TÉCNICO
LISBOA

Trusted Vote -

Secure E-voting on a Trusted Mobile Device

Diogo Leandro Palma Monteiro

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Paulo Jorge Pires Ferreira

Examination Committee

Chairperson: Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. Paulo Jorge Pires Ferreira
Member of the Committee: Prof. Nuno Miguel Carvalho dos Santos

October 2017

Acknowledgments

I would like to thank professor Paulo Ferreira for the support and opportunities given to me in the last year. His attention to detail and guidance had a significant contribution during the elaboration of this dissertation.

A special acknowledgement goes to my great friends Alexandr Ignatiev and Pedro Reganha, who have supported me throughout my journey in Instituto Superior Técnico. What we have achieved as a group, the interesting conversations about our projects still motivate me and make me believe that perfect teams can exist.

I would like to thank my friends André Faustino, Inês Percheiro, Rui Santos, Luís Freixinho and Daniel Trindade for allowing me to have great stories to remember later. This five year adventure would not be the same without them.

I would like to express my gratitude to my girlfriend Anisa Shahidian for being always present, specially in the decisive times of the last year. She was my ally and one of my strengths throughout the work on this thesis.

Finally, I give my entire respect and gratitude to my family for the constant and unconditional support that allows me to pursuit my goals.

Abstract

The wide availability of mobile devices, such as smartphones, enabled mobility in our lifestyles. However, traditional voting systems require physical presence of the voter at a specific place and time, which is incompatible with the concept of mobility. The goal of this dissertation is to propose an Internet voting system, called *TrustedVote*, that allows voters to cast their vote anywhere with high level of security. Not only Internet voting solves the mobility issue, but it also significantly raises the turnout rate, reduces administrative costs and tallying time. In order to tackle malware and other insecurities in the client mobile platform, the solution is based on smartphones with a Trusted Execution Environment (TEE). *TrustedVote* leverages the isolation properties of TEEs available in Android and iOS smartphones to perform the cryptographic steps of an Internet voting system, such as vote encryption and voter authentication.

Keywords

Electronic Voting; Internet Voting; Security; Cryptography; Trusted Computing;

Resumo

A presença de dispositivos móveis, como os *smartphones*, introduziu o conceito de mobilidade no nosso quotidiano. No entanto, os sistemas de voto tradicionais obrigam a que o eleitor esteja presente numa data específica para poder votar, o que é incompatível com o conceito de mobilidade. O objectivo desta dissertação é propor um sistema de votação pela *Internet*, chamado *TrustedVote* que permita aos eleitores votar em qualquer lugar com elevadas garantias de segurança. Os sistemas de voto pela Internet não só resolvem o problema da mobilidade, como também reduzem os custos administrativos, a abstenção e tempo de contagem de votos. Para combater *malware* e outras inseguranças nos dispositivos utilizados pelos eleitores, a solução é baseada em *smartphones* com acesso a um TEE. O sistema *TrustedVote* tira partido da propriedade de isolamento oferecida pelos *TEE* disponíveis nos *smartphones Android* e *iOS* para executar os passos criptográficos do protocolo de voto pela *Internet*.

Palavras Chave

Votação Electrónica; Votação pela Internet; Segurança; Criptografia, Computação confiável;

Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.2 Goal and Requirements . . . 2

1.3 Research History . . . 4

1.4 Solution and Contributions . . . 4

1.5 Document Structure . . . 5

2 Related Work **6**

2.1 Introduction . . . 7

2.2 Background . . . 7

2.2.1 ElGamal cryptosystem . . . 7

2.2.1.A Exponential ElGamal cryptosystem . . . 8

2.2.2 Mix-nets . . . 8

2.2.2.A Decryption Mix-net . . . 9

2.2.2.B Reencryption Mix-net . . . 9

2.2.3 Homomorphic encryption . . . 10

2.2.4 Blind signatures . . . 10

2.3 Prêt À Voter . . . 11

2.4 Helios . . . 12

2.5 REVS . . . 13

2.6 Java Card E-voting . . . 15

2.7 Du-Vote . . . 17

2.8 EVIV . . . 19

2.9 Biometric-Based Voting Protocol . . . 21

2.10 Country E-voting Implementations . . . 22

2.10.1 Estonia . . . 22

2.10.2 Switzerland . . . 23

2.10.3 Washington, D.C . . . 23

2.10.4 Norway 24
2.10.5 Australia 24
2.11 Trusted Execution Environments 25
2.11.1 Intel SGX 25
2.11.2 ARM TrustZone 26
2.11.2.A Software Architecture Overview 27
2.11.2.B Hardware Architecture Overview 28
2.11.2.C Trusted Kernels and Services 29
2.12 Summary 30

3 Architecture 32
3.1 Introduction 33
3.2 Voting process overview 33
3.3 Threat Model 34
3.4 Smartphone components 35
3.5 TrustedVote client 36
3.6 Normal World Components 38
3.7 Secure World Components 39
3.7.1 Secure Operating System 41
3.8 Bootstrap of Secure World Components 43
3.9 Threat Analysis 43
3.10 Summary 44

4 Implementation 45
4.1 Introduction 46
4.2 Overview 46
4.3 Normal World components implementation 49
4.3.1 TrustedVote app 49
4.3.2 TrustedVote system call 49
4.4 Secure World components implementation 51
4.4.1 TZ VMM 51
4.4.2 MarkPledge 3 52
4.4.3 Sign 52
4.4.4 Code Card Translator 53
4.5 Summary 53

5 Evaluation **54**

5.1 Introduction 55

5.2 Methodology 55

5.3 Performance of secure world Operations 55

5.3.1 Secure calls performance 56

5.3.2 World-switch overhead 57

5.4 Trusted Computing Base size 58

5.5 Attack Surface assessment 60

5.6 Summary 60

6 Conclusion **62**

6.1 Conclusions 63

6.2 Future Work 63

A MarkPledge 3 and Network Protocol **69**

A.1 MarkPledge 3 69

A.2 Entities 70

A.3 Protocol 71

A.3.1 Election Setup 72

A.3.2 Voting phase 73

A.3.3 Tally and verification 75

List of Figures

2.1 A decryption mix-net in the context of e-voting with 3 trustees. 8
2.2 A reencryption mix-net in the context of e-voting with 3 trustees. 9
2.3 The Prêt À Voter ballot (taken from Ryan *et. al* [1]) 11
2.4 REVS protocol overview (adapted from Joaquim *et. al* [2]) 14
2.5 Java Card 3 protocol overview (adapted from Mohammadpourfard *et. al* [3]) 16
2.6 Du-Vote architecture overview (taken from Grewal *et. al* [4]) 17
2.7 Du-Vote voter's code card (adapted from Grewal *et. al* [4]) 18
2.8 Overview of the EVIV protocol (taken from Joaquim *et. al* [5]). 20
2.9 The memory layout of an enclave in the context of the application's virtual space. 26
2.10 TrustZone software architecture with a dedicated operating system [6]. 27
2.11 Hardware bus architecture overview. 28

3.1 Overview of the *TrustedVote* protocol phases. The first phase is at the left upper corner, proceeding clockwise. 34
3.2 *TrustedVote* client components 35
3.3 *TrustedVote* client application architecture. 37
3.4 Secure boot process 42

4.1 Implementation overview of the *TrustedVote* client 47
4.2 Workflow of the *TrustedVote* app 48
4.3 Execution flow of *TrustedVote smcall* system call. 50

5.1 World-switch overhead in the Prepare ballot secure call. 57

List of Tables

2.1 Overview of e-voting protocols. 30

5.1 *TrustedVote* client execution time required to cast a valid vote in an election with 10 candidates. 56

5.2 *TrustedVote* service secure calls execution times comparison between secure and normal worlds, in an election with 10 candidates. 57

5.3 Code size comparison of *TrustedVote* and other systems. 58

5.4 Code size of the components that execute in the secure world in our prototype. 59

A.1 An example of an empty ballot. The position of the $YESvote$ is randomly selected. 72

A.2 *TrustedVote* code card. 73

A.3 The final vote produced with the empty ballot of Table A.1 and where Donald is the selected candidate. 74

Acronyms

AC Authentication Center

BC Ballot Center

BIOS Basic Input/Output System

BOEE Washington, D.C Board of Elections and Ethics

CPU Central Processing Unit

CSU Central Security Unit

DMA Direct Access Memory

DMC Dynamic Memory Controller

DRAM Dynamic Random Access Memory

ISA Instruction Set Architecture

JSON Javascript Object Notation

M4IF Multi Master Multi-Memory Interface

PRM Processor Reserved Memory

RAM Random Access Memory

ROM Read Only Memory

SCR Secure Configuration Register

SGX Security Guard Extensions

SMC Secure Monitor Call

SSL Secure Sockets Layer

TCB Trusted Computing Base

TEE Trusted Execution Environment

TLR Trusted Language Runtime

TLS Transport Layer Security

TZ VMM TrustZone Virtual Machine Monitor

TZASC TrustZone Address Space Controller

UART Universal Asynchronous Receiver/Transmitter

VMM Virtual Machine Monitor

VST Voter Security Token

1 Introduction

Contents

1.1 Motivation 2
1.2 Goal and Requirements 2
1.3 Research History 4
1.4 Solution and Contributions 4
1.5 Document Structure 5

1.1 Motivation

Traditional, paper-based, voting systems require the voter to cast his vote during a certain period (the election period) and at a specific place, which is inconsistent with the concept of mobility widely seen today (e.g., with smartphones, tablets, smartwatches, etc.). This lack of mobility compromises the goal of democracy, as citizens may not be available to vote (on the designated places) during the election period.

The most recent example of this problem is related to the portuguese local elections that use the traditional voting system. The Portuguese League of Professional Football scheduled four football matches to the election day, October 1, 2017. One of the matches was between two rival teams, which, according to the portuguese government, introduces significant traffic on the streets and reduces the citizens' availability to move to a voting place (as the traditional voting system requires). The government went further and there are plans to change the law in order to forbid certain sport events at election days. In the case of portuguese local elections, the lack of mobility created an unnecessary tension between the citizens, the government and sports organizations.

Another possible scenario is the scenario where a voter is registered to vote in Lisbon. A business meeting, which the voter must attend, is scheduled at the last minute, to be held in Paris during the election period. With the traditional voting method, the voter would not be able to cast his vote.

With the global presence of the Internet nowadays, there is a tendency to replace traditional paper-based elections with Internet voting systems. The main reasons for the development and use of Internet voting systems are that they significantly raise the turnout rate (because mobility is improved), reduce administrative costs and tallying time. However, in Internet voting systems there is the issue of security on the devices that the voters use to cast the votes. This issue also compromises the goal of democracy. Consider the scenario where voters use smartphones to cast their votes through the Internet. Suppose that a third party exploits a vulnerability on Android and is able to install malware on the smartphones of thousands of voters. In this scenario, the third party that controls the smartphones and can change the vote of thousands of citizens, potentially changing the outcome of the election and affecting the future of an entire nation.

1.2 Goal and Requirements

Thus, the goal of this dissertation is to design, implement, and evaluate an Internet voting system called *TrustedVote* that allows voters to vote anywhere with high level of security. Special focus is given to the need for a trusted software base in the client used by the voter. From this goal, the following requirements are considered:

1. **Accuracy** - it is not possible for an invalid vote to be counted in the final tally.

2. **Integrity** - a malicious attacker cannot, arbitrarily or in a deterministic way, modify a vote without detection at the client, communication channels or server.

3. **Democracy** - only authorized voters may cast a vote and an eligible voter may only cast one vote.

4. **Privacy** - no entity besides the voter learns how he cast his vote. He is not able to prove to a third party how he voted.

5. **Verifiability** - any independent entity is able to verify that all votes were counted correctly. Additionally, a voter can verify if his vote was recorded correctly.

6. **Robustness** - the protocol should consider the following robustness requirements:

   (a) **Availability** - the system should be available during the election period. Furthermore, the client platform that the voter uses should execute the code that creates the vote and introduces privacy, integrity and verifiability without any noticeable overhead to the voter.

   (b) **Collusion Resistance** - the protocol should be resistant to collusion of corrupt voting authorities.

   (c) **Malware Resistance** - the insecure platform problem is defined as the insecurity that is found at the vote casting platforms because they are uncontrolled environments vulnerable to attacks. This problem should also be mitigated by tolerating malware in the client voting machine. Moreover, it is fundamental that the code that is security critical is minimal, i.e. the size of the Trusted Computing Base (TCB) is minimal.

7. **Mobility** - *TrustedVote* should not impose mobility restrictions to the voter. The voter has the freedom to vote anywhere.

8. **Usability** - we define Usability as follows: to successfully cast a vote, it is not required for the user to acquire special or dedicated devices that are not widely available. Moreover, from the point of view of the voter, the voting process imposed by the protocol should be intuitive and easily recognizable.

Thus, we consider the requirements that a non-electronic voting scheme should have, such as accuracy, integrity, democracy, privacy, verifiability, collusion resistance, availability and usability. In addition to non-electronic voting requirements, we consider: 1) mobility, that non-electronic voting cannot achieve because a voter must move to a voting booth to cast his vote and, 2) malware resistance.

1.3 Research History

The design and implementation of an 100% secure Internet voting system is obviously difficult. Protocols that achieve the core security properties (accuracy, integrity, democracy, privacy and verifiability) and robustness [4, 5, 7, 8], have serious usability problems. On the other side, usable e-voting schemes that guarantee the core security properties protecting the vote at server-side [1, 5, 7–9, 9, 10], fail to provide malware resistance. They do not consider malware in the voting client machine. As a matter of fact, malware in the operating system is able to perform arbitrary operations on the vote before being encrypted, compromising the privacy and integrity of the vote without detection.

Thus, there is the need to tolerate malware in the voter's computer while maintaining usability. For that purpose, a Trusted Execution Environment (TEE), i.e., a special area of the main processor that executes in isolation from the rest of the hardware, is a fundamental aspect to consider as part of the solution. With a TEE, it is possible to leverage secure storage and the isolation feature to perform the cryptographic steps of an e-voting algorithm, guaranteeing privacy of the vote even to the operating system. For instance, ARM TrustZone is an example of a TEE available in ARM processors that allows execution of code and services isolated from the operating system.

1.4 Solution and Contributions

In this dissertation we introduce *TrustedVote*, an Internet voting system that allows voters to vote anywhere with high level of security. In *TrustedVote*, we inherit the network protocol, cryptography scheme and entities from the EVIV protocol [11]. The EVIV protocol provides malware resistance on the client, but it is not usable according to our definition in Section 1.2. The architecture of the *TrustedVote* client that runs in the voter's devices does not require a dedicated device from the voter while guaranteeing malware resistance.

The client is responsible for creating a vote, encrypting it and providing a user interface so that voters can cast their vote, while keeping privacy and integrity of the vote even to the operating system running in the mobile device. We split the client in two components: 1) a small trusted component that executes sensitive code and, 2) an untrusted component that implements the steps of the voting protocol and provides a user interface. In the context of an voting client (the mobile device), we consider that the code that creates and manipulates the vote, i.e., has read and write access to the voter's intention, is considered sensitive code. The sensitive code must be trusted, and executed in isolation in order to provide malware resistance. We use ARM TrustZone technology to enable the isolated execution of the trusted component from the remaining of the software (operating system and other applications). By executing the sensitive code on the secure environment provided by ARM TrustZone, we ensure that the vote and the cryptography used to create it are not tampered, guaranteeing privacy and integrity.

The contributions of this dissertation are summarized are follows:

- Architecture of the client system that is running in the mobile devices of the voters.
- Implementation of a prototype of the *TrustedVote* client on the i.MX53 Quick Start Board, which is TrustZone-enabled. The prototype runs a command line tool over Linux on the normal world of ARM TrustZone, while the sensitive code is executed in the secure world on top of Genode *base-hw* microkernel.
- Experimental evaluation of the prototype running in the i.MX53 Quick Start Board, showing that the TCB is small and the overheads introduced by the TEE are not noticeable by the voter.

1.5 Document Structure

The remaining of this dissertation organized as follows:

- Chapter 2 introduces the world of Internet voting. The first part of this Chapter explains the theoretical concepts behind Internet voting systems. The second part describes the state of the art of Internet voting systems. In the last part, we present the main concepts of TEE technologies.
- Chapter 3 details the architecture of *TrustedVote*.
- Chapter 4 discusses the details of the prototype that was implemented.
- Chapter 5 evaluates the implemented prototype regarding the requirements devised before (Section 1.2).
- Chapter 6 concludes this dissertation, summarizing what was developed and the results. Also, we provide insights on what can be in the future with relation to the use of trusted computing in e-voting.

2

Related Work

Contents

2.1 Introduction 7
2.2 Background 7
2.3 Prêt À Voter 11
2.4 Helios 12
2.5 REVS 13
2.6 Java Card E-voting 15
2.7 Du-Vote 17
2.8 EVIV 19
2.9 Biometric-Based Voting Protocol 21
2.10 Country E-voting Implementations 22
2.11 Trusted Execution Environments 25
2.12 Summary 30

2.1 Introduction

This Chapter outlines the related work and is divided in four main parts. The first part describes the theoretical concepts behind Internet voting systems. The second part contains an overview of the state of the art regarding electronic voting systems. We also informally discuss why each system does not meet all the requirements devised in Section 1.2. In the third part of this section, we discuss e-voting mechanisms implemented by some countries and their respective experience, highlighting the security problems and usability issues found at those implementations. In the final part, we overview TEE technologies and how they can be used in the context of e-voting. Finally, we conclude with a summary of all related solutions.

Electronic voting protocols use secure channels to perform network communications, such as Secure Sockets Layer (SSL) and Transport Layer Security (TLS). They assume the existence of Certification Authorities that certificate asymmetric public keys of the protocol entities.

2.2 Background

The Internet voting systems in the literature can be divided in three main categories: based in mix-nets [9,12–14], based in homomorphic encryption [5,15], and based on blind signatures [2]. This Section introduces the theoretical details of each technique, the ElGamal cryptosystem and its exponential variant that are widely used in the e-voting systems.

2.2.1 ElGamal cryptosystem

The ElGamal cryptosystem [16] is an asymmetric key encryption algorithm used in several voting schemes. Its security depends on the difficulty of computing discrete logarithms.

The key generation process of entity A starts by choosing a cyclic group G of order q with generator α. Then, A chooses a random x greater than 0 and smaller than q and computes $y = \alpha^x$ mod q. The entity A publishes his public key (G, q, α, y) and keeps x as his private key. The encryption (E) and decryption (D) operations of a message m are defined in the following way:

$$E(m) = (u, v) = (\alpha^r, my^r) \text{ mod } q, \text{ and } r \text{ is random} \quad (2.1)$$

$$D(u, v) = m = v \times (u^x)^{-1} \text{ mod } q \quad (2.2)$$

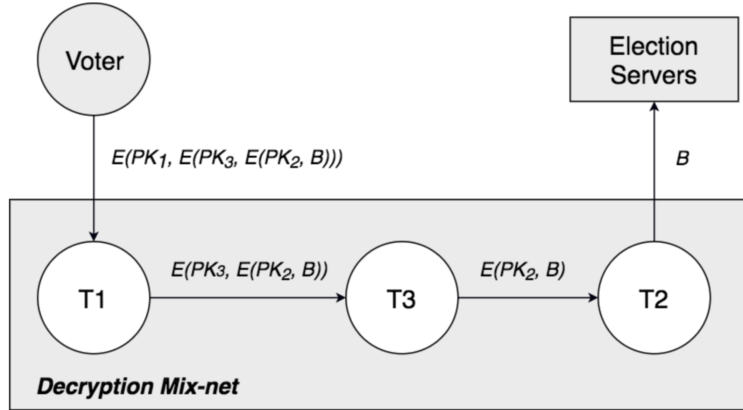

Figure 2.1: A decryption mix-net in the context of e-voting with 3 trustees.

2.2.1.A Exponential ElGamal cryptosystem

The exponential ElGamal cryptosystem is a variant of the ElGamal cryptosystem where the value α^m is encrypted instead of the message m. For exponential ElGamal the encryption E operation is defined in the following way:

$$E(m) = (u, v) = (\alpha^r, \alpha^m \times y^r) \text{ mod } q\text{, and } r \text{ is random} \tag{2.3}$$

However, when decrypting a ciphertext (u, v) with the decryption function D of Equation 2.2, only the value of α^m can be obtained. Having α^m and computing m requires solving the discrete logarithmic problem, so, exponential ElGamal is used when the message m is small and can be brute forced in useful time.

2.2.2 Mix-nets

A mix network (mix-net), proposed by Chaum [17], is a protocol with the goal of hiding communication between a sender and a receiver. The main component of a mix network is the set of trustees. A trustee is a server that act as an intermediary between a sender and a receiver. The trustees, forming a chain, receive messages from multiple senders and shuffle them before sending to the next trustee, breaking the link between the sender and receiver.

Mix-nets are used in the context of voting systems to enable anonymization of the votes by hiding the links between voters and the election servers. The two most widely used mix-net protocols in voting systems are the *Decryption mix-net* [17] and the *Re-encryption mix-net* [18, 19].

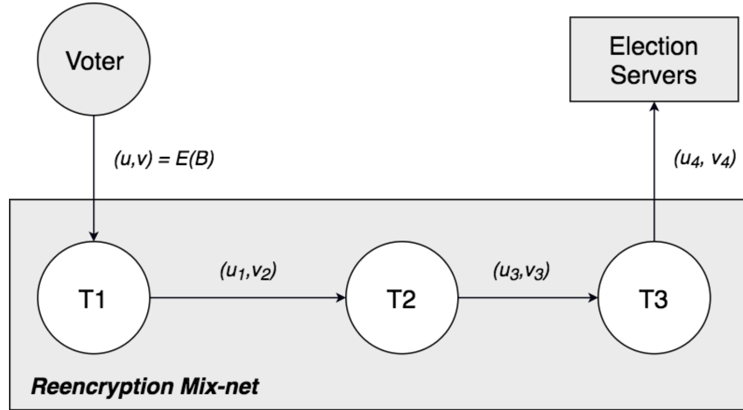

Figure 2.2: A reencryption mix-net in the context of e-voting with 3 trustees.

2.2.2.A Decryption Mix-net

In the first phase of a decryption mix-net the voter gathers the public key of each trustee $PK_1, PK_2, ..., PK_n$ and encrypts its ballot B in a random layered encryption scheme such that:

$$C = E(PK_1, ...E(PK_{n-1}, E(PK_n, B))) \tag{2.4}$$

Then, each trustee in the chain is responsible for the decryption of a layer of encryption and shuffle the ciphertexts before sending to the next trustee. The last trustee in the chain decrypts the final layer and delivers B to the voting servers. Figure 2.1 shows an example of a decryption mix-net in the context of e-voting. This decryption mix-net is composed by 3 distinct trustees. When the voter decided to send his ballot, he chose a random order of trustees ($T1$, $T3$, $T2$) and sent the layered encryption $E(PK_1, E(PK_3, E(PK_2, B)))$ to the first trustee. Each trustee then removes a layer of encryption using the corresponding private key and the final trustee $T2$ sends the decrypted ballot to the election servers.

This scheme is similar to the onion routing of the Tor network [20], which is a network of servers that route traffic in the Internet with the purpose of hiding Internet activity of users.

2.2.2.B Reencryption Mix-net

The reencryption is a property of some cryptographic systems that allows to produce two different random ciphertexts of the same message m.

For example, the ElGamal cryptosystem (Section 2.2.1) allows reencryption. Let (G, q, α, y) be an ElGamal public key and (u, v) be the ciphertext of a message m. Then, the re-encryption of (u, v) can be represented as $(u\alpha^s, vy^s) = (\alpha^{r+s}, my^{r+s})$, which results in a new random ciphertext that can be decrypted to the message m at the final destination.

This property is used by the trustees to mix the votes in an unrecoverable way before sending to the

next trustee in the chain. Figure 2.2 demonstrates an example of a reencryption mix-net in the context of e-voting with 3 trustees. First, the voter encrypts his ballot B with the election public key (where its private counterpart is only known by the election servers) and sends it to the first trustee. Each trustee is responsible to receive a ciphertext (u_i, v_i) and reencrypt it resulting in (u_{i+1}, v_{i+1}). The reencrypted ciphertext is send to the next trustee in the chain. The last trustee in the chain sends the final ciphertext to the election servers, where the original ballot B can be decrypted.

2.2.3 Homomorphic encryption

The homomorphic encryption is a property of some cryptographic systems that allows operations to be performed directly in the ciphertexts such that the decryption of the result matches the operations made on the plaintext.

ElGamal (Section 2.2.1) is a partial homomorphic cryptosystem that supports homomorphic multiplication. Let (G, q, α, y) be an ElGamal public key, and let $E(m_1) = (\alpha^r, m_1 y^r)$, $E(m_2) = (\alpha^s, m_2 y^s)$ be the encryption of a message m_1 and m_2, respectively. Then

$$E(m_1) \times E(m_2) = (\alpha^r \times \alpha^s, m_1 y^r \times m_2 y^s) = (\alpha^{r+s}, (m_1 \times m_2) y^{r+s}) = E(m_1 \times m_2) \quad (2.5)$$

Other example of a partial homomorphic cryptosystem is exponential ElGamal (Section 2.2.1.A) that supports homomorphic addition. Using the same parameters as the example above, then

$$E(m_1) \times E(m_2) = (\alpha^r \times \alpha^s, \alpha^{m_1} y^r \times \alpha^{m_2} y^s) = (\alpha^{r+s}, (\alpha^{m_1+m_2}) y^{r+s}) = E(m_1 + m_2) \quad (2.6)$$

Cryptosystems that support arbitrary homomorphic operations are called *fully homomorphic cryptosystems*. These systems allow the construction of a program with any required functionality (e.g., a voting system) that accepts encrypted inputs (e.g., the votes) and returns the encryption of the output (e.g., the final tallying) without having knowledge of the raw inputs. The Gentry-Sahai-Waters cryptosystem [21] is a fully homomorphic cryptosystem that supports homomorphic multiplications and additions.

2.2.4 Blind signatures

A blind signature, introduced by Chaum in [22], is a signature scheme that allows an authority to sign a message m from a sender without knowing its raw contents.

Consider the RSA cryptosystem, a sender V and a signing authority A with RSA key (n, p, q, e, d) where $n = p \times q$ is the modulus, e is the public exponent and d is the private exponent.

First, V computes the blinded message $m_r = mr^e \bmod n$, where r is a random number (called

the blinding factor) and sends it to the signing authority A. Now, A signs the blinded message m_r by computing $s' = (m_r)^d$ mod n and sends it back to V. The sender V can now remove the blinding factor r in s' by computing

$$s = s'r^{-1} = (mr^e)^d r^{-1} = m^d r^{ed} r^{-1} = m^d \text{ mod } n \quad (2.7)$$

which is the signature of the message m by A. Blind signature schemes can be used in Internet voting systems to separate the authentication of a voter (in order to verify that he is eligible) to the authentication of a ballot, providing privacy as it removes the link between the voter and its ballot.

2.3 Prêt À Voter

The original Prêt À Voter voting scheme [14] were first proposed in 2005 by Chaum *et. al.* Since then, several developments and improvements of this voting scheme have been proposed by Ryan *et. al* [1, 9, 10, 23].

Prêt À Voter assumes the existence of:

- a Registration Authority that registers and authenticates the authorized voters,
- a reliable and tamper-proof public Bulletin Board,
- an Election Authority server that is responsible to collect the votes, update the public Bulletin Board and perform the final tallying.

Donald	
Barack	
Alice	
Crystal	
Edward	
	a6Gq21p

Figure 2.3: The Prêt À Voter ballot (taken from Ryan *et. al* [1])

The key aspect of this voting scheme is the Prêt À Voter paper ballot (Figure 2.3). The ballot is split in two halves that can be physically separated. The left half contains the candidate names in random order and the right half contains the empty boxes where the voter should place his option. The second half also contains encrypted information, called the *onion*, that allows the system to have knowledge of the original candidate order. The ballot was originally implemented in paper, but there are flavors of Prêt À Voter that implement it fully electronically [24]. When the voter casts his vote, he detaches and destroys

the left side and places the right side in a scan reader to be sent to the election authorities. The random order on the left side ensures privacy of the vote.

Several developments and versions of Prêt À Voter were proposed. Generally, they implement different ways of mixing, decrypting and tallying.

The original Prêt À Voter instances [9, 14] use a decryption mix net of trustees to break the link between the voter and his ballot. In these instances, the election authorities prepare the ballots in advance by randomizing the candidate order and computing the onion for each one. The election authorities build the onion by encrypting the encoded candidate order with the public keys of all trustees in the mix net. The auditors, that have the responsibility of auditing the election, use the Randomized Partial Checking [25] method to verify the correct behavior of the mixnet. After the election period, the election authorities tally the votes and announce the results.

Instead of mixnets, homomorphic encryption could also be used to perform the tallying of the votes. In 2006, Adida and Rivest proposed the Scratch and Vote [15] protocol that relies on the simplicity of the Prêt À Voter paper ballots and uses homomorphic encryption, provided by the Pallier cryptosystem, to compute the final tallying of the votes. In Scratch and Vote, a set of trustees computes a shared asymmetric key pair (K_{priv}, K_{pub}) such that each trustee has a share of the private key K_{priv}. Therefore, the decryption of a ciphertext that was encrypted with the public key K_{pub} requires the cooperation of all trustees. The votes are encrypted with K_{pub} and sent to the election authorities. After the election, the vote tallying is performed by the election authorities and the trustees join to release the decrypted result of the election.

The Prêt À Voter schemes preserve the accuracy, verifiability, privacy and integrity of the election, but mobility is not achieved as the paper based ballots require that the voter moves to a voting booth. Democracy is achieved as it is assumed that a Registration Authority registers and authenticates only the authorized voters. Additionally, it is assumed that a single voter is not capable of voting multiple times. The protocol is not robust to the collusion of election authorities as they have the ability to insert votes of valid absentee voters in the bulletin board. Moreover, no explicit mechanisms are implemented to ensure availability of the election authority servers. The scan reader (the client machine), even infected with malware, has no ability to change or send valid votes to the Election Authorities without detection (malware resistance). From the point of view of the voter, the paper based ballot introduces simplicity into the voting process, achieving usability.

2.4 Helios

The first version of Helios [12] is a fully web voting system proposed by Adida in 2008. This protocol only assumes the existence of one entity, the Helios server, which is responsible for all stages of the

voting process. The Helios server has an ElGamal asymmetric key pair (Section 2.2.1) with public key $K_{pub} = (G, q, \alpha, y)$ and private key $K_{priv} = x$ for each election it runs. This key pair is generated by the Helios server when an user creates a new election.

In Helios, the voting process starts when the voter chooses the election to participate and casts a ballot. After this step the system shows the ballot ciphertext and randomness r, allowing the voter to confirm if the encryption was performed correctly. The authentication of the voter is only done after the vote is cast, allowing anyone to test if the ballots are being encrypted correctly. The ballot ciphertext is then posted in a public Bulletin Board if the voter is eligible to vote.

After the election is closed, the anonymization process is enabled by the election administrator. The ElGamal re-encryption mixnet (Section 2.2.2.B) is used to perform the shuffling of all ballot ciphertexts in the public Bulletin Board. Then, all votes are decrypted using K_{priv} and the final tally is computed. The shuffling process yields a proof of correction and the decryption process publishes in the Bulletin Board a proof of correct decryption [26] for each ballot.

If the Helios server is corrupted, the privacy, integrity, democracy and accuracy of the election are compromised as this entity has the ability to insert, decrypt and change votes. Also, the organization with access to the election private key K_{priv} can decrypt ballots and acknowledge how each voter voted, compromising the privacy. The proof of correct shuffling and the decryption proof enable the public auditing of an Helios election, achieving verifiability. The Helios server is also a single point of failure, which is a threat to robustness in terms of availability. Estehghari and Desmedt [27] showed how a malicious candidate can upload a malicious PDF file that installs malware in vulnerable client machines. As a consequence, the malware is able to trick voters into accepting a malicious ballot, compromising the entire election. Mobility is preserved because the vote can be cast using the browser of a mobile phone. Usability is high since the voting process only requires a password authentication and a browser from the voter to successfully cast a vote.

2.5 REVS

REVS [2] is an e-voting protocol proposed by Joaquim *et. al* in 2003. It assumes the existence of the following entities:

- **Commissioner** - supervises the election. It prepares the election configurations and stores the election asymmetric key pair (K_{pub}, K_{priv}).
- **Ballot Distributors** - distribute empty ballots signed by the Commissioner to the voters.
- **Administrators** - decide if a ballot is accepted from a voter or not. Each administrator has an asymmetric key pair that is used to sign a voter ballot if it is accepted.

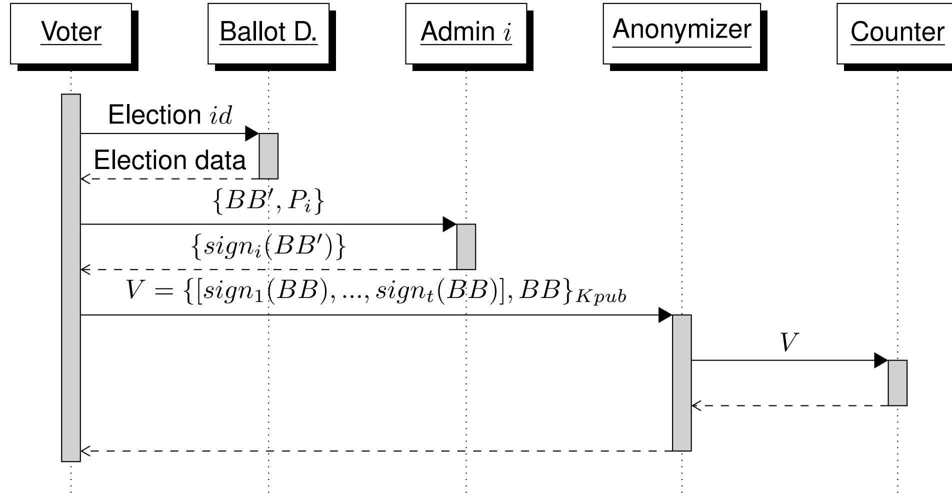

Figure 2.4: REVS protocol overview (adapted from Joaquim *et. al* [2])

- **Anonymizers** - protect the voter from being associated with his ballot.
- **Counters** - verify the administrators' signatures of each ballot and perform the final tally.

In first phase of the REVS protocol (Figure 2.4), the voters contact the Ballot Distributor server to retrieve a blank ballot, the election public key K_{pub} and the configurations for a specific election.

In the second phase of the protocol the voter casts the ballot BB, computes a random blinding factor r and applies it to the digest of the committed ballot. Let BB' be the result of the previous operation. The voter sends BB' to $t > \frac{n}{2}$ administrators, where n is the number of administrators. Each voter authenticates himself to each administrator using a different password. The password for a single administrator is automatically generated using a strong secret provided by the voter. This mechanism prevents the impersonation of a voter by an administrator, as it cannot compute the passwords for other administrators. At each request to sign a ballot, the administrator checks if he has already signed for the requesting voter. He returns the previously saved signature if he has already signed; if he has not signed yet, he signs the ballot and saves the signature. The voter removes the blinding factor r for each obtained signature.

In the last phase of the protocol the voter gathers his ballot and the $t > \frac{n}{2}$ signatures returned from the administrators and builds the voter package V. He sends V encrypted with the election public key (K_{pub}) to the Anonymizer server. The Anonymizer hides the voter's IP address, shuffles the votes, and introduces random delays in the delivery to the Counter server. The random delays and shuffling prevent attacks where an third party is able to discover a voter's encrypted ballot from the time of voting. When the election period is over, the Commissioner releases the election private key (K_{priv}) and the Counters decrypt all the submitted ballots. The Counters perform the final tally, verifying if each ballot has $t > \frac{n}{2}$ signatures from the Administrators. Any vote that does not have the required number of signatures is

discarded. Repeated votes are also discarded at this stage.

The REVS protocol is available as long as one Ballot Distributor, $t > \frac{n}{2}$ Administrators, one Anonymizer and one Counter are available. The availability can be improved by replicating these servers. However, Lebre *et. al* [28] noticed that a malicious voter may send the same package (or garbage) several times to the Counter, because this entity cannot authenticate the voters. The Counters would be storing data that could only be removed at the end of the election, compromising the availability of the system. Verifiability is achieved because any independent entity is capable of auditing the administrators' signatures of each ballot and count all the votes. Democracy holds because a voter cannot obtain two valid votes as he would need $t > \frac{n}{2}$ signatures from the administrators. The election is accurate because an invalid vote cannot be part of the final tally, as anyone can verify the administrators' signatures of every submitted ballot.

However, it is assumed that the machine used by the voter must be trusted and follow the protocol. If this assumption is not hold, then the integrity and privacy of the vote are compromised. This compromises the robustness of the protocol. The system is usable because the voter only has to install a client application that automatizes the protocol tasks and cast his vote. The voter has the freedom to vote anywhere (mobility) as he can install the client application in a mobile phone for example.

2.6 Java Card E-voting

The Java Card technology facilitates the development of Java applications to smart cards. A smart card is a device with very limited amount of memory and processing power. Only a small subset of the Java language and features are available, but it is suitable and powerful enough for the development of object oriented applications [29]. The smart card is tamper resistant. As a consequence, it provides an environment where the processor instructions and contents of the memory cannot be eavesdropped or tampered.

The online voting system proposed by Mohammadpourfard *et. al* [3] takes advantage of the Java Card 3 technology to enhance its security.

The overview of the protocol is in Figure 2.5. The scheme assumes the existence of the following entities:

- **Registration Authority (RA)** - accountable to register and authenticate the eligible voters.
- **Administrator** - responsible to distribute empty ballots.
- **Validator** - signs cast ballots.
- **Tallier** - updates the public Bulletin Board, collects and counts the votes.

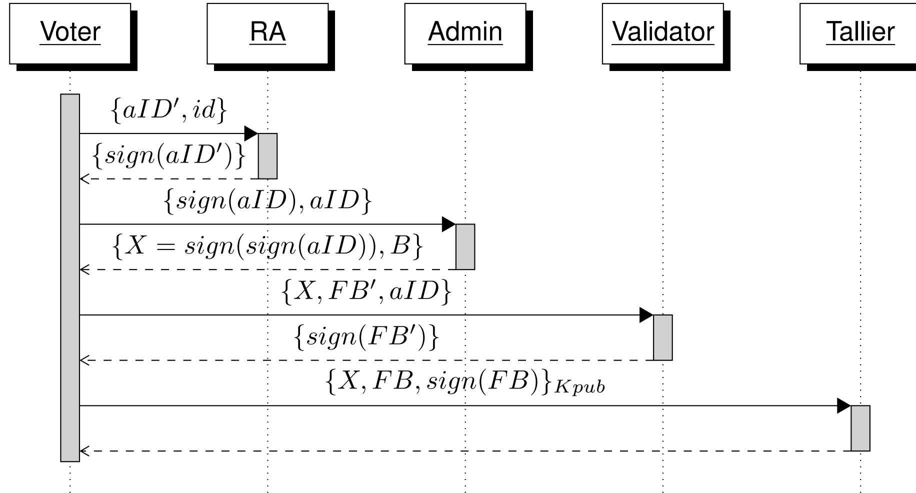

Figure 2.5: Java Card 3 protocol overview (adapted from Mohammadpourfard *et. al* [3])

It is assumed that each entity has an asymmetric key pair and the other entities (including voters) know the corresponding public key of each of them. Moreover, it is assumed the existence of an asymmetric key pair (K_{pub}, K_{priv}) where K_{priv} is split among all entities and candidates, and K_{pub} is known by all entities. Every exchanged message in the network is encrypted with the public key of the recipient.

The voter smart card starts by generating a secret alias aID and blinding it (aID') with a blinding factor r. Along with the aID', the smart card sends to the Registration Authority the voter identification id. The Registration Authority checks if the voter is eligible to vote and signs aID' with its private key. The voter removes blinding factor r to retrieve the signature of aID and he may now use it as a token to authenticate himself to other election authorities.

The voter contacts the administrator to retrieve an empty ballot B. The empty ballot B and the signature of aID is signed by the administrator.

The voter fills the ballot B with his choice, producing the filled ballot FB. The smart card blinds it (FB') with blinding factor t and sends it to the Validator, annexing the double signature of aID. The Validator returns the signature of FB' and the voter removes the blinding factor t, obtaining the signature of FB. Finally, the voter sends the filled ballot FB and all the gathered signatures encrypted with the election public key K_{pub} to the Tallier server. After the election period, the tallier decrypts all the votes with the cooperation of all entities and candidates, performs the tallying and posts the results and votes in a public Bulletin Board.

Any filled ballot FB that is submitted with invalid signatures is discarded, providing accuracy. Democracy holds if the Registration Authority only signs aliases to eligible voters. Privacy is preserved since: 1) the Registration Authority is not aware of which aID the voter has chosen, 2) the authentication of a voter is performed based on aID and not on the real voter's identity, and 3) the filled ballot is blinded or encrypted with K_{pub} when sent via network. The usage of signatures in the empty ballot B and filled

ballot FB also prevents that a malicious attacker modifies the vote, ensuring integrity. As the signatures and filled ballot FB are published in a public Bulletin Board, any independent entity can verify the validity of every vote and the election results, proving verifiability. As the election private key K_{priv} is shared among all candidates and entities, a single and corrupt entity cannot read votes before the end of the election. This protocol requires that a large number of entities and candidates collude to compromise the election, ensuring collusion resistance. The smart card performs the cryptographic steps of the protocol and is tamper resistant, so robustness regarding malware resistance is achieved. The availability of the protocol is ensured if one Registration Authority server, one Administrator, one Validator and one Tallier are available. The availability of the protocol can be improved with replication of these servers. However, replication mechanisms were not considered by the authors of this protocol. Furthermore, the system assumes that all citizens have access to a Java smart card reader, which affects usability. However, the Java smart card reader can be attached to a mobile phone, achieving mobility.

2.7 Du-Vote

Du-Vote [4] is an internet voting protocol that makes use of a simple hardware device in order to completely remove trust from the voting machine. The token is a simple hardware that should have a keypad and screen to accept short decimal inputs and outputs, store a secret key and support module arithmetic. As shown in Figure 2.6, the voter V needs access to a computer platform P and a token H in order to cast his vote. The server S is managed by the election authorities and the protocol makes use of a public Bulletin Board. An election nonce I computed by the election authorities is posted on the public Bulletin Board. Du-Vote assumes the existence of a set of decryption tellers T that is responsible for keeping an election private key secret until the end of the election.

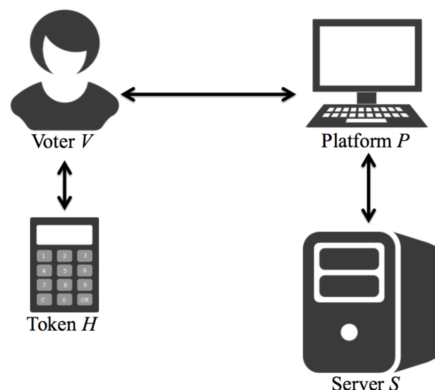

Figure 2.6: Du-Vote architecture overview (taken from Grewal *et. al* [4])

Prior to the election, each voter registers himself to the election authorities and a password is chosen

Candidate	Column A	Column B
a1	A_1^*	B_1^*
a2	A_2^*	B_2^*
...	...	...
a4	A_4^*	B_4^*
Enter your vote code here		

Figure 2.7: Du-Vote voter's code card (adapted from Grewal *et. al* [4])

by V to perform authentication . The election authorities issues a token H configured with a secret key K to each voter V via postal mail or in person. Therefore, the election server S has an association between the voter V and his respective secret key K. Also, the set of trusted decryption tellers T computes a shared El-Gamal election key pair $(K_{pub} = (G, q, \alpha, y), K_{priv} = x)$ such that the private key is split among all decryption tellers [30]. Consider that each decryption teller $t \in T$ has a private key x_t and the corresponding public key $y_t = \alpha^{x_t}$. The product of all public keys $y = \prod_t y_t$ is used as in the public key K_{pub}. It is required the cooperation of all decryption tellers to perform a decryption of a ciphertext encrypted with K_{pub}.

The voting phase starts when V uses P to authenticate himself to S, using the password provided during the registration phase. Then P gathers the candidate list from the public Bulletin Board, computes and displays to the voter the code card. The concept of code cards were first introduced by Chaum [31] in the SureVote system. A code card is an association between a candidate name and a vote code, i.e. a code that the voter uses to select a candidate.

In Du-Vote the vote codes are computed in the following way. Let n be the number of candidates on the election and κ be a security parameter. The computer platform P starts by computing a set of $2n$ decimal codes $\{c_1, c_2, ..., c_{2n}\}$ with length κ determined by the election nonce I and the voter ID. This is implemented by seeding a pseudo-random number generator with $hash(I, voterID)$. P computes $\{A_1^*, A_2^*, ..., A_n^*\}$ as a random shift of $\{c_1, c_2, ..., c_n\}$ and $\{B_1^*, B_2^*, ..., B_n^*\}$ as a random shift of $\{c_n, c_{n+1}, ..., c_{2n}\}$. P now presents to the voter the ballot shown in Figure 2.7.

The voter V flips a coin an inserts into H the vote codes in column A or B depending if the result is tails or heads, and enters the vote code of the chosen candidate that is in the opposite column. H computes and displays $C* = (Kh^d)*$ where d is the chosen vote code by the voter and the $*$ operation returns the last κ digits of the result.

V inserts $C*$ in P and P sends to the server S the vote codes and $C*$. As the computation of the vote codes is deterministic for anyone who knows I and the $voterID$, the server S is now able to confirm that P computed correctly the vote codes and decrypts $C*$ to determine the candidate that V chose. The decryption of $C*$ is made using a brute-force approach with input space size of $2n^2$ and is only possible for the server S that knows K. The server S reencrypts the chosen candidate with the election public

key K_{pub}, posts it on the public Bulletin Board, and produces a non-interactive zero knowledge proof to show that it followed the protocol correctly.

After the election period, the set of decryption tellers T join, decrypt all the votes on the public Bulletin Board using their own share of the private key K_{priv} and tally the votes independently. Du-Vote does not require a specific way to perform the vote tallying. As the votes are encrypted using exponential El-Gamal encryption, homomorphic tallying could be used.

The proposed scheme in Du-Vote tolerates malware in the computer platform P (malware resistance) because it is in the token H that the voter chooses the candidate and the encryption of the chosen candidate $(C*)$ is computed. The usability is low, specially if the number of candidates is high, as the user must insert all the vote codes produced by P into the token H. The server S is a single point of failure and no explicit mechanisms are considered to replicate it, compromising the availability of Du-Vote. The mobility of Du-Vote is not compromised because the hardware token H can be portable and the client platform P can be a mobile phone. Kremer and Rønne [32] showed that a phishing attack and if the hardware token H is used twice for different elections, then the privacy of the election is compromised. Moreover, they have shown how the server S and the computer P can collude to modify the choice of a voter to another random candidate, compromising collusion resistance and integrity. Verifiability holds as the non-interactive zero knowledge proof shows that the server S decrypted and published the vote correctly on the public Bulletin Board, allowing any external entity to verify the correctness of computations performed by S. Only one hardware token H is given to each eligible voter, so he can only vote once, ensuring democracy.

2.8 EVIV

EVIV [5] is an end-to-end verifiable Internet voting system that takes into consideration that the client platform may be insecure and controlled by a malicious attacker. EVIV takes into consideration the following entities and services:

- **Enrollment Service** - responsible for the enrollment of every voter.
- **Election Registrar** - service that voters register to vote on a specific election.
- **Ballot Box** - service that voters use to send their vote.
- **Verification Service** - each organization runs a verification service that verifies if the votes and receipts are correct and valid.
- **Trustees** - set of organizations and parties that keep secret an ElGamal (Section 2.2.1) election asymmetric key pair (K_{pub}, K_{priv}). Each trustee has a share of K_{priv} such that the decryption of a message requires the collaboration of $t < n$ trustees, where n is the total number of trustees [33].

- **Voter Security Token (VST)** - component that encrypts the ballot and authenticates the voter using digital signatures. In EVIV, the VST is implemented using a tamper-proof Java smart card containing the private key of the voter (K_{vPub}, K_{vPriv}).

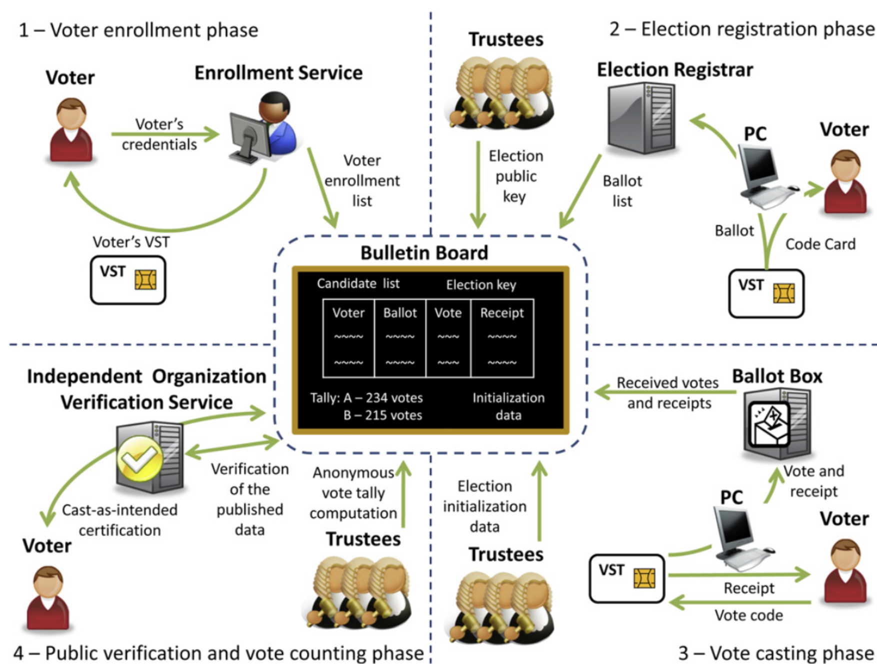

Figure 2.8: Overview of the EVIV protocol (taken from Joaquim *et. al* [5]).

The first phase of the EVIV protocol (Figure 2.8) is the enrollment phase where the voter registers himself to the Electoral Commission. This phase is off-line and is performed via the Enrollment Service. He receives his VST with the voter's key pair (K_{vPub}, K_{vPriv}).

In the second phase, the election setup phase, the Electoral Commission defines the candidate list and the trustees compute the election key pair (K_{pub}, K_{priv}).

In the next phase, the ballot registration phase, the voter connects his VST to a secure computer and the VST pulls the candidate list and the election public key K_{pub} from the Election Registrar. The VST generates a code card for the election and shows it to the voter. The EVIV code card contains a random vote code for each candidate and one confirmation code used by the voter to check if the vote was cast-as-intended. The goal of the voting codes is to provide a secure channel between the voter and the VST, without relying on a centralized code distribution scheme or trusting the voter's PC.

The voters may now use their PCs to insert the vote code of the desired candidate. The voter verifies the vote receipt returned by the VST and checks if the confirmation on their vote card is the verification code of the voted candidate. The client platform is responsible to send the vote and receipt encrypted with K_{pub} to the Ballot Box, which signs the verified data and publishes it on the Bulletin Board.

After the election period is over, the set of trustees gather all the encrypted votes and compute the homomorphic votes aggregation, i.e. the number of votes for each candidate (encrypted with K_{pub}). This is possible because the cryptographic scheme used by EVIV [5] is homomorphic in relation to the addition. The subset of t trustees join, decrypt the votes aggregation, producing a decryption proof. The tally results, the decryption proof and the homomorphic votes aggregation are published in the public Bulletin Board.

Any manipulation of the vote destroys its digital signatures, ensuring integrity. Accuracy is achieved because the VST is the only entity capable of producing a valid digital signature for a voter's vote. The vote is published in the Bulletin Board with the election public key K_{pub}, and the corresponding private key K_{priv} is split among several entities, requiring the collusion of trustees to violate privacy. EVIV is verifiable because any external entity is able to compute the homomorphic votes aggregation and validate its decryption proof that is published in the public Bulletin Board. Democracy is achieved because each voter only has one VST. The protocol promotes mobility while casting the vote as the client platform can be a mobile phone. The protocol is robust regarding malware resistance because the client platform has no way of compromising the integrity and privacy of the voter, as the vote codes create a secure channel between the voter and his VST. The system is available if the Election Registrar, the Ballot Box and the Bulletin Board are available. The Election Registrar and Ballot Box are stateless services that can be easily replicated to ensure availability. EVIV has low usability under the current implementation of the VST, because it assumes that all eligible voters have access to a smartcard reader.

2.9 Biometric-Based Voting Protocol

Alrodhan *et. al* [7] proposed a scheme that leverages biometrics as a trapdoor authentication mechanism. In a trapdoor authentication mechanism two distinct credentials are given to the voter: the *genuine* credential and the *mock* credential. If the voter authenticates uses the genuine credential then his vote will be counted and recorded. If the voter authenticates himself using the mock credential, his vote will not be counted in the final tally.

The system is composed of two election entities. The Authentication Center (AC) is responsible to produce a list of eligible voters, gather their biometrics and authenticate them during the election period. The Ballot Center (BC) is responsible to store the submitted votes by the eligible voters, and announce the final election results. It is assumed that AC and BC share a symmetric key K_s and that all network communications are performed via secure channels (e.g. SSL/TLS).

During the registration phase, the scheme proposed by Alrodhan *et. al* gathers from the eligible voters two fingerprints: one is the genuine *fingerprint* and the other is the *mock* fingerprint. This phase

requires voter physical presence at the AC.

During the election period, voters use a client application to gather a fingerprint F and choose a candidate to vote. Let B be the filled ballot by the voter. Then, the voter blinds B with blinding factor r. Let B' be the result of the previous operation. The voter sends B' and F to the AC. The AC returns the signature of B' and $\lambda = \{flag, nonce\}_{K_s}$ where $flag$ indicates if the voter used the *genuine* credential or the *mock* credential, $nonce$ is a number only used once to prevent replay attacks. The voter removes the blinding factor r from B' and delivers λ, B and the signature of B to the BC. The BC is capable of verifying if the signature of B is correct and if $flag$ indicates that B is a mock vote or a genuine vote. If the voter used the mock credential, then the vote is marked as *fake* in the database.

The final phase of the election is where the BC decrypts all valid ballots, counts the votes for each candidate and announces the final election results. Votes that were cast using *mock* credentials are discarded from the final tally.

Accuracy is preserved in this scheme because votes with invalid signatures or votes cast with the *mock* fingerprint are discarded from the final tally. The protocol does not provide malware resistance as a malware in the client platform can change and read the filled ballot B before sending it blinded to AC, compromising the integrity and privacy of the protocol. Availability is ensured if the AC and BC are available. The authors of the system did not considered explicit mechanisms to ensure the availability of the system, namely replication. Collusion resistance is not achieved since a corrupt AC and BC can learn associations between ballots and voters, compromising privacy. Democracy is achieved because the AC only signs a filled ballot B once per fingerprint. Mobility is preserved because the client application can be installed in a mobile phone with a fingerprint reader attached to it. Usability is low because it assumes that every voter has access to a fingerprint reader. This voting protocol is not verifiable because an external entity is not able to verify if all the recorded votes were counted correctly.

2.10 Country E-voting Implementations

This section describes the e-voting trials conducted by some countries, focusing on their experience regarding security and usability issues that were found. Many governments do not disclose the full details of their voting schemes, so this analysis is mostly performed based on reports by the election officials or security studies performed by independent investigators.

2.10.1 Estonia

Estonia has been using an Internet voting system for their parliamentary, regional and local elections since 2005. More than 30% of the votes were cast via Internet. In 2014, Springall *et al.* [34] performed a security analysis on their voting system. The election officials publish the system source code on

GitHub 2 or 3 weeks in advance. They replicated the voting system used by the government in a laboratory environment and acted as attackers. As the system does not have guarantees against malware resistance, Springall *et al.* developed client-side malware that is able to steal and change votes without detection. Regarding the server code, the authors discovered a denial of service attack that would prevent voters from sending ballots (compromising availability) and a shell injection vulnerability that would allow execution of arbitrary root commands by a system operator (compromising integrity and privacy of the votes). Additionally, the protocol is not end-to-end verifiable because the public is not able to verify that the votes recorded were counted correctly. Although the majority of the server software is available open source, the pieces of code that are critical for the security of the entire system (e.g. the entire client application) are not available to the public.

2.10.2 Switzerland

Switzerland implemented an e-voting system in Geneva and Zurich cantons [35]. In September 2004, the Geneva canton deployed its e-voting system for the regional and canton elections. 21.8% of the voters cast its ballot online. Zurich deployed in November 2005 an e-voting system for a referendum at regional and cantonal level and 20% of the voters used the system. The official documentation of the election is sent by mail with three weeks in advance. The documentation includes an unique voting card and a PIN number that can only be used once. During the election period, the voter uses his voting card, PIN number and date of birth to authenticate himself in the voting website. After the ballot submission, the voter receives a confirmation that the vote was stored by the system. No more details regarding the Swiss e-voting protocols were found. Despite the absence of reported security failures and the implementation of several security measures by the Swiss government [36], the insecure platform problem holds, as confidentiality of the vote could still be compromised by a malicious browser or malware in the voter's computer.

2.10.3 Washington, D.C

Washington, D.C Board of Elections and Ethics (BOEE) developed an open source Internet e-voting system for the November 2010 general elections. The BOEE staged a mock election in a full production environment and encouraged all citizens to test the security and functionality of the system. Wolchok *et. al* [37] started to look for vulnerabilities by analyzing the server application source code. They found a shell injection vulnerability in the code to encrypt a ballot submitted by the voter, allowing them to execute arbitrary commands on the election server by submitting carefully crafted ballots. From this point, the investigators were able to steal the election key pair, change the vote of every citizen, steal the credentials of the voters and hide their attack by modifying the web application logs.

Additionally, Wolchok *et. al* infiltrated the network infrastructure of the election servers and found a pair of unprotected web cameras in the election server room. All the vulnerabilities would allow other malicious intruders to acknowledge the movements and actions of security guards and system administrators in real time. They remained undetected until they left an explicit clue for the BOEE officials. The project was discontinued.

2.10.4 Norway

Norway decided in 2009 to develop the *E-valg 2011* e-voting system to be used in the 2011 municipal and regional elections [13]. The protocol assumes the existence of Ballot Box, a Receipt Generator and a Decryption Service deployed by the election authorities and an Auditor to verify the correctness of the election. Each voter has an ElGamal asymmetric key pair (Section 2.2.1) that is known to the Ballot Box and each entity has a known ElGamal public key. It is assumed that communications are secure (e.g. SSL/TLS). Furthermore, each voter has access to a vote card that associates each candidate with a pre-computed receipt code. The voting process starts when the voter chooses the candidate, signs the filled ballot and sends to the Ballot Box. The Ballot Box and Receipt Generator cooperate to compute the voting receipt and send it to the voter using an out-of-band mechanism (e.g. SMS). Using his vote card, the voter can verify if the returned receipt matches the chosen candidate. The voter now encrypts the vote using the public key of the Decryption Service and sends it to the Ballot Box through a mix net (Section 2.2.2) to provide privacy. After the election period, the Ballot Box sends the encrypted votes to the Decryption Service, where the votes are decrypted and mixed before being published for tallying.

Cortier and Wiedling [38] provided a mathematical proof that the Norwegian protocol guarantees vote privacy except when the Ballot box and Receipt Generator are corrupted. Also, it is assumed that the SMS channel is secure to send the verification codes, which Koenig *et. al* [39] showed to be false. This protocol does not provide malware resistance as there are no efforts to secure sensitive operations in the client application.

2.10.5 Australia

Australia deployed iVote as one of the largest instances of Internet voting. It was used in March 2015 for the New South Wales state elections and over 280,000 voters cast their vote through iVote.

Halderman and Teague [40] conducted an independent security analysis on the iVote system during the election period. When accessing the iVote website, they found that the browser loaded JavaScript code from an external server with insecure TLS configuration. On these conditions, a man-in-the-middle attack can inject arbitrary JavaScript code, compromising the privacy and integrity of the entire election.

The state of Victoria, Australia deployed the vVote [24, 41] system for their November 2014 elections.

The intent of the system was to allow blind and remote voters to cast their votes in a way that preserves the verifiability and usability properties. The vVote protocol is a flavour of the Prêt À Voter scheme (Section 2.3), where the ballots are implemented in an electronic way. In the two weeks of voting period, 1121 votes were cast in vVote. A survey made to the voters to assess the functionality and practicality of the system concluded that voters were satisfied with the voting experience. However, few voters understood the verifiability features stating that the vote receipt contained information on how they voted (which is not true). Despite the security and usability features of vVote, mobility is not achieved as citizens are required to move to a voting center to cast their vote.

2.11 Trusted Execution Environments

There is a clear trade off between malware resistance and usability in e-voting protocols. Protocols that tolerate malware in the client platforms have low usability and protocols with high usability do not tolerate malware. In the context of e-voting protocols, TEEs are able to provide malware resistance. They can protect against privacy and integrity attacks executed by malicious software in the operating system, without compromising usability.

A TEE is a dedicated area of the main processor that executes in isolation from the remaining of the hardware. It offers a secure environment for applications to execute. This Section delineates the main concepts of TEE technologies and their applications.

2.11.1 Intel SGX

Intel Security Guard Extensions (SGX) is a technology available in the Intel Skylake CPU micro-architecture that allows applications to ensure confidentiality and integrity to their computations even to a corrupted operating system and Basic Input/Output System (BIOS) [42, 43]. Intel SGX introduced the concept of enclaves. An enclave is a protected area of an application's virtual address space that adds confidentiality and integrity to the code and data inside it, even to privileged malware. SGX also implements enclave attestation, i.e., authentication to a remote party of the code running inside the enclave.

Figure 2.9 demonstrates the memory layout of an enclave. The enclave memory is associated with an application running in user mode, and, therefore, lives in its virtual memory space. The code, data and management data structures of the enclave are then stored in Processor Reserved Memory (PRM). PRM is a part of Dynamic Random Access Memory (DRAM) that cannot be accessed by software or Direct Access Memory (DMA) as the Central Processing Unit (CPU) memory controllers reject DMA operations on the region that is allocated for the PRM.

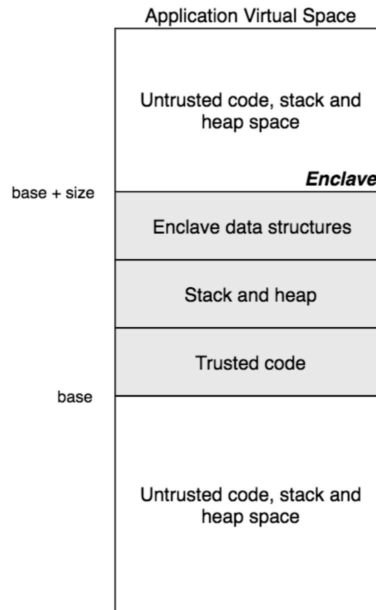

Figure 2.9: The memory layout of an enclave in the context of the application's virtual space.

Intel SGX-enabled processors add new specific instructions to their Instruction Set Architecture (ISA), such that the management of enclaves' life cycle and operations (such as creation, exit, memory allocation and permissions, creation of hardware cryptographic keys, etc.) is performed entirely in hardware. The ISA provides instructions only available in privileged mode (ring 0) to the enclave (such as memory management and debugging) and user mode instructions in ring 3 (e.g., creation of an enclave). Therefore, the ring protection enables that the enclave code runs in privileged mode and achieves high performance as it runs at the native processor speed.

However, as of today, Intel Skylake processors are widely available in server and desktop environments but not in mobile phones, compromising the mobility of an e-voting application if implemented in the Intel SGX technology.

2.11.2 ARM TrustZone

ARM TrustZone is a security extension architecture (Figure 2.10) available in ARM processors that allows execution of code and services isolated from the operating system [6]. This architecture not only gives confidentiality and integrity to any asset, but also reduces the development costs of security solutions and the size of the final TCB. The security extensions partition the system resources so that they are available only in one of two worlds: the secure world for the secure assets (code, data and peripherals), and the normal world for the remaining assets. Therefore, ARM TrustZone provides a TEE inside its secure world.

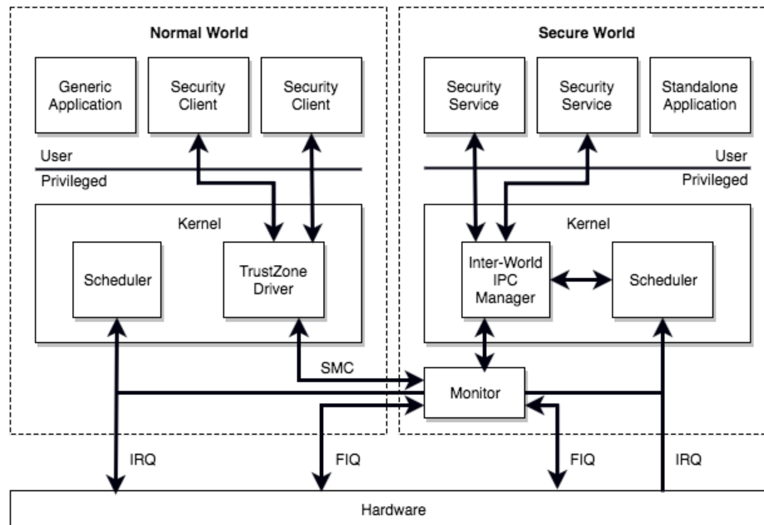

Figure 2.10: TrustZone software architecture with a dedicated operating system [6].

2.11.2.A Software Architecture Overview

Figure 2.10 shows the software architecture of a TrustZone-enabled application. In this architecture, the secure world is running a dedicated operating system. In this context, the software stack is split in two worlds of execution: 1) the normal world where the untrusted code runs, and 2) the secure world where the security sensitive subsystem is executed. At a given time, only one world of execution is running and each world has its memory space and access permissions to peripherals. In the normal world is where the rich operating system (such as Linux, Android, etc.) resides, providing multiple features to the user. In the secure world is where the secure dedicated operating system lives, providing basic scheduling, memory management and inter-world communication.

The application is divided (by the developer) in two components, an untrusted Security Client running as a user application of the rich operating system and a trusted Security Server running as an application of the secure operating system. Therefore, when a Security Client wants to delegate execution to a Security Service, it issues a system call to the rich operating system, which in turn uses the TrustZone driver to execute a Secure Monitor Call (SMC) instruction causing a software exception. The Secure Monitor, running in secure world, handles the exception and switches worlds, and transfers the processor control to the secure operating system, where the Security Service has the opportunity to execute.

In order to prevent an attacker from modifying the secure operating system image, and therefore compromising the secure world, the authenticity and the integrity of the secure operating system is ensured by a secure boot mechanism that a TrustZone-enabled processor also implements. This secure boot mechanism is executed by the software located in ROM bootloader, where it uses public key cryptography to verify whether the integrity and authenticity of the image has been compromised or not.

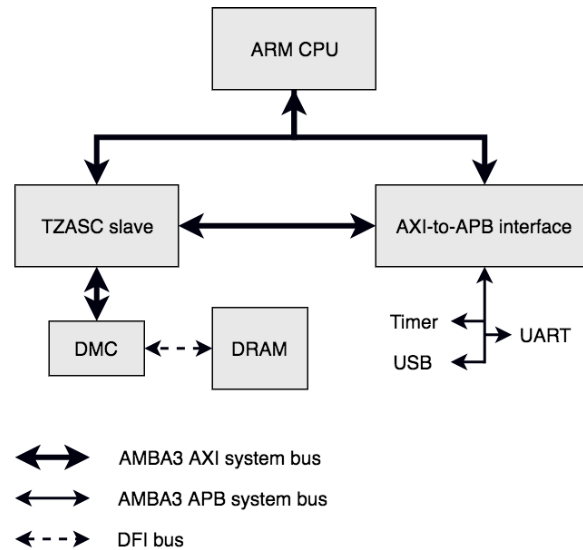

Figure 2.11: Hardware bus architecture overview.

In the next Section we explain how the world isolation is implemented in the hardware layer.

2.11.2.B Hardware Architecture Overview

The TrustZone hardware guarantees a secure border between both normal world and secure world, where resources allocated for the secure world cannot be accessed by the normal world. The hardware ships with the AMBA3 AXI system bus and the AMBA3 APB peripheral bus, which are fundamental components that isolate the hardware resources between worlds.

Figure 2.11 shows how these buses are used to achieve resource isolation. The main system bus (AMBA3 AXI system bus) has an extra control bit (NS bit) for every read and write transaction performed. When $NS = 1$, then the transaction is performed by the normal world, and in the secure world otherwise. This allows that the bus slaves read the NS bit and ensure the required security separations. For instance, the Dynamic Memory Controller (DMC) component does not support internally the creation of secure world and normal world partitions, the TrustZone Address Space Controller (TZASC) is a AXI bus slave that uses the NS bit available in the AXI bus to partition the DRAM into regions for Secure/normal world. The AMBA3 APB peripheral bus does not include the NS bit, and it it the responsibility of the AXI-to-APB interface to assign a NS bit in the AXI bus when a peripheral performs a transaction in the APB bus, and also to block reads/writes from the AXI bus with $NS = 1$ to a peripheral configured as secure.

The CPU also stores the NS bit in a special purpose register unavailable to the normal world, the Secure Configuration Register (SCR). The processor inserts the current NS bit when performing transactions on the main system bus.

The world switch mechanism is very controlled and is managed by the Secure Monitor that executes in the secure world. To switch worlds the ISA of the processor provides the SMC instruction that generates an exception, passing control to the Secure Monitor. The responsibility of the Secure Monitor is to change the NS bit that is stored in the SCR, save the processor state (registers) restore the previous world state and execute a return-from-exception instruction that restarts the execution in the restored world.

There are other hardware mechanisms that trigger the execution of the Secure Monitor from normal world. The interrupt controller of TrustZone-based processors allows that Interrupt Requests (IRQs), Fast Interrupt Requests (FIQs), external Data Abort and external Prefetch Abort exceptions are configured as secure or non-secure interrupts. Interrupts assigned as secure are, therefore, handled by the Secure Monitor.

2.11.2.C Trusted Kernels and Services

As the ARM processors are widely available in Android and iOS phones today, the ARM TrustZone design allows the development of secure applications for mobile environments, making it possible to leverage it to develop secure components of a e-voting application. Malware resistance in a voting application can be achieved if it delegates sensitive operations (such as cryptographic steps and the casting of a ballot) to the secure world, as malware in the rich operating system cannot read or write in secure world memory, nor intercept network communications issued by the secure world.

The remainder of this Section describes TrustZone-enabled systems that can be leverages for the development of a voting application.

The Android Keystore [44] is a service available in the Android operating system that stores cryptographic keys in a secure way. Once the keys are stored in the keystore, they can be used to perform cryptographic operations without the need to export the key to the application. If a TEE is available on the Android device, the key resides persistently in secure hardware, such as ARM TrustZone. Furthermore, the code that executes cryptographic operations (e.g., encrypt, decrypt, sign, etc.) is executed inside the secure hardware (e.g. inside the secure world of ARM TrustZone). In the context of e-voting applications, this service could prevent a malware-infected operating system from compromising voter's private keys and ensure the correctness of cryptographic operations.

TrustyTEE [45] is a set of software that supports TEEs on Android mobile devices . Trusty TEE consists of: 1) a secure operating system (Trusty OS) that executes in isolation, 2) drivers for the Android kernel (Linux) to facilitate the communication between trusted and untrusted components of applications, allowing them to exchange arbitrary messages between them, and 3) high level libraries for Android to facilitate access to the kernel drivers. The Trusty OS is already shipped with the Android operating system, but the development of third party trusted applications is not supported.

Requirement	Prêt À Voter	Helios	REVS	Java Card	Du-Vote	EVIV	Biomet-ric
Accuracy	Yes	**No**	Yes	Yes	Yes	Yes	Yes
Integrity	Yes	**No**	**No**	Yes	Yes	Yes	**No**
Democracy	Yes	**No**	Yes	Yes	Yes	Yes	Yes
Privacy	Yes	**No**	**No**	Yes	Yes	Yes	**No**
Verifiability	Yes	Yes	Yes	Yes	Yes	Yes	**No**
Availability	**No**	**No**	Yes	Yes	**No**	Yes	Yes
Collusion Resistance	**No**	**No**	Yes	Yes	Yes	Yes	**No**
Malware Resistance	Yes	**No**	**No**	Yes	Yes	Yes	**No**
Mobility	**No**	Yes	Yes	Yes	Yes	Yes	Yes
Usability	Yes	Yes	Yes	**No**	**No**	**No**	Yes

Table 2.1: Overview of e-voting protocols.

Trusted Language Runtime (TLR) was proposed by Santos *et. al* [46] and makes use of ARM TrustZone to provide a runtime that protects the integrity and confidentiality of .NET mobile applications from a malware-infected operating system. Within the TLR framework, the application must be split in two parts: an *untrusted component* that implements most of the application functionality and a small *trusted component* where sensitive data and computations are performed. The former runs in the normal world of ARM TrustZone and the latter is executed in the secure world of ARM TrustZone. Moreover, as device drivers have a large surface of attack and large code bases, the trusted component has no access to peripheral devices. This way, the amount of trusted code that is executed in secure world is significantly reduced. TLR uses ARM TrustZone to protect the sensitive parts of an application from malicious software on the operating system. Moreover, TLR is capable of interpreting object oriented code in the secure world, which simplifies the development and deployment processes of an e-voting application.

The concept of trust lease was introduced by Santos *et. al* [47] and is a mode of execution where constraints apply to the mobile device. This allows the temporary restriction of functionality of devices when needed. It can be denied, for example, access to the network or storage, execution of certain applications or changes to the device configuration. A trust lease must have a condition of expiration such as a time out. Furthermore, there might be the need to demonstrate to a remote third-party that the constraints are being applied correctly. The remote attestation capability can be implemented in the ARM TrustZone hardware and is able to test the integrity of the operating system, verifying if there were changes to the installed software.

2.12 Summary

Table 2.1 compares each protocol described in the previous sections against the requirements defined in Section 1.2. The trade off between malware resistance and usability is clear. Prêt À Voter (Section 2.3) is the only considered protocol that tolerates malware and is usable, but, in Prêt À Voter, the ballot is paper-based which compromises mobility.

The protocol that is closest to our requirements is EVIV (see Section 2.8). Therefore, we chose EVIV for the base of *TrustedVote*. EVIV tolerates malware, but it is missing usability. This trade off can be solved with TEEs. A TEE is a special area of the main processor that executes in isolation from the remaining of the hardware. An application can order the execution of code inside the TEE, ensuring the integrity and confidentiality of its computations while maintaining a high usability.

3

Architecture

Contents

3.1 Introduction 33
3.2 Voting process overview 33
3.3 Threat Model 34
3.4 Smartphone components 35
3.5 TrustedVote client 36
3.6 Normal World Components 38
3.7 Secure World Components 39
3.8 Bootstrap of Secure World Components 43
3.9 Threat Analysis 43
3.10 Summary 44

3.1 Introduction

This Chapter describes the design of *TrustedVote* Internet voting system. *TrustedVote* is an end-to-end verifiable Internet voting system that tolerates malware in the client computers, while keeping high levels of usability, i.e., without the need to use a device that is not widely available to the public.

As mentioned in the previous Chapter, *TrustedVote* is based on the EVIV network and cryptography protocol (see Section 2.8 and Appendix A). The EVIV system is missing usability as we defined it at Section 1.2, because in its current implementation, it requires dedicated hardware (smart card reader) to be used by the voter to successfully cast a vote. *TrustedVote* introduces an architecture of the client that can be executed in any smartphone with ARM TrustZone (see Section 2.11.2). Therefore, it does not require a separate tamper proof device to cast a vote with high level of security, as it leverages the isolation features provided by ARM TrustZone. This Chapter is focused only on the architecture of the client, because the remaining of the protocol remains unchanged. The details of the network and cryptography protocols can be found in Appendix A.

The remainder of this Chapter is organized as follows. The overview of the voting process of *TrustedVote* is given in Section 3.2. Section 3.3 describes the threat model, i.e., the assets and the possible attacks on *TrustedVote*. Section 3.4 describes the functionality of the voter's client, the smartphone. The details of the *TrustedVote* client are discussed in Section 3.5. Section 3.6 describes, in detail, the components that are not secure, while Section 3.7 provides details on the components that must be executed securely. Section 3.8 explains how the system must be bootstrapped in order to give guarantees to the voters that their devices are running *TrustedVote* in the secure world. Section 3.9 describes the assumptions and threat analysis. Finally, Section 3.10 summarizes this Chapter.

3.2 Voting process overview

The global overview of the voting process of *TrustedVote* is illustrated in Figure 3.1. *TrustedVote* replaces the voter's PC and voter's VST in the EVIV's voting process with a smartphone, which adds mobility to the voting system.

Before the election period (e.g., a few weeks), in the Voter Enrollment Phase, the voter identifies himself (in person) to the Enrollment Service. He brings his smartphone, that generates an asymmetric key pair. The Enrollment Service is responsible for the registration of the voters for the elections. This service extracts the public part of the generated key pair, associates it with the voter's identity and he can now use its smartphone to vote in the subsequent elections.

After the enrollment, the Election Registration Phase begins. The smartphone creates and shows a code card to the voter. A code card is an association between candidates and small random strings (vote codes) and a single confirmation code. The smartphone creates an empty ballot and sends it to

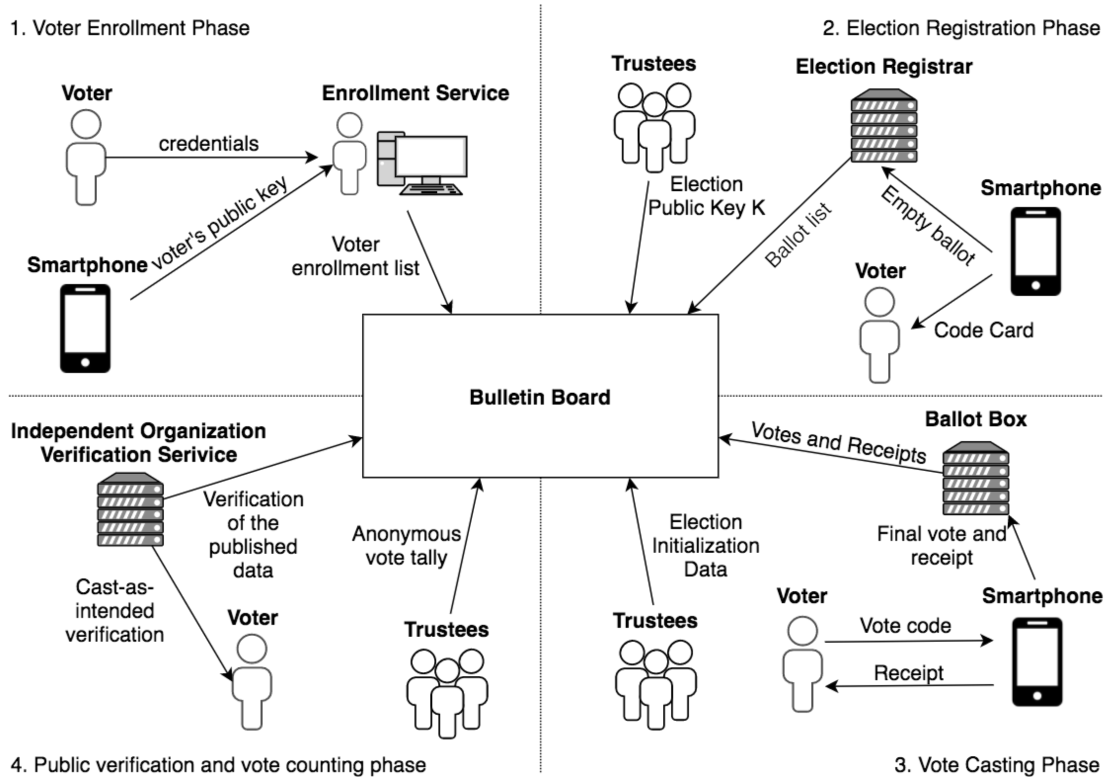

Figure 3.1: Overview of the *TrustedVote* protocol phases. The first phase is at the left upper corner, proceeding clockwise.

the Election Registrar. The ballot is signed with the voter's private key for authentication purposes.

During the election period (Vote Casting Phase) the voter is able to use his smartphone to cast his vote. To do so, he uses his code card and selects the vote code that corresponds to the candidate he wants to vote. The smartphone generates the final vote and the final receipt. A receipt is an association between candidates and confirmation codes. The voter confirms the receipt, verifying if the chosen candidate has the code card's confirmation code. The final vote is submitted to the Ballot Box and posted at the public Bulletin Board. The vote is counted in the final tally during the Public verification and vote counting phase.

3.3 Threat Model

This Section describes the threat model that we consider for *TrustedVote*, i.e., the assets and the possible attacks that can be potentially executed against *TrustedVote*. Regarding the smartphone used by the voter, an attacker is interested in the vote sent to the election servers, the code card used to translate a vote code into a valid candidate, or the voter's private key that uses to authenticate himself.

A possible attack, that compromises integrity of the election, is when the attacker uses malware to

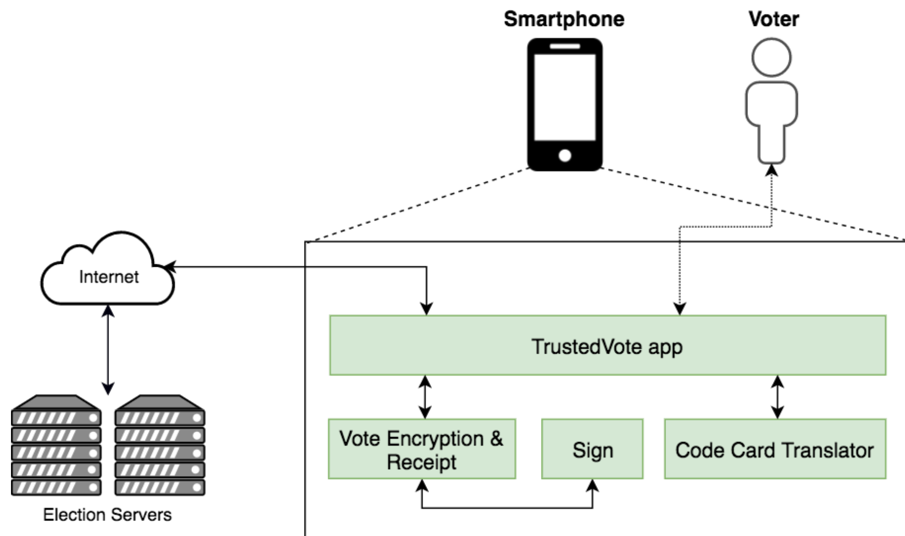

Figure 3.2: *TrustedVote* client components

steal the voter's private key from the memory of the smartphone. He can compromise the integrity of the election because he can modify the vote according to his intentions, and generate a valid signature that will be accepted by the election servers.

If an attacker controls the smartphone and is capable of reading the code card from its memory, then the privacy of the election is compromised. He is capable of: 1) intercepting the vote code used by the voter and translating it to the chosen candidate (privacy), and 2) change the vote code issued by the voter to a valid vote code of his choice (integrity). An attacker can try to modify the software that executes in the client that encrypts the vote and authenticates the user, also compromising integrity and privacy of the vote.

The attacker can also modify the contents of the vote before being encrypted, if he has full access to the memory of the smartphone. This way, integrity is also compromised.

An attacker that controls the smartphone can also block the voter from voting (denial of service in the client), which compromises the availability of the voting system to the voter.

An attacker (with control of the network) can also attempt to eavesdrop or modify the vote while it is on the network. This attack can also compromise the privacy and the integrity of the election.

3.4 Smartphone components

This Section focuses and explains the functionality of the smartphone that is represented in Figure 3.1.

During the Election Registration phase, the smartphone must create and show the code card to the voter, and create an empty ballot to send to the Election Registrar. Moreover, during the Vote

Casting Phase, the smartphone is also responsible for capturing the intention of the voter, create the corresponding digital vote and send it to the Ballot Box to be tallied. As a consequence, the device must have a user interface so that the voter can choose the desired candidate, has a network interface and has cryptographic capabilities to protect the vote from being leaked (privacy) or being modified (integrity).

Figure 3.2 provides more detail on the components and functionality of the smartphone. The smartphone functionality is divided in four main components (represented in green in Figure 3.2): the *TrustedVote* app, the Vote Encryption & Receipt, the Sign and the Code Card Translator. The *TrustedVote* app provides the user interface and is responsible for communicating with the election servers (Election Registrar and Ballot Box) with the network protocol of EVIV. Vote Encryption & Receipt is a component that encrypts the vote so that it can remain private, and computes receipts so that the voter has guarantees that the vote expresses his intentions. This component depends on the Sign component, that manages the key pair of the voter and generates digital signatures of votes. Finally, the Code Card Translator component is responsible for creating a code card for each election and for translating the vote code provided by the user into a candidate.

The Code Card Translator is an important component of the client. It contains the code card structure that allows for the translation of a vote code to a candidate. We want to prevent attackers to have knowledge of the code card or to modify it. If the attacker has read or write access to the region of memory where code card is located, then he can control the final vote.

A similar line of thought is valid for the Vote Encryption & Receipt and Sign components. If the attacker has read or write access to the region of memory where the plaintext vote is stored, then he can control the final vote. Also, if the attacker has access to the region of memory where the private key of the voter is stored, then he can impersonate him.

Thus, the components Code Card Translator, Vote Encryption & Receipt and Sign are critical for the privacy and integrity of the vote, and, for that reason, we consider them sensitive components. The next Section describes how the sensitive components are protected from the reach of the attacker in *TrustedVote*.

3.5 TrustedVote client

In order to protect reads and writes on the vote from the attacker, the sensitive components of *TrustedVote* (Code Card Translator, Vote Encryption & Receipt and Sign as discussed in the last Section) must be executed outside of the attacker's control.

The solution is to use a smartphone that has ARM TrustZone hardware. The architecture of the *TrustedVote* client is illustrated in Figure 3.3. In this architecture, there are two distinct software stacks

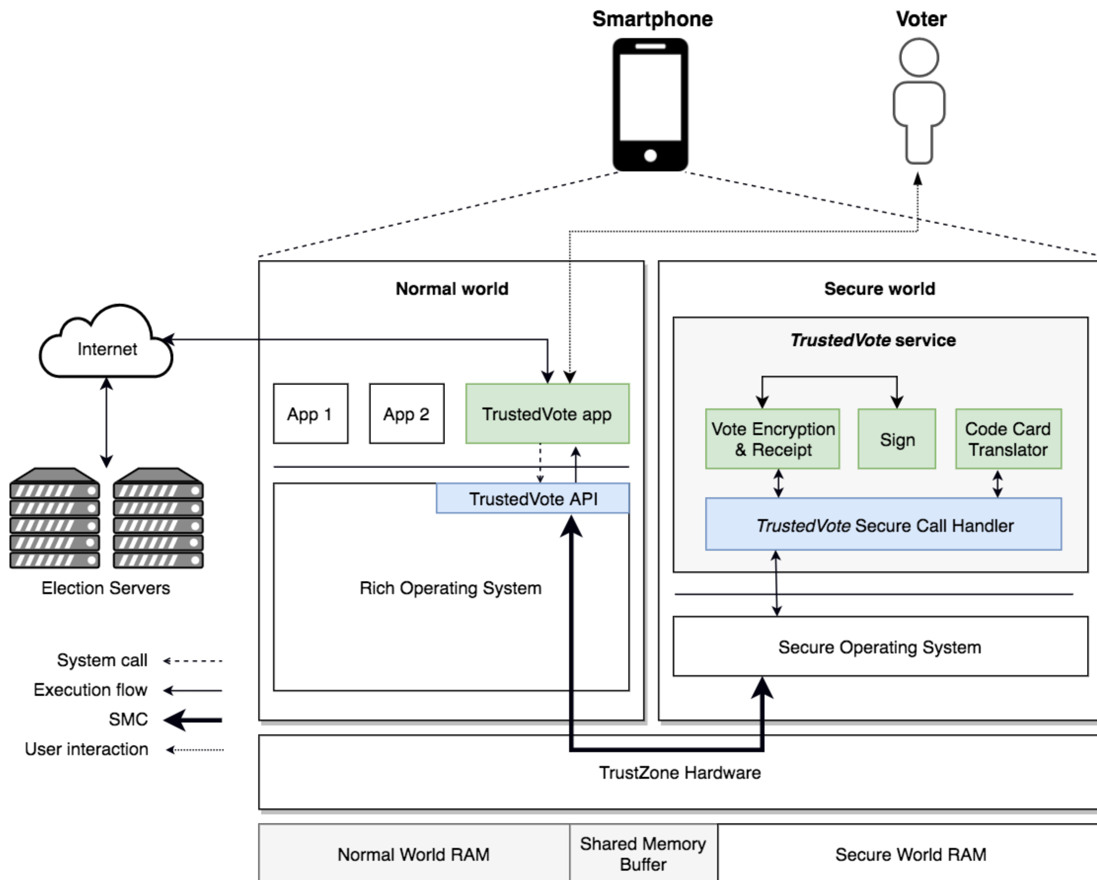

Figure 3.3: *TrustedVote* client application architecture.

running over the ARM processor: the software running in the normal world and the software running in the secure world of ARM TrustZone. The component of the voting client that executes in the normal world is the *TrustedVote* app, while the *TrustedVote* service executes in the secure world. The *TrustedVote* service contains the sensitive components: Vote Encryption & Receipt, Code Card Translator and Sign as they are sensitive components.

These components are supported by two distinct operating systems: a Rich Operating System running in normal world, and a Secure Operating System running in secure world. The Secure Operating System is a microkernel that is significantly smaller than the Rich Operating System, as it only implements the required functionality to support the execution of *TrustedVote* service in the secure world, i.e., dynamic memory management, basic scheduling and abstractions to write on persistent memory.

Each world of execution has distinct RAM spaces. *TrustedVote* app only has access to normal world RAM, while Vote Encryption & Receipt, Sign and Code Card Translator have access to secure world RAM. During the process of voting, it is necessary that the vote created in the secure world by the Vote Encryption & Receipt component reaches the normal world so that it can be sent to the election servers. Therefore, in the architecture of *TrustedVote* there is a Shared Memory Buffer that is accessible by

both worlds of execution. The *TrustedVote* Secure Call Handler and the *TrustedVote* API (represented in blue in Figure 3.3) are auxiliary components that manage the Shared Memory Buffer and perform reads/writes on it. The *TrustedVote* API provides the interface that the *TrustedVote* app must invoke in order to delegate execution to the sensitive components executing in the secure world.

3.6 Normal World Components

The rich operating system (e.g., Android) is executed in the normal world. The *TrustedVote* app is an application running in user-mode on top of rich operating system. This normal world client leverages the functionalities of the rich operating system, such as implementations of the network stack and user interface, to: 1) mediate the communications between the *TrustedVote* service and the election servers, 2) provide a user interface to the user, and 3) accept the inputs (e.g., the vote code) from the voter. *TrustedVote* app executes the following steps (in order) through out the election process:

1. During the Election Registration Phase:
   (a) Retrieve the election candidates list CL and its signature from the election servers. Send CL and its signature to the secure world using the *TrustedVote* API available in the rich operating system.
   (b) Retrieve the election key K and its signature from the Bulletin Board. Send K and its signature to the secure world using the *TrustedVote* API.
   (c) When the voter requests, use the *TrustedVote* API to generate and show the code card to the user in the secure world as defined in Appendix A.
   (d) Use the *TrustedVote* API to request that the secure world computes and returns the empty ballot (as described in Appendix A). *TrustedVote* app sends the empty ballot directly to the election servers.
2. During the vote casting phase:
   (a) Accept the vote code from the voter.
   (b) Send the vote code to the secure world through the *TrustedVote* API. The secure world computes the final vote and returns it.
   (c) Receive the final vote from the secure world, and send it directly to the election servers.

The *TrustedVote* API is a component that provides an interface to the *TrustedVote* app to delegate execution to the secure world. The responsibilities of this API are: 1) allocate the Shared Memory Buffer (Figure 3.3) inside the normal world RAM region, 2) copy data from and to the Shared Memory Buffer,

and 3) perform world switches, i.e. execute the SMC instruction. This component executes in kernel mode. This is because, according to the ARM TrustZone specification [6], the SMC instruction used to switch worlds can only be executed inside the kernel.

The Shared Memory Buffer is allocated in the normal world because the rich operating system is responsible for the management of the normal world memory and knows which regions of Random Access Memory (RAM) are free for allocation. There would be a risk of overwriting allocated memory in the case where the secure world allocates the Shared Memory Buffer. As a consequence, after the allocation of the Shared Memory Buffer, the *TrustedVote* API copies the physical location of the Shared Memory Buffer to a location known by both worlds of execution.

The *TrustedVote* API is also responsible for: 1) copying user space data from the *TrustedVote* app to the Shared Memory Buffer when there are inputs to the operations implemented in the secure world, and 2) copying data returned by the *TrustedVote* service from the Shared Memory Buffer to the user memory space of *TrustedVote* app.

3.7 Secure World Components

TrustedVote service is executed in the secure world and implements the functionality that manipulates directly the code card during the Election Registration Phase (see Figure 3.1), the creation of the empty ballot in the Election Registration Phase, the final vote during the Vote Casting Phase and the voter's private key to digitally sign the empty ballot and final vote. The components that execute in the secure world are the Vote Encryption & Receipt, Sign, the Code Card Translator and the *TrustedVote* Secure Call Handler. This Section describes the interactions between these components and the data structures that each one requires.

The *TrustedVote* Secure Call Handler is the entry point of the *TrustedVote* service that implements a set of secure calls, allowing *TrustedVote* app to request execution of the components Vote Encryption & Receipt, Sign and Code Card Translator. Next, we describe the secure calls provided by the *TrustedVote* Secure Call Handler and are exposed to the normal world:

1. During the Election Registration Phase:

   (a) `RECEIVE_CANDIDATES_CALL`: $(candidate_list, (candidate_list)_{K_{pEC}}) \rightarrow ()$

   (b) `RECEIVE_ELECTION_PARAMETERS_CALL`: $(election_key, (election_key)_{K_{pEC}}) \rightarrow ()$

   The purpose of the first two secure calls is only to transfer public election data to the secure world memory. `RECEIVE_CANDIDATES_CALL` secure call takes the candidate list returned by the Election Registrar as input and stores it on secure memory. `RECEIVE_ELECTION_PARAMETERS_CALL` receives the election public key K and its signature. The signature is verified before storing

K. These two secure calls must be first ones to be executed, as the remaining calls depend on the candidate list and the election public key K.

The signatures of all passed arguments are verified using the appropriate public key. If any signature is not valid, the execution is aborted.

(c) `PREPARE_BALLOT_CALL`: $() \rightarrow (ballot, (ballot)_{K_{p\mathcal{V}}})$

Secure call that creates an empty *TrustedVote* ballot during the election setup phase. The ballot is encrypted by the Vote Encryption & Receipt component with the election public key K, and signed with the voter's private key by the Sign component. The secure call returns the ballot and its signature.

(d) `GENERATE_CODE_CARD_CALL`: $() \rightarrow (code_card)$

Secure call that generates a code card for the election. The code card is generated by the Code Card Translator component. The code card is stored in secure memory address space and shown to the voter prior to the election. This secure call can only be executed once per election and during the election registration phase.

2. During the Vote Casting Phase:

(a) `SELECT_CANDIDATE_CALL`: $(vote_code) \rightarrow (receipt)$

Secure call that only executes during the election period, and accepts as input the vote code of the chosen candidate. Using the previously generated code card, the vote code is translated into a candidate and the vote encryptions (in the ballot generated at `PREPARE_BALLOT_CALL`) are rotated until the $YESvote$ is aligned with the chosen candidate. For each candidate, a receipt ϑ is computed and shown to the voter for confirmation.

At the end of this call, the final vote is computed and ready for submission.

(b) `OK_SUBMIT_CALL`: $() \rightarrow (vote)$

After the voter has confirmed that the confirmation code on his code card is aligned with the chosen candidate in the receipt, the final vote stored in secure memory must be sent to the Ballot Box. Therefore, `OK_SUBMIT_CALL` returns the final vote produced by `SELECT_CANDIDATE_CALL` to the normal world.

Vote Encryption & Receipt component defines how the vote is encrypted and validated such that the voter can verify, with high probability, that its vote correspond to its choice. This component residing in the secure world implements the following functions: Vote Encryption ($\mathcal{VE}_h$) encrypts the vote with the election public key and Receipt Creation ($\mathcal{RC}_h$) that creates a receipt that allows the voter to confirm if the vote matches his intentions. The ballot data structure is created and defined inside this component,

using $\mathcal{VE}_h$ and $\mathcal{RC}_h$ functions. A ballot is a data structure that stores vote encryptions, the corresponding vote validities and receipts for each candidate.

The Code Card Translator is a component that creates the code card structure. The code card associates each candidate with a random string. The voter uses the random strings on his code card to identify the candidate that he wishes to vote. So, the interface provided by this component is the following:

- `CREATE_CODE_CARD`: $(candidates_list) \rightarrow ()$

  This function receives as input the candidates list and assigns a random string (vote code) for each of them. This mapping is stored in secure memory.

- `GET_CANDIDATE_FROM_CODE`: $(vote_code) \rightarrow (candidate)$

  This function translates a vote code into a candidate, returning the candidate. It returns an error if the vote code does not correspond to a candidate.

The Sign component is responsible for signing ballots with the voter's private key $K_{p\mathcal{V}}$. It creates the voter's key pair $(K_{\mathcal{V}}, K_{p\mathcal{V}})$ once, and stores it in secure persistent memory. The interface provided by this component is the following:

- `SIGN`: $(ballot) \rightarrow ((ballot_{K_{p\mathcal{V}}}))$

  This function receives a ballot as an parameter and returns its signature.

- `GET_PUBLIC_KEY`: $() \rightarrow (K_{\mathcal{V}})$

  This function returns the public portion of the voter's asymmetric key. It is called in the Voter Enrollment Phase, when the voter registers himself (in person) to the Enrollment Service.

3.7.1 Secure Operating System

The architecture of the operating system that runs in the secure world is a design decision that has a major impact on the size of the TCB and the overall security of the system. There are three alternatives regarding secure world architectures that were proposed by the ARM TrustZone specification [6]: a dedicated operating system, a synchronous library and a hybrid of the previous approaches. Each one has their own set of advantages and drawbacks.

The first one is to have a dedicated operating system running in the secure world, allowing multiple secure applications to run in parallel with independent memory spaces, and dynamic loading of downloaded secure applications. This is the most complex and powerful architecture, but it implements more functionalities than needed, which results in TCB that is too large.

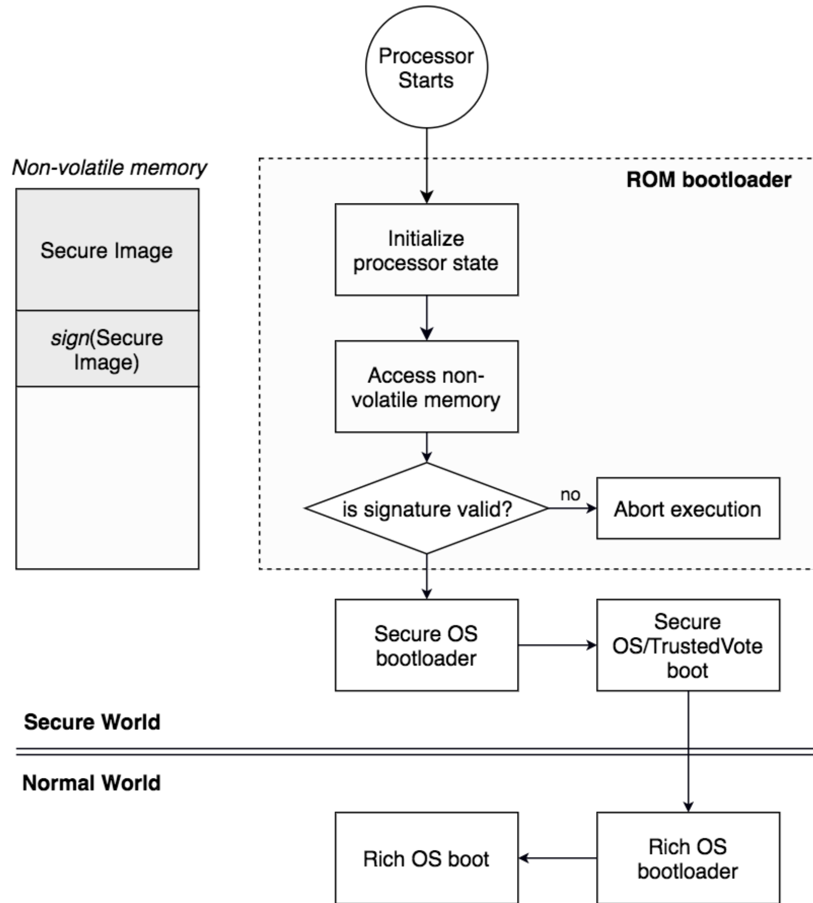

Figure 3.4: Secure boot process

The second approach is to have a synchronous library running in the secure world. This approach has the potential to reduce the TCB size, but also has the potential to require the reimplementation of components that are already implemented in the operating systems, such as dynamic memory allocation and timer services for random number generation.

The last approach proposed is an intermediate option between the dedicated operating system and the synchronous library. We opted for the intermediate option, because it has the advantage of reducing significantly the size of the TCB, while providing basic implementation of operating systems functions. Therefore, we designed the secure world where the operating system is a micro kernel that implements basic memory management, such as dynamic memory, scheduling and abstractions to write on persistent memory.

3.8 Bootstrap of Secure World Components

In order to execute the security critical code on the mobile device, the trusted vendor of the device writes the secure image that contains the secure operating system and the *TrustedVote* service code into non volatile memory. To prevent an attacker from modifying the code that runs in the secure world, we need a boot process that verifies its integrity.

To prevent that an attacker manipulates the code running in secure world, integrity checks must be performed in the boot process. Through signatures, we can guarantee the integrity and authenticity of the code. Therefore, we trust that the vendor of the devices generates a key pair $K_i = (K_{iPub}, K_{iPriv})$ and inserts its public component K_{iPub} in the hardware. The key is inserted in a way that the attacker cannot modify it, e.g., inside a special cryptographic chip. Furthermore, the vendor signs the secure image with K_{iPriv}, copies the image and the computed signature to non volatile memory.

Figure 3.4 demonstrates an overview of the secure boot process. When the system boots, the processor starts in secure world and the Read Only Memory (ROM) bootloader is executed. This component starts by setting the initial state of the processor and initializing hardware components such as the memory controller. Then, its responsibility is to access the non volatile memory and verify the signature of the image. If the signature is not valid, then the execution is aborted. Otherwise, the ROM bootloader loads the secure image into RAM and jumps to the secure operating system bootloader. The secure operating system bootloader boots the secure operating system, and, finally, normal world is activated where the rich operating system is booted.

This way, the system is initialized with the guarantee that the *TrustedVote* service is actually running in the secure world and the vote is actually processed and created by *TrustedVote* service.

3.9 Threat Analysis

TrustedVote protects the vote from being revealed or changed by malicious third parties. Section 3.3 described possible attacks on *TrustedVote* voting system, and, in this Section we describe how those attacks are mitigated in our solution.

In order to understand our threat analysis, it is important to consider our assumptions. On the smartphone, we assume a powerful adversary that is able to control the entire software stack that is running in the normal world of ARM TrustZone, i.e., the rich operating system and other applications. The attacker can, consequently, manipulate the memory and peripherals assigned to the normal world and intercept any computation performed while the processor is executing in normal world.

Regarding the hardware, we assume that it behaves correctly, i.e., the isolation provided by the ARM TrustZone hardware is implemented correctly. Also, we assume that the ROM bootloader cannot be tampered. Consequently, hardware attacks, side-channel attacks and other physical attacks are

not taken into consideration on the threat model, e.g., hardware disassembly, power monitoring, bus monitoring and timing attacks. Attacks to the secure operating system is also out of the scope of this dissertation.

We also assume that the attacker has the ability to sniff network communications between clients and servers, and execute arbitrary code on election servers except Trustee servers (see Section A.2).

In *TrustedVote*, the valueable assets to the attacker that reside on the smartphone memory (the vote, the code card and the voter's private key) are protected by the secure world of ARM TrustZone. These assets are stored in a memory region allocated only for the secure world, outside of the attacker's reach. Therefore, even if the attacker has full control of the rich operating system (normal world), he cannot read/write the vote, the code card or the voter's private key.

The attacker can perform denial of service in the smartphone, as he can: 1) ignore the calls to the *TrustedVote API*, and 2) prevent operations on the Shared Memory Buffer.

The attacker cannot modify the software running in the secure world. If an attacker modifies the secure world image, the ROM bootloader detects this (at boot time) because the signature generated by the trusted vendor would not be correct (see Section 3.8)

3.10 Summary

This Chapter presents the architecture of *TrustedVote*, an Internet voting system that allows voters to vote anywhere with high level of security. We start by describing a global overview of the voting process of *TrustedVote*. Then, we describe the assets and possible attacks on *TrustedVote*. We describe the functionality of the voter's client, the smartphone and the details of the *TrustedVote* client, that uses ARM TrustZone. We describe the details of the secure world components and normal world components, and describe how the hardware bootstraps the system to give guarantees to the voters that their devices are running *TrustedVote* in the secure world. We conclude this Chapter by discussing which of the possible attacks on the threat model are feasible.

4

Implementation

Contents

4.1 Introduction 46
4.2 Overview 46
4.3 Normal World components implementation 49
4.4 Secure World components implementation 51
4.5 Summary 53

4.1 Introduction

This Chapter describes the implemented prototype of *TrustedVote*, an Internet voting system that allows voters to vote anywhere with a high level of security. This description focuses primarily on the implementation of the client of the voting system, i.e., the smartphone.

The prototype of the client is implemented in the i.MX53 Quick Start Board from Freescale, because it is one of the few available devices with ARM TrustZone that has the technology unlocked. While the ARM TrustZone-enabled processors, such as the Cortex A57 and Cortex A53, are predominant in the mobile phone market, the vendors impose restrictions on the usage of TrustZone for development purposes. In some boards (e.g. Samsung, TI, NVIDIA) the technology is not available because the ROM bootloader switches to normal world. Moreover, there is no standard regarding the implementation of TrustZone mechanisms, which results in some boards that have incomplete TrustZone implementations. For instance, in the Raspberry Pi 3 [48] is not possible to assign protected memory regions to the secure world, because it does not implement any kind of TZASC. This limitation results in an implementation that is dependent on this particular board.

The Linux kernel runs as the operating system in the normal world, because it is the operating system that Android is based on. We implemented a command line tool that contains the functionality of the *TrustedVote* app. The *TrustedVote* API was implemented as system call for the Linux operating system. In the secure world, we modified the Genode base-hw microkernel that ships with the Genode framework to implement the *TrustedVote* service on top of it. In our prototype, we did not implement the secure bootstrap mechanism as described in Section 3.8. For development, the disk image, containing the secure code and the Linux kernel, is flashed in a SD Card. Then, the i.MX53 Quick Start Board loads the u-boot bootloader, which in turn loads and boots the disk image.

This Chapter is organized as follows: Section 4.2 describes the overview of the implementation. Section 4.3 describes the implementation of the components that execute in the normal world of ARM TrustZone, while Section 4.4 describes the implementation details of the components that execute in the secure world.

4.2 Overview

Figure 4.2 illustrates the implementation of the *TrustedVote* client prototype. This Figure maps directly to the Figure 3.3, that overviews the architecture of *TrustedVote* client.

In the normal world, the rich operating system is the Linux kernel, because: 1) it is the base of Android mobile operating system that executes in the majority of the smartphones, and 2) its source is public, and therefore, we can modify it to include the TrustedVote system call. The TrustedVote system call is our implementation of the *TrustedVote* API. The *TrustedVote* command line tool is a user process

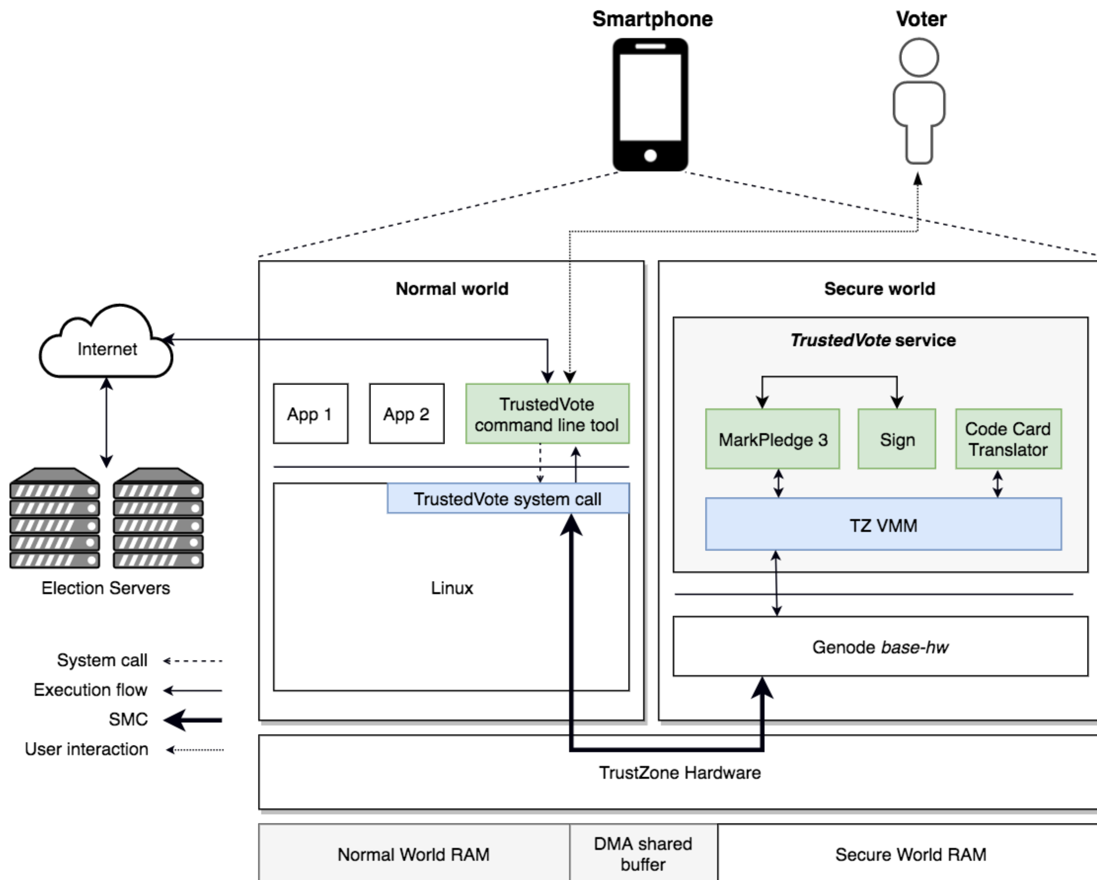

Figure 4.1: Implementation overview of the *TrustedVote* client

of Linux that implements the functionality of *TrustedVote* app.

Developing a custom kernel to run in the secure world is a time consuming task and requires that the developer has full knowledge of low-level details of the target platform. By adapting an existing microkernel, we can focus on the development of the desired functionality of the *TrustedVote* service. Our choice for the secure microkernel that executes in secure world is the Genode *base-hw* microkernel. This is because Genode developers provide: 1) extensive documentation of the framework, 2) support by an active community and 3) working demonstrations of ARM TrustZone in the i.MX53 Quick Start Board.

The Genode *base-hw* microkernel implements the base functionalities of an operating system, such as memory management and scheduling. This microkernel is designed to run on bare metal, i.e., without the need of an underlying operating system, and, therefore, can be executed in the i.MX53 Quick Start Board. The configuration of *base-hw* includes an application running on top of the microkernel that: 1) is able to manage a rich operating system running in the normal world and 2) contains a Secure Monitor implementation that handles world switches. This application is called TrustZone Virtual Machine Monitor (TZ VMM), because it is able to manage a rich operating system as a Virtual Machine Monitor (VMM)

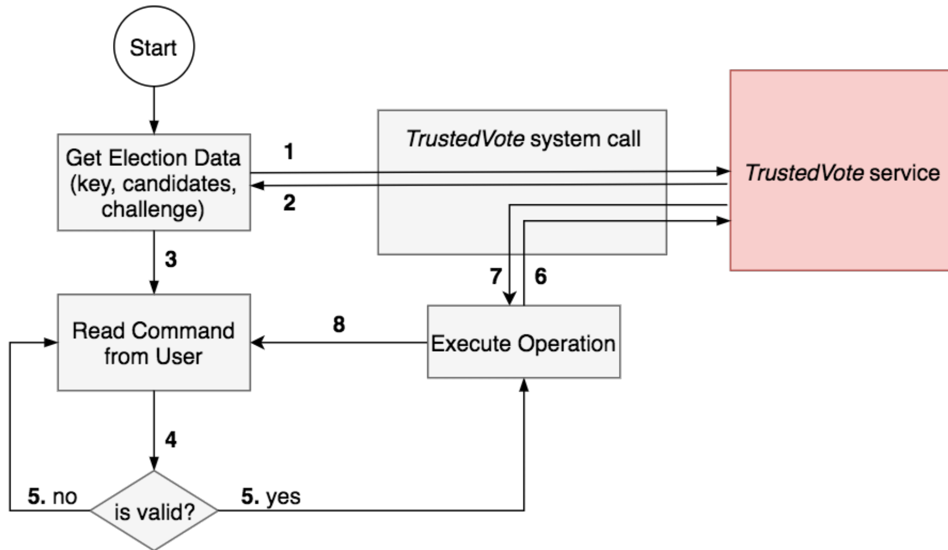

Figure 4.2: Workflow of the *TrustedVote* app

also does. Therefore, Genode *base-hw* has a low TCB, high flexibility and support for the i.MX53 Quick Start Board, which makes it attractive as a starting point for our prototype implementation. We modified the TZ VMM application to perform the role of the *TrustedVote* Secure Call Handler component.

In our implementation, each secure call is uniquely identified by an integer (called operation identifier) that is assigned in the source code of *TrustedVote* app and TZ VMM. Therefore, to invoke an operation in *TrustedVote* service, the *TrustedVote* app must pass the operation identifier as an argument to the *TrustedVote* system call.

In the prototype, the Vote Encryption & Receipt component is implemented using MarkPledge 3 technique (see Appendix A). As MarkPledge 3 depends on the exponential ElGamal cryptosystem, it also contains a software implementation of ElGamal and exponential ElGamal (see Section 2.2.1). To implement the Sign component we use the RSA cryptosystem. The Code Card Translator component is implemented as a map between a candidate's name and a random string.

Our implementation of the Shared Memory Buffer uses DMA buffers that are allocated by the Linux kernel inside the *TrustedVote* system call. The buffer is uncached, which means that bytes are written directly in the DMA shared buffer region (with the guarantee that they are flushed, i.e., does not stay in caches).

The following Sections provide further details on the implementation of each described components.

4.3 Normal World components implementation

4.3.1 TrustedVote app

The client application running in normal world is implemented in C++ and is a command line tool. The purpose of this application is to simulate a mobile application that can leverage the *TrustedVote* service that executes in the secure world to create, in a secure environment, a voter's empty ballot and final ballot as in MarkPledge 3 specification (see Appendix A). Figure 4.2 represents the execution flow of the *TrustedVote* app that executes as a user process over the Linux kernel.

The first step of this application is to gather election data (public key, election challenge as required by MarkPledge 3 and list of candidates) from the election servers. Then, in step 2, it invokes the *TrustedVote* system call to send the retrieved data to the *TrustedVote* service running in the secure world.

Then, the application is now able to accept commands from the voter. Once it receives a command from the voter, it verifies if it is a valid one. If the command is valid, the corresponding operation is executed (step 5. yes). The *TrustedVote* system call is invoked whenever execution of the *TrustedVote* service is required (step 6). The execution is restored when the latter returns (step 7) and the application is ready (again) to read a command from the user (step 8).

The valid commands accepted by *TrustedVote* app are the *prepare* command, the *codecard* command and the *vote* command. The *prepare* registers the voter in the upcoming election and uses *TrustedVote* service to create the empty ballot in the secure world. The *codecard* command invokes *TrustedVote* service to compute and show the code card for the upcoming election (Election Registration Phase). The *vote* command receives a vote code as an argument, and sends it to the *TrustedVote* service in order to compute and encrypt the final vote. The final vote is sent, via the Internet, to the election servers.

4.3.2 TrustedVote system call

In order to implement the *TrustedVote* API, we need a way to request kernel services. System calls are the most common way of requesting kernel services. Therefore, we added a new system call, called *smcall*, to the Linux kernel, that implements the functionality of *TrustedVote* API.

The *smcall* system call takes four parameters: `int op`, `unsigned int arg`, `int size` and `char* return_address`. The first parameter, `int op` represents the unique operation identifier (as explained in Section 4.2) that is used by the secure world to identify which secure call must be executed.

The following parameters are used to enable inter-world communication. The *arg* parameter points to a memory location (in *TrustedVote* app address space) that must be copied to the Shared Memory Buffer. The *size* parameter indicates the number of bytes to allocate in the Shared Memory Buffer. As the Shared Memory Buffer is allocated in the kernel (see Section 3.6), the *TrustedVote* app does not

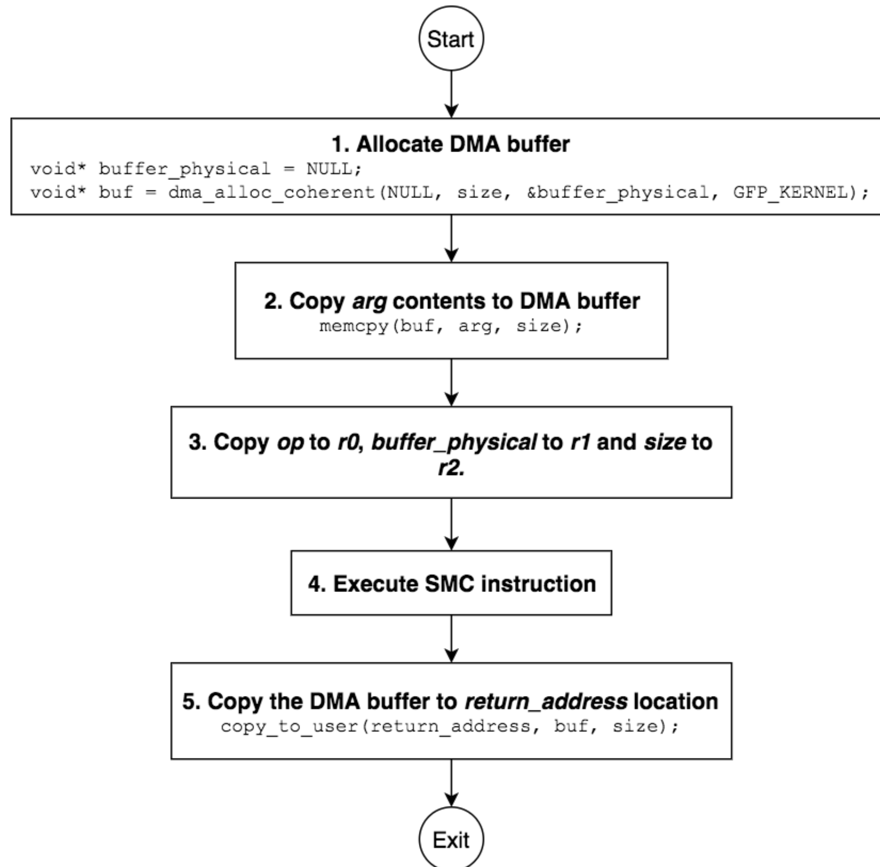

Figure 4.3: Execution flow of *TrustedVote smcall* system call.

have a way to read the contents in the Shared Memory Buffer in its memory space. The *return_address* parameter is a user space memory location (of *TrustedVote* app), that the kernel copies the content of the Shared Memory Buffer after the secure world returns.

Figure 4.3 illustrates the flow of execution of *TrustedVote smcall* system call. Step 1 is to allocate an uncached DMA buffer, available to the kernel via the `dma_alloc_coherent` function. After the `dma_alloc_coherent` function returns, the variable *buf* points to the virtual address of the DMA buffer, while *buffer_physical* points to the corresponding physical address.

The second step is to copy (via memcpy) the contents in the memory region defined by the parameters *arg* and *size* to the allocated DMA buffer.

The system call proceeds, in step 3, to copy *op* parameter to the register $r0$, the variable *buffer_physical* to the register $r1$, and the variable *size* to the register $r2$. At this point, *smcall* executes the SMC instruction.

When the secure world returns, the contents of the DMA buffer are copied to the address space of *TrustedVote* app (to the *return_address* argument), via the `copy_to_user` function available to the Linux kernel.

4.4 Secure World components implementation

4.4.1 TZ VMM

The TZ VMM component (already implemented by Genode developers), was modified in order to have the functionality provided by the *TrustedVote* Secure Call Handler (see Section 3.7).

TZ VMM is an application running on top of Genode *base-hw* kernel that 1) configures the regions of memory and peripherals that belong to the secure and normal worlds, 2) loads the Linux kernel in the normal world and 3) provides a world switch routine that is executes whenever a SMC instruction is executed in the normal world.

The TZ VMM application is responsible to configure the TrustZone hardware to assign secure RAM regions to the secure world. In the particular case of the i.MX53 Quick Start Board, the Multi Master Multi-Memory Interface (M4IF) is the hardware component that controls memory accesses from the CPU and other masters on the main AXI system bus (see Section 2.11.2). This hardware component supports the creation of a protected memory region only accessible to the secure world. According to the specification of the i.MX53 Quick Start Board, the protected memory region must be continuous, a multiple of 4 KB and cannot be greater than 256 MB. The size of the secure memory region allocated by the Genode developers is 256 MB, and this value remains unchanged for the implementation of our prototype.

The TZ VMM application also configures the TrustZone hardware to assign peripherals to either secure or normal world. The Central Security Unit (CSU) is the hardware component of i.MX53 Quick Start Board that assigns devices groups for each world of execution. We assign the Universal Asynchronous Receiver/Transmitter (UART) module (that is used for serial communication) to the normal world. This is because, in our prototype we must be able to issue commands, through the UART, in the Linux kernel.

The TZ VMM application registers a world switch handling routine in the Genode *base-hw* kernel that is executed whenever a SMC instruction is called from the normal world (e.g., in the *TrustedVote* system call). The Genode *base-hw* microkernel catches the CPU exception generated by the SMC instruction, saves the CPU registers in secure memory, and jumps to the handling routine located inside the TZ VMM. When the TZ VMM handling routine returns, the *base-hw* microkernel restores the previously saved CPU registers and switches to normal world.

We modified the TZ VMM application to include the inter-world communication mechanism described in Section 4.3.2. As we can read the saved CPU registers (before the SMC is executed) in secure world memory, we read $r0$, $r1$ and $r2$ to retrieve *op*, the unique operation identifier, *buffer_physical*, the physical address of the DMA buffer, and *size*, the size of the DMA buffer (sent by *TrustedVote* system call as described in Section 4.3.2).

The DMA shared buffer is a continuous region of memory that the software running in both worlds

of execution use to copy data. The data structures that need to be copied to this buffer (e.g., the ballot) are complex. So, we use Javascript Object Notation (JSON) as a data-interchange format to represent complex data structures in the DMA shared buffer. We use *rapidjson* library to convert C++ objects into JSON strings, and the other way around. This consumes more memory space in the DMA shared buffer, and more time to perform the convertions, but, this also has the advantage that the code running in the secure world is less coupled to its normal world counterpart.

We also modified TZ VMM to include an implementation of the secure calls as defined in Section 3.7. In addition to these secure calls, we implement a new secure call, called set election challenge, which is required by the MarkPledge 3 technique. This secure call receives as input (in the DMA shared buffer) the election challenge, verifies its signature and stores it in secure world RAM. When the TZ VMM world-switch handling routine is executed (because an SMC instruction was executed), it executes a switch/case over the operation identifier (that can be found in register $r0$) to execute the corresponding secure call.

4.4.2 MarkPledge 3

In our prototype, MarkPledge 3 implements the functionality of the Vote Encryption & Receipt component of *TrustedVote* architecture (see Section 3.7). MarkPledge 3 (see Appendix A) is a technique that defines how a vote is encrypted such that the voter can verify with high probability that the vote encryption corresponds to the voter's choice.

MarkPledge 3 requires an implementation of exponential ElGamal cryptosystem (see Section 2.2.1.A). Therefore, we need a way to represent integers with arbitrary precision. As the i.MX53 Quick Start Board processor is 32 bits, a single memory address is not able to represent integers with 1024 bits, for instance. Also, we need a way to perform arithmetic operations on these big numbers, such as modular exponentiations. We include the OpenSSL implementation of big numbers in the secure world to represent integers with arbitrary precision. As a consequence, MarkPledge 3 component is implemented on top of OpenSSL implementation of big numbers.

4.4.3 Sign

The Sign component creates the digital signatures of the ballot and the final vote with the voter's private key (see Section 3.7). In our prototype, we use the RSA cryptosystem for the signature generation.

To implement this functionality, we included the OpenSSL RSA library in the secure world. The voter's private key is generated with the `RSA_generate_key` available in the OpenSSL RSA library, and stored in secure world RAM. In our implementation, we do not store the voter's private key in persistent memory.

$$h = sha1(ballot)$$
$$s = h^d \mod n \tag{4.1}$$

The signature computation is described in Equation 4.1. To compute the signature of a ballot, our prototype reads the bytes of the ballot, applies SHA-1 digest and computes the RSA signature of the resulting digest, where d is the private exponent of the voter's private key and n is the modulus.

4.4.4 Code Card Translator

As mentioned in Section 3.7, the Code Card Translator component generates random strings (vote codes) for each candidate of the election. Our Code Card Translator implementation iterates over the list of candidate names, and creates a libc++ map where the key is the candidate name and the value is a random string. To show the code card to the voter, this component writes the name and the corresponding vote code for each candidate to the serial console. The serial console is communication is performed through the UART.

4.5 Summary

In this Chapter, we described the details of the implementation of our prototype of *TrustedVote* Internet voting system. We started with an overview of *TrustedVote* client implementation. Then, we described the implementation of the normal world components: 1) the *TrustedVote* app, a command line tool that simulates a mobile application that leverages the *TrustedVote* service to create the vote, and 2) the *TrustedVote* system call, that performs the world-switch. Next, we described the implementation of secure world components, namely, the TZ VMM, MarkPledge 3, Sign and Code Card Translator components.

5

Evaluation

Contents

5.1 Introduction 55
5.2 Methodology 55
5.3 Performance of secure world Operations 55
5.4 Trusted Computing Base size 58
5.5 Attack Surface assessment 60
5.6 Summary 60

5.1 Introduction

The evaluation of TrustedVote is focused on the performance (availability requirement) and malware resistance of the client architecture, as the EVIV network protocol and its cryptography protocol (Mark-Pledge 3) satisfy accuracy, integrity, democracy, privacy, verifiability, availability and collusion resistance (see proof in Joaquim et. al [5]).

To evaluate the prototype, we measure different aspects of this implementation. We start by discussing the methodology used to evaluate the prototype in Section 5.2. In Section 5.3, we discuss the penalty of running the *TrustedVote* service in the secure world of ARM TrustZone and compare the execution times of *TrustedVote* service and the EVIV *smartcard* implementation. The size of the TCB is discussed in Section 5.4. We assess the attack surface in our prototype in Section 5.5. Finally, the summary of the evaluation is given in Section 5.6.

5.2 Methodology

The evaluation of the prototype was performed on the same board that we implemented the prototype: the i.MX53 Quick Start Board. This board has 1 GB of DDR3 RAM, a 1 GHz ARM Cortex-A8 processor, and has ARM TrustZone hardware available for development purposes.

In order to run the prototype, the modified Genode and Linux kernels were flashed in a mini SD card and inserted in corresponding slot of the board. The code from both normal and secure worlds was compiled with the *g++* compiler and with the O3 flag. The O3 flag enables code optimizations, resulting in improved performance, at the cost of higher compilation time.

We measure execution times using the performance counters available in ARM processors. Performance counters are special purpose registers used to store counters of hardware-related events. In particular, we read the Cycle Count (CCNT) register that counts CPU cycles. Since the processor runs at 1 GHz, it means that each CPU cycle corresponds to 1 nanosecond. To measure the execution time of an operation in milliseconds, we measure the CCNT register before and after the operation code and take the CPU cycle difference multiplied by 10^{-6}. The execution time values presented in the evaluation experiments' are the result of the average of 20 runs.

For this evaluation, we set up an ElGamal election key with 1024 bits.

5.3 Performance of secure world Operations

The goal of this analysis is to compare execution times between normal world vs secure world execution, to measure the secure world impact on the performance of the client. Using these measures,

	tvote-client	tvote-client-emulated
TrustedVote app	1 ms	1 ms
TrustedVote service secure calls	3544 ms	3068 ms
Total	3545 ms	3069 ms

Table 5.1: *TrustedVote* client execution time required to cast a valid vote in an election with 10 candidates.

we can verify if the *TrustedVote* client introduces noticeable overheads to the voter during the voting process.

To compare execution times of secure and normal worlds we set up two different binaries containing the client functionality: the *tvote-client* binary and the *tvote-client-emulated* binary. The *tvote-client* binary offers a console interface to the voter and each time a secure operation must be executed, the TrustedVote system call is called, where the required DMA buffer is allocated and the SMC is issued to trigger the switch to secure world. The execution is then delegated to the *TrustedVote* service that implements the operation. At the end, the required data is copied to the DMA shared buffer, the normal world is restored and the DMA shared buffer is copied to a buffer in normal world memory. In the *tvote-client-emulated* binary, the *TrustedVote* service runs in the normal world, avoiding the allocation of DMA buffers, the context switch triggered by the system call and the SMC instruction.

We start by measuring the total client execution time required to cast a vote in an election with 10 candidates, with *tvote-client* and *tvote-client-emulated*. We chose an election with 10 candidates because it is close to the number of candidates on real elections (e.g., the last presidential elections in Portugal had 10 candidates). Table 5.1 shows the results. The total client execution time can be divided in two main parts: the execution time of *TrustedVote* app (running in normal world) and the execution time of *TrustedVote* service operations. In the latter, we include the world-switch time in the case of the *tvote-client* binary (secure world). We define the world-switch time as the sum of the time required to perform the context-switch (from user mode to kernel mode in normal world), to execute the *TrustedVote* system call and the *SMC* instruction. Furthermore, the execution times measure the total time to cast a valid vote, but do not include network communications and time required for a voter to insert commands.

The binary *tvote-client* executes in 3545 ms, while *tvote-client-emulated* executes in 3069 ms. We notice an overhead of 476 ms (which represents an overhead of 16 %) in the case where the *TrustedVote* service executes in the secure world (*tvote-client*). However, the 476 ms of overhead when *TrustedVote* service executes in the secure world (*tvote-client* binary) are not noticeable to the voter.

5.3.1 Secure calls performance

In Table 5.1, we can see a 476 ms overhead in the execution time of *TrustedVote* service secure calls in the secure world. To understand the source of overhead, we measured the execution times of

TrustedVote secure call	tvote-client	tvote-client-emulated
Receive Election Candidates	45 ms	1 ms
Set Election Challenge	44 ms	1 ms
Set Election Key	49 ms	1 ms
Prepare ballot	3099 ms	3023 ms
Generate codecard	106 ms	2 ms
Select Candidate	97 ms	2 ms
Ok submit	104 ms	38 ms
Total	3544 ms	3068 ms

Table 5.2: *TrustedVote* service secure calls execution times comparison between secure and normal worlds, in an election with 10 candidates.

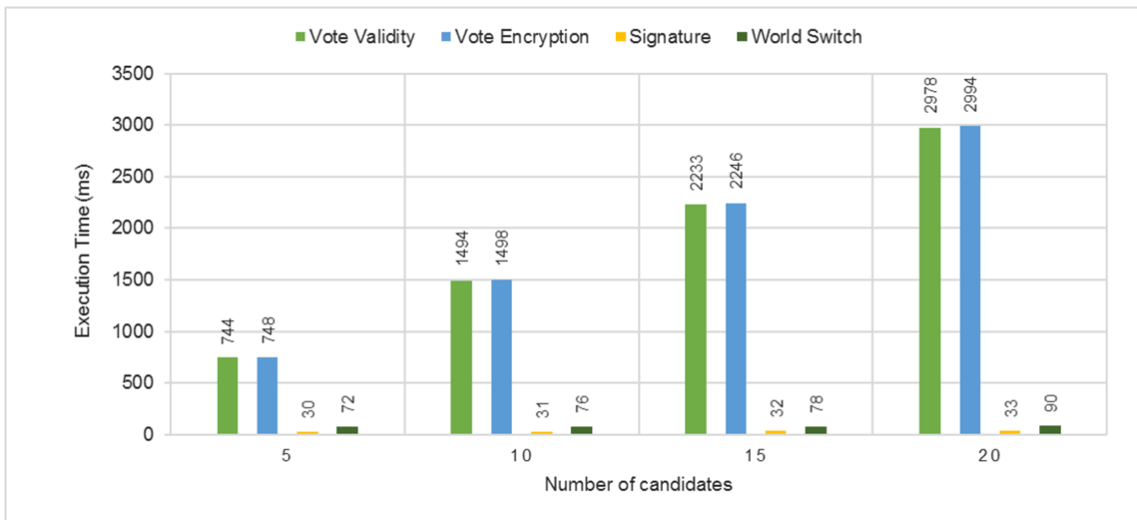

Figure 5.1: World-switch overhead in the Prepare ballot secure call.

each secure call individually, using the binaries *tvote-client* and *tvote-client-emulated*. The results are presented in Table 5.2.

From these results, we notice that: 1) the overhead is not constant across all secure calls, and 2) the largest overhead is in the prepare ballot secure call (which is also the most CPU intensive).

5.3.2 World-switch overhead

The overhead for each secure call that is present when the *tvote-client* binary executes is due to the world-switch execution time. We chose the most CPU expensive operation (the Prepare ballot secure call) to study the overhead introduced by the world-switch. This operation is also the one that allocates more space on the DMA shared buffer during the world-switch.

In Figure 5.1, we show the execution times for the Prepare ballot secure call for an election with 5,

System/Library	Code size
Linux 2.6.35	14000 KLOC
OpenSSL 1.0.1	460 KLOC
TrustedVote service	48.3 KLOC
Genode *base-hw* microkernel	20 KLOC

Table 5.3: Code size comparison of *TrustedVote* and other systems.

10, 15 and 20 candidates. These number of candidates are in the range of candidates for a national election, and, at the same time, allow us to understand the impact of the size of the election (i.e., the number of candidates) in the world-switch overhead. The number of candidates is presented on the horizontal axis, while the execution time (in milliseconds) is shown in the vertical axis. We measured the Prepare ballot secure call execution times in its individual parts: the Vote Validity time, the Vote Encryption time, the Signature time and the World-Switch time.

The execution time is dominated by the Vote Validity and Vote Encryption components. The signature execution time increases as the number of candidates increases, however it is negligible. We noticed that the percentage of world-switch time over the total time of the secure call does not remain constant. For example, for 5 candidates, $\frac{72}{1594} = 4.5\%$, while for 20 candidates, this percentage is only $\frac{90}{6092} = 1.5\%$. We can conclude that the world-switch time, as expected, has a constant cost (evident with less candidates) and a variable cost that depends on the number of candidates.

This happens because in the world-switch time we include: 1) the time of the context switch, which does not depend on the number of candidates of an election, and 2) the time for the allocation of the DMA shared buffer. The operations on the DMA shared buffer are dependent on the number of candidates. The size of a ballot grows with the number of candidates, and, as a consequence, the time to copy it to/from the DMA shared buffer increases.

5.4 Trusted Computing Base size

The code that runs in the secure world is critical for the security of the entire election. One possible way to assess the probability of bugs in a system is to measure its code base size and compare to other systems. A small code base allows meticulous code audits from third parties, and narrows the attack surface for attackers. With a narrow attack surface, developers have the chance to sanitize all the inputs from the normal world. Minimizing the TCB size is essential in any security-driven design. Therefore, we assess the TCB size of *TrustedVote* prototype and compare it to other systems and libraries.

Table 5.3 compares the code base size of different libraries and kernels. The Linux kernel is one of the most complex systems with around 14000 KLOC under the version we use on the prototype (2.6.35).

Component	Code size
Genode *base-hw* micro kernel	21 KLOC
MarkPledge 3	1.1 KLOC
TZ VMM	0.7 KLOC
Sign	0.04 KLOC
Code Card Translator	0.02 KLOC
OpenSSL BN	2.5 KLOC
OpenSSL RSA	5 KLOC
libc++ vector	7.8 KLOC
rapidjson	10.2 KLOC
***TrustedVote service* total**	48.3 KLOC

Table 5.4: Code size of the components that execute in the secure world in our prototype.

It is composed of multiple optimized subsystems that aim to support several functionalities and devices. The next library is OpenSSL, which implements the TLS and SSL protocols. The library also implements the most common cryptography algorithms. Finally, *TrustedVote* service and Genode *base-hw* have the desired code base size to run in secure world. The TCB size of *TrustedVote* service is 2.4 times greater than Genode *base-hw* microkernel. That is because the former implements the required functionality (such as MarkPledge 3, secure calls, etc.) for a secure election client. However, it is 296 times smaller than a full Linux kernel and 9.7 times smaller than OpenSSL library.

To comprehend better why the code size of *TrustedVote* service is 2.4 times greater than the code size of Genode *base-hw*, Table 5.4 discriminates the number of lines of code for each component of the former. As mentioned in Section 4.2, *TrustedVote* service is built on top of Genode *base-hw* microkernel, that provides basic scheduling, memory management and basic SMC handling. In order to transfer complex objects (such as a ballot) between worlds, we used rapidjson library to serialize those objects to JSON format. This library sums 10.2 KLOC to the TCB size. The list of candidates is stored in a *vector* object that is implemented in libc++, adding 7.8 KLOC to the TCB size. We used the RSA and big number implementation of the OpenSSL library in order to implement ElGamal cryptosystem and MarkPledge 3. The code that implements exponential MarkPledge 3 (and ElGamal), contributes with 1.1 KLOC to the final TCB size. The modified TZ VMM application represents 0.7 KLOC, while the Sign component has 0.04 KLOC and the Code Card Translator 0.02 KLOC.

Further improvements could have been made to reduce even more the code size of the TCB of *TrustedVote* service. A custom serializer of ballots would have removed the need to use the general purpose *rapidjson* library, removing 10.2 KLOC. Also, linked lists could have been implemented in the secure world to store the list of candidates, removing the libc++ *vector* from the TCB. Finally, as *OpenSSL* is a general purpose library, custom big number and RSA implementations with only the required operations could also have been implemented. However, custom implementations are more error prone than a

public library that has been tested extensively. Therefore, we decided to include them in the TCB.

5.5 Attack Surface assessment

In this Section we discuss possible attacks that could be performed on *TrustedVote* client implemented in our prototype. The *TrustedVote* service implementation has a narrow set of attack vectors, i.e. the secure calls defined in Section 4.4.1. This reduces the amount of vulnerabilities that can be exploited in order to compromise the privacy and integrity of the vote, because third party auditors (such as the political parties) can audit the code in useful time.

The first attack we consider is when malware residing in the Linux kernel prevents the access to the secure world (denial of service on the client). Malware residing in the Linux kernel can ignore the *TrustedVote* system call and the voter is not able to cast his vote. This attack is detectable by the voter, because he is not able to cast his vote. In this case, he can: 1) perform another device registration in the Election Registrar, or 2) perform a clean install of operating system on the same in order to remove the malware. Nonetheless, the integrity and confidentiality of the vote are never compromised.

The DMA buffer is used as a shared memory mechanism to read and write data, never to execute code from it. Any data modification performed by the attacker during the execution in the normal world is detected by *TrustedVote* service, as all data is signed by the election servers and the vote codes are random. The probability of malware guessing a valid vote code is $number_of_candidates/15^{\alpha}$ where α is the size of the vote code. For instance, in an election with 5 candidates and $\alpha = 5$, the probability of guessing one valid vote code is 0.000007.

Although ARM TrustZone hardware protects the memory region assigned to the secure world from being accessed by the normal world, it does not encrypt it. This way, an attacker that has access to the client hardware, p.e. through a specific device that reads the memory bus, is able to learn the contents of secure memory, such as the voter's private key. Nevertheless, hardware attacks and other side channel attacks (timing attacks, power monitoring attacks, etc.) are out of the scope of this dissertation, as mentioned in Section 3.9.

Furthermore, vulnerabilities in Genode *base-hw* code can lead to malware injection by the attacker, compromising the secure memory and, therefore, the integrity and confidentiality of the vote. Nonetheless, we do not consider attacks to Genode *base-hw*.

5.6 Summary

The EVIV network protocol and its cryptography scheme guarantee that accuracy, integrity, democracy, privacy, verifiability, robustness, availability and collusion resistance are satisfied at server side. If

malware resistance is not satisfied at client side, then the integrity and privacy of the vote cannot be guaranteed. *TrustedVote* achieves malware resistance by implementing MarkPledge 3 on the secure world of ARM TrustZone with unnoticeable performance penalties to the voter. The main performance penalty is from the world-switch that is performed in order to delegate execution from the normal to the secure world. There is a focus on having a small TCB size, which *TrustedVote* achieves, so that third party companies or entities (e.g., security companies, political parties, etc.) can afford code audits in useful time for deployment on a national election, for instance.

6

Conclusion

Contents

6.1 Conclusions 63
6.2 Future Work 63

6.1 Conclusions

The lack of mobility found on paper based voting systems (that require physical presence of voters in a specific date and local) compromise the goal of democracy. It creates unnecessary problems, such as the prohibition of sport events during election days (as debated in Portugal), and citizens may not be available to vote.

The goal of this dissertation is to propose a fully mobile Internet voting system named *TrustedVote*. With this system, the voters can vote anywhere with an Internet connection and be certain that neither the vote nor the election is compromised. The design of *TrustedVote* is similar to the EVIV protocol. However, we use ARM TrustZone to execute the sensitive operations on the voting system: the vote encryption and voter authentication. This is possible because ARM TrustZone allows the isolated execution of code in mobile devices where TrustZone is enabled. Therefore and unlike previous systems, *TrustedVote* keeps a high level of usability as the voter does not have to acquire special hardware devices to successfully cast his vote.

We implemented a prototype of *TrustedVote* client using real TrustZone hardware (on the i.MX53 Quick Start Board). We implemented MarkPledge 3 technique to encrypt the vote and RSA signatures to authenticate the voter. The security critical code of *TrustedVote* is very small, with only 48.3 KLOC under the prototype implementation, which reduces the number of vulnerabilities that can be exploited. In an election with 10 candidates, we observed that the cost of running this security critical code on the secure world to successfully cast a vote is 476 ms. This overhead is unnoticeable to the voter, when he is casting a vote.

6.2 Future Work

In the future, *TrustedVote* can be improved in five ways. The first is to find a smartphone that allows the development of applications with ARM TrustZone. The second is to implement a graphical user interface to the prototype. The third is to add support for secure display of code cards in the secure world when using a graphical user interface. The fourth is to further reduce the size of the TCB, by implementing only the required functions of big numbers and linked lists. The last one is to add coercion resistance to *TrustedVote* by, for instance, allowing voters to vote multiple times.

Bibliography

[1] P. Y. A. Ryan, D. Bismark, J. Heather, S. Schneider, and Z. Xia, “Prêt À Voter: a Voter-Verifiable Voting System,” *IEEE Transactions on Information Forensics and Security*, vol. 4, no. 4, pp. 662–673, Dec 2009.

[2] R. Joaquim, A. Zúquete, and P. Ferreira, “REVS – A Robust Electronic Voting System,” *IADIS International Journal of WWW/Internet*, vol. 1, no. i, pp. 47–63, 2003.

[3] M. Mohammadpourfard, M. A. Doostari, M. B. Ghaznavi Ghoushchi, and N. Shakiba, “A new secure Internet voting protocol using Java Card 3 technology and Java information flow concept,” *Security and Communication Networks*, vol. 8, no. 2, pp. 261–283, 2015.

[4] G. S. Grewal, M. D. Ryan, L. Chen, and M. R. Clarkson, “Du-Vote: Remote Electronic Voting with Untrusted Computers,” *Proceedings of the Computer Security Foundations Workshop*, vol. 2015-September, pp. 155–169, 2015.

[5] R. Joaquim and C. Ribeiro, *An Efficient and Highly Sound Voter Verification Technique and Its Implementation*. Springer Berlin Heidelberg, 2012, pp. 104–121.

[6] “ARM. ARM Security Technology - Building a Secure System using TrustZone Technology. ARM Technical White Paper. 2009,” http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, accessed: 2016-11-28.

[7] W. Alrodhan, A. Alturbaq, and S. Aldahlawi, “A mobile biometric-based e-voting scheme,” *2014 World Symposium on Computer Applications and Research, WSCAR 2014*, 2014.

[8] D. Petcu and D. A. Stoichescu, “A hybrid mobile biometric-based e-voting system,” in *2015 9th International Symposium on Advanced Topics in Electrical Engineering (ATEE)*, May 2015, pp. 37–42.

[9] P. Y. Ryan, “A variant of the Chaum voter-verifiable scheme,” in *Proceedings of the 2005 Workshop on Issues in the Theory of Security*. ACM, 2005, pp. 81–88.

[10] P. Y. Ryan and S. A. Schneider, “Prêt À Voter with re-encryption mixes,” in *European Symposium on Research in Computer Security*. Springer, 2006, pp. 313–326.

[11] R. Joaquim, P. Ferreira, and C. Ribeiro, “EVIV: An end-to-end verifiable Internet voting system,” *Computers & Security*, vol. 32, pp. 170–191, 2013.

[12] B. Adida, “Helios: Web-based Open-Audit Voting.” *USENIX Security Symposium*, pp. 335–348, 2008.

[13] K. Gjøsteen, “The Norwegian internet voting protocol,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7187 LNCS, pp. 1–18, 2012.

[14] D. Chaum, P. Y. A. Ryan, and S. Schneider, “A Practical Voter-verifiable Election Scheme,” in *Proceedings of the 10th European Conference on Research in Computer Security*, ser. ESORICS’05. Springer-Verlag, 2005, pp. 118–139.

[15] B. Adida and R. L. Rivest, “Scratch & Vote: Self-Contained Paper-Based Cryptographic Voting,” in *Proceedings of the 5th ACM Workshop on Privacy in Electronic Society*, ser. WPES ’06. ACM, 2006, pp. 29–40.

[16] T. Elgamal, “A public key cryptosystem and a signature scheme based on discrete logarithms,” *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, Jul 1985.

[17] D. L. Chaum, “Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms,” *Commun. ACM*, vol. 24, no. 2, pp. 84–90, Feb. 1981.

[18] R. Aditya, K. Peng, C. Boyd, E. Dawson, and B. Lee, “Batch Verification for Equality of Discrete Logarithms and Threshold Decryptions,” in *International Conference on Applied Cryptography and Network Security*. Springer, 2004, pp. 494–508.

[19] P. Golle, M. Jakobsson, A. Juels, and P. Syverson, “Universal re-encryption for mixnets,” in *Cryptographers’ Track at the RSA Conference*. Springer, 2004, pp. 163–178.

[20] R. Dingledine, N. Mathewson, and P. Syverson, “Tor: The Second-generation Onion Router,” in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM’04. USENIX Association, 2004, pp. 21–21.

[21] C. Gentry, A. Sahai, and B. Waters, “Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8042 LNCS, no. PART 1, pp. 75–92, 2013.

[22] D. Chaum, "Blind signatures for untraceable payments," in *Advances in cryptology*. Springer, 1983, pp. 199–203.

[23] P. Y. Ryan, "Prêt À Voter with Paillier encryption," *Mathematical and Computer Modelling*, vol. 48, no. 9, pp. 1646–1662, 2008.

[24] C. Burton, C. Culnane, and S. Schneider, "vVote : Verifiable Electronic Voting in Practice," no. August, 2016.

[25] M. Jakobsson, A. Juels, and R. L. Rivest, "Making Mix Nets Robust for Electronic Voting by Randomized Partial Checking," in *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, 2002, pp. 339–353.

[26] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *Annual International Cryptology Conference*. Springer, 1992, pp. 89–105.

[27] S. Estehghari and Y. Desmedt, "Exploiting the client vulnerabilities in Internet e-voting systems: Hacking Helios 2.0 as an example," *Proc. 2010 Electronic Voting*, pp. 0–27, 2010.

[28] R. Lebre, P. Ferreira, R. Joaquim, and A. Zuquete, "Internet Voting: Improving Resistance to malicious Servers in REVS," *IADIS International Conference on Applied Computing*, no. March, 2004.

[29] Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[30] F. Brandt, "Efficient Cryptographic Protocol Design Based on Distributed El Gamal Encryption," in *Proceedings of the 8th International Conference on Information Security and Cryptology*, ser. ICISC'05. Springer-Verlag, 2006, pp. 32–47.

[31] D. Chaum, "Surevote," *International patent WO*, vol. 1, no. 55940, p. A1, 2001.

[32] S. Kremer and P. B. Ronne, "To du or not to du: A security analysis of Du-Vote," *Proceedings - 2016 IEEE European Symposium on Security and Privacy, EURO S and P 2016*, pp. 473–486, 2016.

[33] R. Cramer, R. Gennaro, and B. Schoenmakers, "A Secure and Optimally Efficient Multi-authority Election Scheme," in *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*, ser. EUROCRYPT'97. Springer-Verlag, 1997, pp. 103–118.

[34] D. Springall, T. Finkenauer, Z. Durumeric, J. Kitcat, H. Hursti, M. MacAlpine, and J. A. Halderman, "Security Analysis of the Estonian Internet Voting System," *Proceedings of the 21st ACM Conference on Computer and Communications Security*, no. May, p. 12, 2014.

[35] J. Gerlach and U. Gasser, “Three Case Studies from Switzerland: E-Voting,” *Berkman Center Research Publication*, p. 17, 2009.

[36] N. Braun and D. Brändli, “Swiss E-Voting Pilot Projects: Evaluation, Situation Analysis and How to Proceed,” *Electronic Voting 2006*, vol. P-87, pp. 27–35, 2006.

[37] S. Wolchok, E. Wustrow, D. Isabel, and J. A. Halderman, “Attacking the Washington, D. C. Internet Voting System,” *System*, pp. 1–18, 2012.

[38] V. Cortier and C. Wiedling, “A formal analysis of the Norwegian e-voting protocol,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7215 LNCS, no. 258865, pp. 109–128, 2012.

[39] R. E. Koenig, P. Locher, and R. Haenni, “Attacking the verification code mechanism in the norwegian internet voting system,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7985 LNCS, pp. 76–92, 2013.

[40] J. A. Halderman and V. Teague, “The New South Wales iVote system: Security failures and verification flaws in a live online election,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9269, no. March 2015, pp. 35–53, 2015.

[41] C. Culnane, P. Y. A. Ryan, S. Schneider, and V. Teague, “vVote: A Verifiable Voting System,” *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, pp. 3:1–3:30, Jun. 2015.

[42] “Software Guard Extensions Programming Reference. Intel Corp., 2013,” http://software.intel.com/sites/default/files/329298-001.pdf, accessed: 2017-01-04.

[43] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative Instructions and Software Model for Isolated Execution,” in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP ’13. ACM, 2013, pp. 10:1–10:1.

[44] “Android Key Store,” https://developer.android.com/training/articles/keystore.html, accessed: 2016-11-29.

[45] “Trusty TEE,” https://source.android.com/security/trusty/, accessed: 2017-01-04.

[46] N. Santos, H. Raj, S. Saroiu, and A. Wolman, “Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications,” no. i, 2014.

[47] N. Santos, N. O. Duarte, M. B. Costa, and P. Ferreira, “A Case for Enforcing App-Specific Constraints to Mobile Devices by Using Trust Leases,” *HotOS*, 2015.

[48] "Raspberry Pi 3 Model B," https://www.raspberrypi.org/products/raspberry-pi-3-model-b/, accessed: 2017-06-25.

MarkPledge 3 and Network Protocol

A.1 MarkPledge 3

MarkPledge 3 was proposed by Joaquim and Ribeiro [5] and defines how a vote is encrypted and validated such that the voter can verify, with high probability, that the vote encryption corresponds to the voter's choice.

This technique assumes the existence of an ElGamal election public key with public parameters p, q, g and h and requires that all encryptions are performed using the exponential ElGamal cryptosystem (Section 2.2.1). Its interface is divided in the following set of functions:

1. **Vote encryption** $\mathcal{VE}_h(b, \theta, \tau, \delta) =< u = E_h(b, \tau), v = E_h(\theta, \delta), voteValidity >$

   The vote encryption function produces an encryption u of a $NOvote$ $(b = -1)$ or $YESvote$ $(b = 1)$ that can be associated with a candidate. The encryption of a $YESvote$ or $NOvote$ is performed using exponential ElGamal with a random number τ, i.e. a $YESvote$ corresponds to the exponential ElGamal encryption of 1 and a $NOvote$ corresponds to the exponential ElGamal encryption of -1.

The parameter θ corresponds to a random confirmation number assigned to the candidate, and the vote encryption function also encrypts θ with random factor δ, resulting in the return value v.

Finally, this function also computes vote validity attributes. The vote validity attributes are used to verify if the encryption u is in fact an encryption of a $YESvote$ or $NOvote$. These attributes are computed based on the ballot validity proof proposed by Cramer et. al [33].

2. **Vote validity** $\mathcal{VV}_h(u, voteValidity) = True$ or $False$

   The vote validity function returns $True$ if the vote encryption u corresponds to an encryption of a $NOvote$ $(b = -1)$ or $YESvote$ $(b = 1)$, i.e. if it is a valid vote. Returns $False$ otherwise.

3. **Receipt creation** $\mathcal{RC}_h(b, \theta, \tau, \delta, c) = (\vartheta, \omega)$

$$\vartheta = \begin{cases} \theta & \text{if } b = 1 \ (YESvote) \\ 2c - \theta \mod q & \text{if } b = -1 \ (NOvote) \end{cases} \quad \omega = \tau \times (c - \vartheta) + \delta \mod q \tag{A.1}$$

   The receipt creation function outputs a receipt ϑ, ω that allows the voter to verify if the voter's computer encrypted the vote according to his intentions. The $\mathcal{RC}_h$ function starts by computing a verification code ϑ that corresponds to the candidate's confirmation code θ if the vote is a $YESvote$, and the symmetric of θ where c is the symmetry axis. The function also computes ω which is a combination of the random factors τ and δ previously used to compute u and v.

4. **Receipt validity** $\mathcal{RV}_h(u, v, c, \vartheta, \omega) = result$

$$result = \begin{cases} True & \text{if } E_h(c, \omega) = u^{c-\vartheta} \times v \\ False & \text{otherwise} \end{cases} \tag{A.2}$$

   The receipt validity function returns $True$ if a receipt is valid and $False$ otherwise. The verification code ϑ is validated by verifying if $u^{c-\vartheta} \times v$ is the encryption of c with random factor ω. This is a zero knowledge verification that can be done with only the knowledge of the public values.

5. **Tally** $\mathcal{TF}(homomorphicVoteAggregation) = (voteCount_1, ..., voteCount_n)$

   The tally function takes as input the homomorphic aggregation of the votes, i.e. the multiplication of every submitted vote encryption u for each candidate, and returns the final tally, i.e. the number of $YESvotes$ for each candidate.

A.2 Entities

TrustedVote takes into consideration the following entities and services:

- **Electoral Commission (EC)** - responsible for the entire election process, and authentication of all public data.
- **Enrollment Service (ES)** - responsible for the enrollment of every voter.
- **Election Registrar (ER)** - service that voters register to vote on a specific election.
- **Ballot Box (BBox)** - service that voters use to send their vote.
- **Bulletin Board (BB)** - public service that shows the current state of the election, i.e. submitted ballots and election public key.
- **Verification Service (VS)** - each organization runs an instance of the verification service that verifies if the votes and receipts are correct and valid.
- **Trustees ($\mathcal{T}$)** - set of organizations and parties that keep secret an ElGamal [16] election asymmetric key pair (K_{pub}, K_{priv}). Each trustee has a share of K_{priv} such that the decryption of a message requires the collaboration of $t < n$ trustees, where n is the total number of trustees [33].
- **Voter's device ($\mathcal{V}$)** - an ARM TrustZone-enabled mobile device that the voter uses to cast a vote in elections.

It is assumed that each entity has an asymmetric key pair publicly known by the other entities. For instance, the key pair of the Election Registrar service is represented as (K_{ER}, K_{pER}), where K_{ER} is the public key, and K_{pER} the private key.

The encryption of $message$ with K_{ER} is denoted by $(message)_{K_{ER}}$. The $message$ and concatenation of its signature with K_{pER} is denoted by $(message)_{K_{pER}}$

A.3 Protocol

TrustedVote is built on top of an ElGamal election key (K, K_p) [16]. However, it requires that the election private key K_p is split between a set of n trustees, such that a decryption operation requires the cooperation of $t < n$ trustees [33]. The votes are encrypted using the MarkPledge 3 technique, that ensures the voter (or any independent organization) can verify if the vote is counted and is encrypted correctly.

The protocol is divided in three phases: 1) the election setup, 2) the voting phase, and 3) the tally and verification phase.

Candidate	Encryption u	Vote Validity	Receipt ϑ
Alice	$NOvote1$	$vv1$	-
Bruno	$YESvote$	$vv2$	-
Donald	$NOvote2$	$vv3$	-
Garvin	$NOvote3$	$vv4$	-

Table A.1: An example of an empty ballot. The position of the $YESvote$ is randomly selected.

A.3.1 Election Setup

The Election Setup stage is performed before the election period and is where the election public key K is computed by the Trustees and validated by the Electoral Commission. Also at this phase, the voters who intend to vote on the upcoming election must register and commit an empty ballot. In this Section, we describe the details of the Election Setup phase.

1. $\mathcal{T}_t \rightarrow Bulletin\ Board \mid (pk_t)_{K_{p\mathcal{T}_t}}$

   Each trustee t sends his share pk_t of the election public key K to the Bulletin Board. The election public key K is now available.

2. $EC \rightarrow Bulletin\ Board \mid (candidateList)_{K_{pEC}}, (electionParameters)_{K_{pEC}}$

   It is the responsibility of the Electoral Commission to provide the candidate list and verify the computation of the election public key K. Therefore, in this step the Electoral Commission sends the signed candidate list, and validates the election public key K by sending a signature to the Bulletin Board.

3. $ER \rightarrow \mathcal{V} \mid (candidateList)_{K_{pEC}}, (electionParameters)_{K_{pEC}}$

   The voter client retrieves the candidate list, the election public key from the Election Registrar and verifies the corresponding signatures.

4. $\mathcal{V} \rightarrow ER \mid (ballot)_{K_{p\mathcal{V}}}$

   A voter registers to vote by creating and committing an empty ballot to the Bulletin Board. A *TrustedVote* ballot is a structure that contains the following information for each candidate:

   - The canonical vote encryption, i.e. the u element returned by the vote encryption function $\mathcal{VE}_h$ of MarkPledge 3.
   - The vote validity proof computed by the vote validity function $\mathcal{VV}_h$ of MarkPledge 3.
   - The vote receipt as returned by the $\mathcal{RC}_h$ function of MarkPledge 3.

   To create an empty ballot the voter creates a $YESvote$ encryption ($\mathcal{VE}_h$ with $b = 1$) and randomly assigns it to a candidate. For the remaining candidates, it creates $NOvote$ encryptions ($\mathcal{VE}_h$ with

Candidate	Vote code
Alice	A4CD
Bruno	FF25
Donald	BED1
Garvin	1AF5
Confirmation code: A32F	

Table A.2: *TrustedVote* code card.

$b = -1$) for each one of the remaining candidates. The voter also includes in the empty ballot the corresponding vote validities vv for each vote encryption. The vote receipt is not included in the empty ballot.

The empty ballot is signed with the voter's private key K_{pV} before sending to the Election Registrar.

Upon receiving the ballot, the Election Registrar validates the vote using $\mathcal{VV}$ from the MarkPledge 3 specification, respectively. Moreover, it also verifies if only one $YESvote$ is present in the ballot, by verifying the homomorphic sum of the vote encryptions. It signs the ballot and sends it to the Bulletin Board.

5. $\mathcal{V}$ generates a random code card for the upcoming election.

   Table A.2 shows an example of a code card. The code card is an association between a random vote code (string) and a candidate. The code card also contains a confirmation code, that is the confirmation code of the candidate that has the $YESvote$ encryption in the empty ballot. This confirmation code allows the voter to confirm in the receipt, by visually comparing strings, if the vote matches his intentions.

   The code card must be generated in a secure context and must be kept private from third parties as the voter will use the vote codes (instead of the direct identification of the candidates) to vote.

A.3.2 Voting phase

The voting phase starts when the voting period starts and ends when the voting period ends. In this phase, each voter that previously registered to vote in the Election Setup phase (Section A.3.1) creates his final ballot and submits it to the Ballot Box.

1. $\mathcal{T}_t \rightarrow Bulletin\ Board \mid (random_number)_{K_{p\mathcal{T}_t}}$

   Each trustee t sends a random number to the public Bulletin Board in order to generate an election challenge. The final election challenge c is computed by applying bitwise XOR to all random numbers submitted by the trustees. This ensures that if at least one trustee is honest, then the

Candidate	**Encryption** u	**Vote Validity**	**Receipt** ϑ
Alice	$NOvote3$	$vv4$	18DF
Bruno	$NOvote1$	$vv1$	BA1D
Donald	$YESvote$	$vv2$	**A32F**
Garvin	$NOvote2$	$vv3$	19EE

Table A.3: The final vote produced with the empty ballot of Table A.1 and where Donald is the selected candidate.

resulting election challenge is really random. The Electoral Commission validates the election challenge c by posting its signature on the Bulletin Board.

2. $Bulletin\ Board \rightarrow \mathcal{V} \mid (c)_{K_{pEC}}$

   Then, the voter receives the election challenge computed in the previous step and validates its signature. The election challenge is used, in the next step, by the voter, to create the vote receipt.

3. $Voter \rightarrow \mathcal{V} \mid (voteCode)$

   The voter must now consult his code card and insert in the voter's mobile device V the desired vote code. The vote code is translated into the corresponding candidate and the final vote is computed.

   The final vote is computed by rotating the vote encryptions (and the corresponding vote validities) until the $YESvote$ matches the desired candidate. Finally, the voter's device $\mathcal{V}$ computes the receipt ϑ for each candidate, using the receipt creation function RC_h of MarkPledge 3, and prints it to the voter. The voter confirms that the vote matches his intentions by checking if the confirmation code on the code card is linked with the desired candidate.

   Table A.3 demonstrates the final vote when Donald (from the empty ballot in Table A.1) is the selected candidate. In the code card (see Table A.2) the confirmation code is A32F. When the receipt is shown to the voter, he confirms that the final vote matches his intentions by checking whether A32F is linked with the candidate Donald. Therefore, only a person with access to the code card acknowledges how a voter cast his vote.

4. $\mathcal{V} \rightarrow BBox \mid (final_vote)_{K_{p\mathcal{V}}}$

   After the voter confirms that the final vote matches his intentions, the final vote (the vote encryptions, vote validity attributes and receipts) are sent to the Ballot Box. The Ballot Box service invokes the vote validity $\mathcal{VV}_{\langle}$ and the receipt validity $\mathcal{RV}_{\langle}$ functions of MarkPledge 3 for each candidate in order to verify the correctness of the final vote. After verifying its signature, the final vote is published in the public Bulletin Board. The voter must confirm that his vote is published in the Bulletin Board.

A.3.3 Tally and verification

The tally and verification phase is the last phase and occurs after the voting period has ended. In this phase, the homomorphic aggregation of the votes is computed by the Bulletin Board and signed by the Electoral Commission, and later decrypted with the cooperation of $t < n$ trustees. With the results posted in the Bulletin Board, any independent third party organization can verify the correctness of the entire election.

1. $EC \rightarrow Bulletin\ Board \mid (homomorphic_vote_aggregation)_{K_{pEC}}$

   After the election period is over, the Bulletin Board computes the homomorphic vote aggregation and asks the Electoral Commission to validate the computation by signing it.

2. $\mathcal{T}_i \rightarrow Bulletin\ Board \mid (partial_decryption, decryption_proof)_{K_{p\mathcal{T}_i}}$

   Each trustee i collects the homomorphic vote encryption (verifying its signature) from the Bulletin Board and sends its partial decryption and decryption proof to the Bulletin Board. With t valid partial plain texts, the Bulletin Board is able to decrypt the final tally. The Electoral Commission signs the result and now the results are available.

3. $Bulletin\ Board \rightarrow VS \mid\ (candidate_list, ballot_list, final_votes)_{K_{pEC}}$

   Any independent organization can retrieve the election data from the public Bulletin Board and verify every vote encryption validity for every vote, every receipt validity for every receipt and all the signatures.