jussi Nikander, Onni Manninen, and Mikko Laajalahti. Requirements for cybersecurity in agricultural communication networks. *Computers and Electronics in Agriculture*, 179: 105776, December 2020. doi:10.1016/j.compag.2020.105776.

[42] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 259–268, New York, NY, USA, November 2006. Association for Computing Machinery. ISBN: 978-1-59593-249-5. doi:10.1145/1180875.1180916.

[43] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys*, 51(6):1–36, November 2019. ISSN: 0360-0300, 1557-7341. doi:10.1145/3291047.

[44] Bernardo Portela, Hugo Pacheco, Pedro Jorge, and Rogério Pontes. General-Purpose Secure Conflict-free Replicated Data Types, 2023. Publication info: Preprint.

[45] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A Commutative Replicated Data Type for Cooperative Editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403, Montreal, Quebec, Canada, June 2009. IEEE. doi:10.1109/ICDCS.2009.20.

[46] Nuno Preguiça. Conflict-free Replicated Data Types: An Overview, June 2018. arXiv:1806.10254 [cs].

[47] Gramine Project. Gramine – a library os for unmodified applications, 2023.

[48] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted Execution Environment: What It is, and What It is Not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, pages 57–64, Helsinki, Finland, August 2015. IEEE. ISBN: 978-1-4673-7952-6. doi:10.1109/Trustcom.2015.357.

[49] Luis Sanchez, Erika Rosas, and Nicolas Hidalgo. Crowdsourcing Under Attack: Detecting Malicious Behaviors in Waze. In Nurit Gal-Oz and Peter R. Lewis, editors, *Trust Management XII*, volume 528, pages 91–106. Springer International Publishing, Cham, 2018. ISBN: 978-3-319-95275-8 978-3-319-95276-5. Series Title: IFIP Advances in Information and Communication Technology. doi:10.1007/978-3-319-95276-5_7.

[50] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting Third Party Attestation for Intel® SGX with Intel® Data Center Attestation Primitives. *Intel*, 2018.

[51] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. Research Report RR-7687, INRIA, July 2011.

[52] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. report, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011. Pages: 50.

[53] Meital Ben Sinai, Nimrod Partush, Shir Yadid, and Eran Yahav. Exploiting Social Navigation, October 2014. arXiv:1410.0151 [cs]. doi:10.48550/arXiv.1410.0151.

[54] William Stallings. *Network Security Essentials: Applications and Standards*. Pearson, Boston, 4th edition edition, March 2010. ISBN: 978-0-13-610805-4.

[55] Douglas Terry, Marvin Theimer, Karin Petersen, Alan Demers, Mike Spreitzer, and Carl Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. *SOSP*, 29, September 2000. doi:10.1145/224056.224070.

[56] Trusted Firmware Org. Mbed TLS documentation hub — Mbed TLS documentation, 2023.

[57] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, Amsterdam The Netherlands, April 2014. ACM. ISBN: 978-1-4503-2704-6. doi:10.1145/2592798.2592812.

[58] Jiadai Wang, Jiajia Liu, and Nei Kato. Networking and Communications in Autonomous Driving: A Survey. *IEEE Communications Surveys & Tutorials*, 21(2):1243–1274, 2019. ISSN: 1553-877X. Conference Name: IEEE Communications Surveys & Tutorials. doi:10.1109/COMST.2018.2888904.

[59] Wireshark. Wireshark · Go Deep, 2023.

[60] Yusnani Mohd Yussoff and Habibah Hashim. Trusted Wireless Sensor Node Platform. *Proceedings of the World Congress on Engineering 2010 Vol I WCE 2010*, 2010.