University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Secure Deployment and Configuration Management for a Decentralized Remote Electronic Voting System

Roger Staubli
Zurich, Switzerland
Student ID: 12-913-778

Supervisor: Christian Killer, Bruno Rodrigues
Date of Submission: July 09, 2021

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

The integrity and validity of elections are cornerstones in each modern democracy. With the introduction of e-voting systems, and in particular, remote electronic voting (REV) systems, completely new and unique challenges arise. REV systems allow citizen to cast their votes from a remote, uncontrolled device. To securely provide such a system, high privacy and verifiability requirements need to be fulfilled. As privacy and verifiability requirements are in direct opposition, wide research has already been conducted to provide cryptographic protocols in order to fulfill these requirements. However, as research mainly focuses on theoretical protocols and on proving the fulfillment of these requirements, less effort is made to provide implementations or productional deployments of such systems. Since the security of an application is only as strong as its weakest component, all aspects need to be carefully analyzed. In particular, a reproducible deployment of a REV system can provide a starting point to test the security, scalability, and usability of voting protocols in an end-to-end fashion.

This work focuses on the deployment of Provotum, a decentralized REV system. To provide a reproducible and testable application, a modular deployment framework was designed and implemented. This framework allows configuring and setting up the whole infrastructure with infrastructure-as-a-code (IAAC) and consists of a continuous integration / continuous delivery (CI/CD) pipeline, a private Docker registry, a vulnerability scanner, and a monitoring service. Within this infrastructure, it is possible to build Provotum securely and deploy it in a decentralized environment. Finally, the infrastructure and the deployed application are evaluated according to security, scalability, and usability. In addition, a literature research was carried out to identify core privacy and verifiability properties of REV systems.

Zusammenfassung

Die Integrität und Gültigkeit von Wahlen sind Eckpfeiler jeder modernen Demokratie. Mit der Einführung von E-Voting-Systemen und insbesondere von Remote Electronic Voting (REV) Systemen ergeben sich völlig neue und einzigartige Herausforderungen. REV-Systeme ermöglichen es Bürgern, ihre Stimmen von einem unkontrollierten Gerät aus abzugeben. Um ein solches System sicher bereitzustellen, müssen hohe Anforderungen an Privatsphäre und Verifizierbarkeit erfüllt werden. Da Privatsphäre und Verifizierbarkeitsanforderungen in direktem Widerspruch stehen, wurden bereits umfangreiche Studien durchgeführt, welche probieren, mit kryptografischen Protokollen diese Anforderungen zu erfüllen. Da sich die Forschung jedoch hauptsächlich auf theoretische Protokolle und den Beweis der Erfüllung dieser Anforderungen konzentriert, werden weniger Anstrengungen unternommen, um Implementierungen oder produktive Deployments solcher Systeme bereitzustellen. Da die Sicherheit einer Anwendung nur so stark ist wie ihre schwächste Komponente, müssen alle Aspekte sorgfältig analysiert werden. Insbesondere kann ein reproduzierbares Deployment eines REV-Systems einen Ausgangspunkt bieten, um die Sicherheit, Skalierbarkeit und Benutzerfreundlichkeit von Abstimmungsapplikationen zu testen.

Diese Arbeit konzentriert sich auf das Deployment von Provotum, einem dezentralen REV-System. Um eine reproduzierbare und testbare Anwendung bereitzustellen, wurde ein modulares Deployment Framework entworfen und implementiert. Dieses Framework ermöglicht die Konfiguration und Einrichtung der gesamten Infrastruktur mit Infrastructure-as-a-Code (IAAC) und besteht aus einer Continuous Integration / Continuous Delivery (CI/CD)-Pipeline, einer privaten Docker-Registry, einem Schwachstellen-Scanner und einer Überwachungsanwendung. Innerhalb dieser Infrastruktur ist es möglich, Provotum sicher zu bauen und in einer dezentralen Umgebung zu deployen. Schliesslich werden die Infrastruktur und die bereitgestellte Anwendung nach Sicherheit, Skalierbarkeit und Benutzerfreundlichkeit bewertet. Darüber hinaus wurde eine Literaturrecherche durchgeführt, um die wichtigsten Datenschutz- und Verifizierbarkeitsanforderungen von REV-Systemen zu identifizieren.

Acknowledgments

Hereby, I would like to express my gratitude to everyone who has supported me in achieving my academic goals during the last few years.

First of all, I would like to thank Christan Killer, who has guided me throughout my thesis. Without his great feedback and valuable discussions, the thesis would not be at the point where it is now. In addition, my thanks go to Prof. Dr. Burkhard Stiller for offering me the possibility to write the thesis at the Communication Systems Group.

Moreover, my thanks go to my brother Cyrill, who encouraged me during my whole study and always motivated me when I got tired.

Without their help, this achievement would not have been possible.

Contents

Chapter 1

Introduction

"Is there a reproducibility crisis?". The famous survey from Nature's journal about the reproducibility crisis started a broad discussion in all areas of science [2]. Can researchers reproduce other scientist's experiment results? Can they even produce their own results? The survey showed that 70% could not reproduce other researchers' results, and more than half of them could not even reproduce their own experiments. In addition, the majority of those surveyed answered that the replication of the lab environment is often not possible, and most of them wish to have a more robust experimental design [2]. If we apply those results to the field of computer science, many experimental setups within integrated systems are challenging to reproduce. Although computer science experiments are based on algorithms, reproducibility is often hard to achieve. These problems can have many different causes. Researchers might not publish the source code. They might not specify dependent software components, not providing well-documented code, or their hardware setup might not be reproducible, to name some of them [31].

Much research in the field of Remote Electronic Voting (REV) systems is currently going on [32]. These systems aim to allow citizens to vote from an electronic device in an uncontrolled environment (e.q. voting with the mobile phone over the internet) [19]. Due to the importance of fair elections in a democracy and the use of REV systems in an uncontrolled environment, a high level of privacy and verifiability is essential within such systems. As research often focuses on designing such voting protocols and on prooving of privacy and verifiability properties, reproducible prototypes are difficult to find. Since implementation details of such systems can significantly impact security, scalability, or usability, a reproducible and testable application is essential for other researchers. With that, they can repeat experiments, find implementation issues, improve the voting protocol or compare implementations with each other. For these reasons, a reproducible and configurable system is a small step forward in the challenging research of REV.

In this work, we aim to provide a prototype for the cloud deployment of a REV system. The goal is that other scientists, voting authorities, or third parties can fastly deploy the system in the cloud and test it without limitations.

1.1 Description of Work

The overarching goal of this thesis is to design and implement a reproducible deployment and configuration management system for Provotum 3.0 [19], a decentralized, REV system. The system aims to provide a secure and straightforward way to deploy a decentralized application within a cloud infrastructure. To reach this goal, a continuous integration/continuous delivery pipeline was designed and deployed, a container security analysis was conducted, and a configuration management prototype was implemented for Provotum 3.0. In addition to that, security, scalability, and usability for the deployed application were evaluated and analyzed.

Moreover, a literature research was performed to analyze the core properties of relevant centralized and decentralized remote electronic voting systems.

1.2 Thesis Outline

The thesis is structured as follows: Chapter 2 introduces the main properties of REV systems and compares relevant REV systems according to these properties. Chapter 3 gives an overview of important deployment concepts that, later on, will be used to design and deploy the infrastructure. Next, chapter 4 briefly presents system architectures of other relevant REV systems. The following chapter 5 starts with a container security threat model and continues with the CI/CD infrastructure design and the design of the deployment of Provotum 3.0. Chapter 6 displays implementation details of the developed prototype, which will be evaluated in chapter 7. Finally, chapter 8 summarizes the work and gives suggestions for future work.

Chapter 2

Property Analysis of REV Systems

This chapter focuses on the main concepts and properties of REV. It defines central properties for REV systems and compares systems from academic, commercial, and country-based implementations concerning these properties, domains, and underlying trust assumptions.

2.1 Definitions

REV systems aim to automatize the whole voting process from election creation over vote casting to votes tallying. Since elections are the cornerstone of many democracies, companies, or organizations, fair elections are essential within these areas. Based on the importance, many properties for REV systems have emerged intending to achieve a high level of privacy (e.q. nobody can see for whom someone voted) and at the same time a high level of verifiability (e.q. everybody can verify that the election outcome was counted right), even with untrusted authorities.
As privacy and verifiability are conflicting requirements [11], REV systems need unique characteristics to achieve the two at the same time. They mostly rely on cryptographic primitives aiming to satisfy these two requirements [34]. The following subsections will define the most critical properties used to compare REV systems afterward, separated into privacy and verifiability.

2.1.1 Privacy

Definition 1. *Ballot-secrecy: "A voter's vote is not revealed to anyone" [53]*

The idea of Ballot-secrecy within electronic voting was firstly introduced by Chaum et al. [10]. Their idea was to use anonymous channels in electronic voting, which led to an increased research interest in the whole domain. The property expresses that anyone (including the voting authority) must not see for whom a voter voted. Ballot-secrecy is stated in the Universal Declaration of Human Rights [57] which builds a cornerstone of

each election.

Definition 2. *Receipt-freeness: "A voter does not gain any information (a receipt) which can be used to prove to a coercer that she voted in a certain way" [18]*

Benaloh and Tuinstra [4] realized that all existing electronic voting systems up to this time allowed voters to take away a receipt of how they voted. This receipt provides vote-buying and coercion since the voter can prove how she voted. A receipt-free system must ensure that any voter receives a receipt such that she can not prove to someone how she voted. This property only arises within electronic voting, whereas in traditional paper-based voting, the deniability of a vote is given by the ballot box. Ballot-secrecy is a subset of receipt-freeness, meaning each receipt-free system also provides ballot-secrecy.

Definition 3. *Coercion-resistance: "A voter cannot cooperate with a coercer to prove to him that she voted in a certain way." [18]*

Coercion-resistance is a stronger property than receipt-freeness (hence also stronger than ballot-secrecy) [18]. It states that a voter can not prove to a coercer how she voted even when they both cooperate. The main difference between coercion-resistance and receipt-freeness lies in the proof generation at different voting phases. Receipt-freeness expects that a voter does not get a receipt after the vote casting. In contrast, coercion-resistance expects a voter to not prove to an adversary how she voted while executing the voting protocol. Most coercion-resistant voting systems allow voters to cast indistinguishable fake votes while they are coerced. Before tallying the votes, the voting system deletes fake votes [12].

Definition 4. *Everlasting privacy: A voting protocol provides everlasting privacy if an adversary using a computationally unbounded system can not gain information about for whom a voter voted. [41]*

Everlasting privacy removes the requirement of a computationally bounded system [41]. However, compared to ballot-secrecy it assumes a trusted voting authority [35]. REV systems that rely on asymmetric cryptography can not achieve everlasting privacy since computationally unbounded systems can break them. [11]

Definition 5. *Unconditional privacy: "A voting system with unconditional privacy does not allow anyone, neither voters, outside observers nor voting authorities, be they computationally unbounded or not, to determine for whom another voter voted" [35]*

Additional to everlasting privacy, unconditional privacy does not require anybody to be trusted within the system. It combines the property of ballot-secrecy with the property of everlasting privacy [35]

Definition 6. *Software independence: "A voting system is software independent if an (undetected) change or error in its software cannot cause an undetectable change or error in an election outcome." [50]*

Within the domain of REV systems, software independence mainly rely on the cryptographic primitives included. Due to cryptography, changes can be immediately detected within the system. Let us assume a voter's encrypted ballot (including a signature with the voter's private key) gets changed. If anyone else checks the ballot's signature using the voter's public key, he immediately realizes an invalid certificate, independently of the underlying system. The same holds if the final tallying was computed incorrectly [50].

2.1.2 Verifiability

Definition 7. *Individual verifiability: "A voter can verify that the ballot containing her vote is in the published set of "all" (as claimed by the system) votes." [32]*

Individual verifiability alone states that only a single voter can verify that his vote counts correctly [32]. To guarantee verifiability for the whole election, everybody (observers, voters, voting authorities, etc.) should have the possibility to verify that votes were counted correctly. For this reason, Sako and Kilian [52] introduced the property of universal verifiability.

Definition 8. *Universal verifiability: "Anyone can verify that the result corresponds with the published set of "all" votes." [32]*

Cryptographic based REV systems often rely on the use of a public bulletin board (PBB) as a published set of votes to ensure universal verifiability [1] [8] [12]. The PBB contains all information (encrypted votes, public keys, cryptographic proofs, etc.) such that everyone has the information to calculate and verify the final tally.

Definition 9. *Eligibility verifiability: "Anyone can check that each vote in the election outcome was cast by a registered voter and there is at most one vote per voter." [37]*

An additional property to the verifiability properties is eligibility verifiability. A system that aims to satisfy eligibility verifiability must allow anyone to check that all the votes were cast and tallied by eligible voters.

Definition 10. *End-to-end verifiability*

Cast-as-intended: "A voter can verify that her choice was correctly denoted on the ballot by the system" [32]

Recorded-as-cast: "A voter can verify that her ballot was received the way she cast it" [32]

Tallied-as-recorded: "A voter can verify that her ballot counts as received" [32]

End-to-end verifiability is a newer notion of verifiability, firstly introduced by Chaum [9] and consists of three parts: cast-as-intended, recorded-as-cast, and tallied-as-recorded. This property ensures that verifiability can be checked along the whole voting process such that a voter can verify that her choice is represented in the final tally.

The first property is cast-as-intended. The system should allow the voter to verify that his choice is represented on his ballot. In the domain of REV systems, voting clients (e.q. mobile phones) might run malicious code that modifies the voter's choice. REV systems that satisfy the property cast-as-intended often use a challenge-or-cast mechanism [3]. The created ballot can be challenged such that the validity is audited. This ballot needs to be discharged afterward, and a new ballot needs to be created not to violate receipt-freeness. This procedure is repeated until the voter is sure that his choice is represented on the ballot. He can then cast the ballot.
Another possibility to satisfy cast-as-intended is to send a code through a separate channel (e.q. postal mail) to the voter for each possible choice. This code can then be compared with the code that appears after the creation of the ballot. If it is correct, the ballot contains the voter's choice [25, 56].

The second property, recorded-as-cast, is fulfilled if the voter can verify that the cast ballot was stored without modification.

The third property, tallied-as-recorded, is fulfilled when the voter can verify that a stored ballot was counted without modification.

2.2 Relevant REV Systems

REV system implementations are often built upon well-known voting protocols. To get indications on the fulfillment of privacy and verifiability properties, it is helpful to evaluate the implementation's domain protocols. These domain protocols use cryptographic building blocks and follow different approaches to satisfy various levels of privacy and verifiability. In this section, relevant REV systems are presented categorized in their domain protocols.

2.2.1 CGS97

The protocol from Cramer, Gennaro, and Schoenmakers [17] was the first that introduced an efficient distributed authority setting. This multi-authority setting reduces the trust assumption since trust can be distributed among different authorities [32]. The protocol uses a DKG (DKG) among all the authorities such that their public key shares could be combined to an elections public key. The voter then uses this public key to encrypt her

ballot in the voting phase. During the validation phase, the encrypted ballots get summed up using homomorphic encryption, and in the tallying stage, each authority decrypts his partial sum [17]. The combination of the decrypted shares results in the number of yes votes for the election. Due to the homomorphic operation and decryption of the sum, the protocol only allows elections in a binary form (e.q. yes or no).

CGS97 is universally verifiable since all the validation and tallying operations are done on the PBB, containing proofs of the operations (decryption proofs from the authorities) and can be seen by anyone. Hence, anyone can verify that the result corresponds to the set of published votes [32]. As long as not all authorities collude, the protocol also fulfills ballot-secrecy since only the sum of the ballots gets decrypted and not each ballot by itself. However, it is not receipt-free as each voter can reproduce the whole voting process with her vote and therefore prove how she voted. Coercion resistance is neither fulfilled. The protocol itself does not satisfy end-to-end verifiability, as it does not consider compromised voting devices. Hence, cast-as-intended can not be fulfilled. In addition, eligibility verifiability is also out of scope from the protocol.

Figure 2.1 displays an overview of different implementations from the base protocol CGS97.

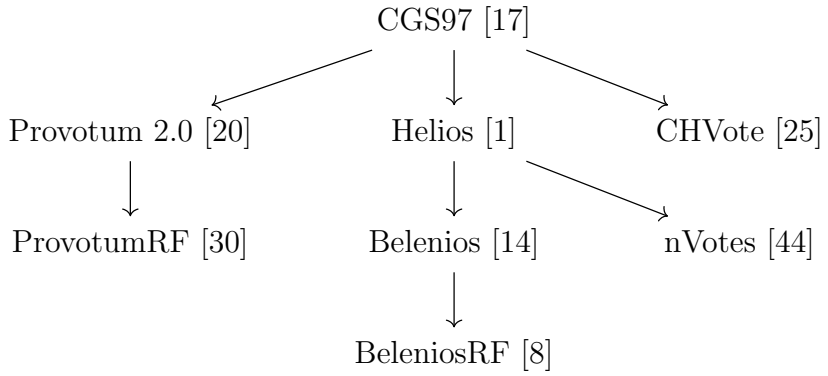

Figure 2.1: CGS97 domain protocol and their dependent implementations

Helios [1]: Helios is an open-source, web-based REV system, which can be used by anyone to run their election [1]. The protocol is heavily based on CGS97. It uses the DKG for a multi-authority setting and homomorphic encryption in the style of CGS97. The Helios authors do not claim to add new concepts to the base protocol rather implement a practical system such that it can be used in real-world elections [32]. They also say that Helios should be used in low-coercion environments, like in small university or association elections [1].

As such, Helios does not fulfill receipt-freeness, coercion resistance and only partly fulfills ballot-secrecy due to the famous replay attack [15].

Looking at verifiability properties, Helios implemented a challenge-or-cast mechanism to fulfill the cast-as-intended property. Recorded-as-cast is fulfilled since the voter can observe his encrypted ballot on the PBB, and tallied-as-recorded can be shown by the distributed decryption of the sum of the votes (including right decryption proof from

the authorities). Moreover, Helios does not fulfill the property of eligibility verifiability. Voters must trust the voting authority that only eligible voters are accepted [38].

Helios was already used in small but legally binding elections within academic contexts like a presidential election at the Universite Catholique de Louvain in 2009 [34].

Belenios [14]: Belenios is a web-based, REV systems that aims to fulfill ballot-secrecy, end-to-end verifiability and eligibility verifiability [14]. The protocol builds upon the Helios protocol with minor differences.

Firstly, it adds a trusted registrar, which is responsible for distributing a private-public key pair to all the eligible voters (using an authenticated or an off-line channel). The voter then uses the private key to sign the ballot. All public keys are published on the PBB such that everyone can verify that only eligible voters have cast ballots [16]. If the registrar is trusted, Belenios additionally fulfills eligibility verifiability.

Secondly, since the ballot is signed by the voter, a replay attack with using the same ballot as another voter is not possible anymore [14].

Thirdly, Belenios does not implement a challenge-or-cast functionality, compared to Helios, since the authors claim that the challenge functionality was not used often in practice and voters that don't do the challenge are easy targets for attacks [14].

BeleniosRF [8]: BeleniosRF builds on the protocol of Belenios. As the name already indicates, BeleniosRF adds receipt-freeness to the protocol. The authors introduced the functionality of a randomizer that re-encrypts the ballot before casting [8]. As long as the voter and randomizer do not collude, the voter can not reproduce his encrypted ballot anymore and, therefore, does not have a receipt for his voting decision.

Provotum 2.0 [20]: Provotum 2.0 is a web-based, REV system which is strongly based on the domain protocol CGS97. The protocol uses a multi-authority scheme implementing DKG, homomorphic tallying, and the Ethereum blockchain [21] as the PBB. In addition, an identity provider and an access provider are implemented to distribute trust among different authorities.

The protocol is built up in a registration, a pairing, a key generation, a voting, a tallying, and a result stage. In the registration stage, sealers (signing authorities) generate their Ethereum wallet and send the wallet address to the voting authority. In the pairing stage, sealers start their node and start validating blocks in a proof-of-authority protocol. After the protocol is running, the voting authority can deploy a smart contract containing the election question. Next, in the key generation stage, the DKG is performed as sealers submit their ElGamal public key to the smart contract. They are then combined to the votes public key, which encrypts the voter's ballots. During the voting phase, voters can authenticate against the identity provider, generate an Ethereum wallet, and request access from the access provider to cast a vote. After the voter expresses her choice, the ballot is encrypted with the vote's public key and submitted (along with a proof of

validity) to the smart contract. In the tallying phase, each sealer homomorphically sums up the encrypted votes and decrypts its share. The combination of all decrypted shares finally results in the sum of all yes votes.

The implementation of Provotum 2.0 fulfills similar properties to the domain protocol CGS97. It fulfills ballot-secrecy as the ballots are always encrypted, and only the sum of the votes gets decrypted. Hence, nobody can see which voter voted for whom or what. Receipt-freeness is not fulfilled since the voter can reproduce his ballot in front of an adversary or vote buyer and proof how she voted. Coercion-resistance is therefore not fulfilled either.

End-to-end verifiability is not fully satisfied since cast-as-intended is violated for the same reason as in CGS97. Recorded-as-cast is satisfied since the ballot can be found on the smart contract (PBB) after casting. Tallied-as-recorded is also fulfilled due to the right decryption proofs of the sealers. Eligibility verifiability is fulfilled (as long as the identity and access provider are trusted) because the access provider publishes the public Ethereum address of an eligible voter on the smart contract [20].

ProvotumRF [30]: ProvotumRF builds on the implementation of Provotum 2.0. The protocol is extended with a randomizer that re-encrypts the voter's encrypted ballots. Similar to BeleniosRF, the voter then can not reproduce her ballot anymore. Assuming the randomizer and the voter do not collude, ProvotumRF additionally fulfills receipt-freeness.

CHVote [25]: CHVote 2.0 was started in 2016 by the Swiss state of Geneva with cooperation with the Bern University of Applied Sciences. The goal was to introduce a cryptographic REV system with maximum transparency and full verifiability that can be used in public elections. Due to financial reasons, the cooperation came to an end, and Bern University of Applied Science continued the project in 2019 with a new founding partner [25]. Their requirements for the project are end-to-end verifiability, individual verifiability, eligibility verifiability, and the distribution of trust (multi-authority) [25].

Likewise, in CGS97, the protocol uses DKG among different election authorities. The combined election public key is then used to encrypt voters' ballots. After the voting phase, in contrast to CGS97, the votes are mixed in a mix-net to disconnect ballots from voters. To tally the votes, the ballots can, in the end, be partly decrypted by the election authorities.

To ensure end-to-end verifiability, the protocol uses verification, voting, confirmation, finalization, and abstention codes, which are printed by a printing authority and are sent over postal mail to the voter. The voter can then use the codes to check cast-as-intended, recorded-as-cast, and tallied-as-recorded [25]. It is important to note that end-to-end verifiability only holds if some trust assumptions are met. Mainly, the printing authority needs to be trusted since they could see all the codes on the voting paper and, therefore, behave as valid voters.

As CHVote uses a mix-net implementation, many different election styles can be performed with the protocol. The authors cover all different election styles in Switzerland,

like a referendum, approval voting, or cumulative voting.

nVotes [44]: nVotes is a web-based REV system developed by the Spanish company Agora Voting SL [44]. Since the protocol is not research standard, minimal information about the protocol is publicly available. However, del Blanco et al. [5] had insights into the protocol and could analyze some privacy and verifiability properties.

According to del Blanco et al. [5] the voting protocol is heavily based on Helios and starts with a DKG among different authorities. A public key for the election is then generated. Voters can afterward login to the web platform using an SMS code. After choosing the answer and creating the encrypted ballot, a challenge-or-cast functionality is available to ensure cast-as-intended. Before the tallying phase, the votes get mixed with a verificatum mix-net [58]. In the last step, partial decryption is employed, and the votes are tallied.

The protocol is considered to fulfill end-to-end verifiability if many voters use the challenge-or-cast functionality and if the authorities not collude [5]. Receipt-freeness and, therefore, coercion-resistance is not given for the same reason as in the Helios case.

According to the website, more than 2 million votes have been cast in total, and they performed more than 150'000 votes within one election [44]. Their clients range from public administrations (Barcelona Provincial Council, Madrid City Council) to education institutions (UNED) [5].

2.2.2 JCJ05

Jules, Catalano, and Jakobsson were the first that introduced a new privacy property, namely coercion-resistance, and proposed a new protocol that satisfied the property [33]. The protocol uses a multi-authority scheme with DKG, a mix-net to disconnect ballots from the voters, and private credentials generated by a registrar [33].

During the registration phase, the voter gets the private credentials from the registrar over an untappable channel. The registrar stores the encrypted private credentials on the PBB for verification later on. These credentials define an eligible voter. The voter can cast the encrypted ballot, the encrypted credentials, and proof of validity in the voting phase. During tallying, the authorities check the proofs, remove duplicates, mix the ballots and then remove ballots with invalid private credentials. The removal is done with plain-text equivalence tests such that the private credentials are never exposed to anyone. The authorities can then decrypt the final list of ballots, and the result can be calculated.

The reason why this protocol is coercion-resistant lies in the fact that voter can submit ballots with fake credentials. After the mixing (when ballots and voters can not be mapped anymore), these fake ballots are removed by the protocol such that nobody can realize anymore who cast the fake ballot [32]. With this functionality, the voter can use fake credentials during coercion and repeat the voting with a valid ballot when she is not coerced. Therefore, the voting protocol also fulfills receipt-freeness (the voter can not prove if she used fake or valid credentials) and ballot-secrecy (the decrypted ballot is not

connected back to a voter). The protocol does not fulfill end-to-end verifiability since cast-as-intended is not proved within an environment of compromised voting devices.

Figure 2.2 shows important dependent implementations from the base protocol JCJ05

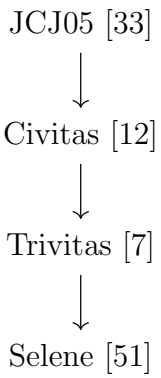

Figure 2.2: JCJ05 domain protocol and their dependent implementations

Civitas [12]: Civitas is the first known implementation that fulfills the strong notion of coercion resistance [32]. The project implemented the protocol JCJ05 but with some differences. The authors implemented a registrar and different registration tellers to split up trust over the distributor of private credentials. The registration tellers only generate a share of the final voters' private credentials. The voter then requests all shares from the registration tellers and combines them with his full private credential. With this functionality, only the voter owns the full plain-text of the private credentials (as long as not all registration tellers do collude) [32].

One main disadvantage of the implementation is the fact that the removal of invalid ballots with a plain-text equivalence test takes quadratic time in computation [43]. However, newer approaches for this problem have already reduced the computational time complexity from quadratic time to linear time [61] [55].

Civitas also fulfills universal verifiability since all operations on the PBB can be verified by anyone. Proofs for right mixing, for the plain-text equivalence test for removing invalid votes, and for right decryption can be checked on the PBB [7].

Trivitas [7]: The authors of Trivitas aim to improve individual verifiability based on the implementation of Civitas. The authors criticize the complex mathematical operations required for a voter to verify that his ballot is included in the final published set of votes within famous implementations like Civitas or Helios. Moreover, these implementations normally require that a voter performs the verification on a trusted device.

Trivitas introduces the notion of a trial ballot. The trial ballot contains trial credentials that can be cast like a valid or fake ballot but will not be counted in the final tally. A flagged trial ballot can be used as an audit ballot in three different voting phases. Before the mixing, trial ballots get decrypted and posted on the PBB such that the voter can assure that the votes have correctly been encrypted and recorded on the PBB. Next, the set of trial ballots also get mixed with the valid and fake ballots and afterward posted

on the PBB. The voter can assure that all his votes have been used as an input for the mix-net. Finally, trial ballots get decrypted and posted on the PBB after the mixing. The voter can assure that the vote occurs on the election outcome. With this functionality, the voter can track the trial ballot along with the voting protocol and can ensure that it works properly [7]. The idea of a trial ballot is similar to a challenge-or-cast ballot. However, it does not get discharged after the ballot creation rather tracked during all steps of the voting protocol.

As such, Trivitas adds the fulfillment of cast-as-intended to the base protocol of JCJ05 because the trial ballot can be audited in the decryption step before the mixing. Hence, a voter can check on the PBB that a trial ballot correctly denoted his choice.

Selene [51]: Selene is a voting protocol based on JCJ05 but takes the approach of a private tracking number for each voter. The intention is to make it easier for a voter to check end-to-end verifiability. The main idea is that decrypted ballots are published on the PBB along with a public tracking number. The voter can then easily check if her vote and the corresponding tracking number are included in the published list. The protocol implements the functionality using a private (α) and a public (β) part of the tracking number. The private part is transferred to the voter, and the public part is published on the PBB. The voter can then combine these two parts and decrypt them using his private key to get the public tracking number. It is also possible for the voter to fake α such that her decryption results in another tracking number published along with the decrypted ballots.

With this property, it is not possible for a voter to prove to someone how she voted. Hence, the protocol fulfills receipt-freeness. It is not coercion-resistant since a coercer could watch the voter casting his vote. However, it can reduce the coercion risk since the voter can present a wrong tracking number to the coercer.

The protocol fulfills end-to-end verifiability since to voter can create his tracking number from the true α and β and check if the vote was correct in the final list of tallied ballots. Eligibility verifiability was not considered in this protocol [51].

2.2.3 HRZ08

Hao, Ryan, and Zielinski proposed a self-tallying, two-round, distributed voting protocol [27]. The idea is that no central authority is required for the protocol, and the tallying is self-enforced by the protocol. All n voters first agree on (G, g) where G denotes a finite cyclic group of prime order q and g denotes the generator in G. Each voter generates a secret $x_i \in \mathbb{Z}_q$ and a public part g^{x_i}.

In the first round, the voter submits g^{x_i} and a zero-knowledge proof for x_i. After every voter has submitted the public part, each voter checks the others zero-knowledge proof validity and computes

$$g^{y_i} = \prod_{j=1}^{i-1} g^{x_j} / \prod_{j=i+1}^{n} g^{x_j}$$

In the second part each voter submits the ballot $g^{x_i y_i} g^{v_i}$ and a zero knowledge proof showing that v_i is either 0 or 1 (binary election). If every voter has submitted the ballot, every observer can compute

$$\prod_{i=0}^{n} g^{x_i y_i} g^{v_i} = g^{\sum_{i=0}^{n} v_i}$$

The term $\sum_{i=0}^{n} v_i$ reveals the number of yes votes of the election. $n - \sum_{i=0}^{n} v_i$ reveals the number of no votes [27].

The protocol HRZ08 fulfills ballot-secrecy since nobody can decrypt a single vote. However, receipt-freeness and coercion resistance is not given. The voter can prove how she voted by decrypting her vote. End-to-end verifiability is not given since cast-as-intended can not be assured if a voting device is compromised. Eligibility verifiability was not considered in this protocol. Moreover, the protocol assumes that every eligible voter that performed the first round also needs to perform the second round. Otherwise, it is not possible to calculate the final tally.

Figure 2.3 shows the dependent protocol on HRZ08 that will be analyzed in this thesis.

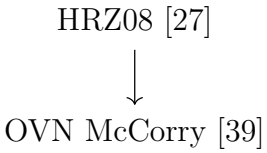

Figure 2.3: HRZ08 domain protocol and their dependent implementation

OVN McCorry [39]: OVN McCorry is a direct implementation of the domain protocol HRZ08. The implementation uses a smart contract on the Ethereum blockchain [21] as the PBB. A central authority generates a smart contract, defines (G, g), and adds all the eligible voters. The voters then perform the two-round protocol of HRZ08.
The implementation fulfills the same properties as the domain protocol but also adds eligibility verifiability to some extent. If the central authority is trusted and only eligible voters are included in the smart contract, everyone can check that only eligible voters cast votes to the final tally.
The implementation, however, also generates some usability issues. The use of smart contracts on the Ethereum blockchain only allows for approximately 60 votes. Otherwise, the maximal transaction fee would be too high [39].

2.2.4 Unrelated Systems

The property analysis conducted in this thesis does also contain REV systems that can not clearly be related to a domain protocol. This section presents these systems.

Voatz [54]: Voatz is the first REV system used in US federal elections. It is a smartphone app that should provide a simple user interface to cast votes. Moreover, a blockchain is

used as a PBB. Since Voatz does not provide much information about their protocol, this analysis is based on [54].
The authors of the work could not see the server infrastructure and therefore had to make assumptions about the systems and reverse engineer the app. They found out that the app had several security vulnerabilities and serious privacy issues. In addition, the server had complete control over the protocol and did not include proofs. Hence, the voting authority could potentially include fake votes or delete votes without detection. Privacy and verifiability are therefore not given if the authority is not trusted. In addition, they found vulnerabilities in the non-standard encryption scheme between the communication of the app and the server. The encryption scheme has the property that the encrypted ballot has a similar size to the ballot in plain text. Therefore, someone could guess the plain text by just seeing the encrypted ballot.

Estonia [28]: The Estonian REV system used in public elections between 2005 and 2015 did not provide strong verifiability. [28] proposed improvements in verifiability to the base protocol. The new protocol provides a PBB, homomorphic encryption, and a mix-net. Voters can encrypt their ballots using the elections public key, sign them using their private key, and send them to the PBB. To provide cast-as-intended and recorded-as-cast, the voter can request his ballot from the system, locally generate the ballot for all options again (with the same randomness), and compare these ballots with each other. When the encrypted ballot is the same, she can assure cast-as-intended and recorded-as-cast. However, receipt-freeness is therefore clearly violated. After the voting phase, the ballots get mixed and decrypted. Decrypted votes, proofs of correct decryption and mixing, and the eligible voter's public keys are published on the PBB. This ensures tallied-as-recorded and eligibility verifiability.

2.3 Comparison

Table 2.3 compares the discussed REV system base protocols and implementations. The properties claimed in the table have defined underlying trust assumptions. For example, if all authorities collude in a multi-authority protocol, they could decrypt all ballots and see the vote in plain text, and ballot-secrecy would be violated. Hence, it is important to define the underlying trust assumptions. The following list presents the trust assumptions for table 2.3:

- At least one authority in a multi-authority protocol is trusted.
- The voting device is not trusted.
- The PBB is trusted.
- The voter and the randomizer do not collude.

Specific assumptions for a REV system are denoted as superscripts directly under the table.

A note on the everlasting and unconditional privacy: Since all REV systems compared in this section are based on asymmetric cryptography, they can provide neither everlasting nor unconditional privacy.

A note on software independence: JCJ05 and Civitas do not provide software independence as they need to trust the private credentials creator in the deletion of fake votes step. If the system deletes a valid vote, there does not appear a detectable change in the voting outcome.

	Ballot-secrecy	Receipt-freeness	Coercion-resistance	Everlasting Privacy	Unconditional privacy	Software independence	Individual verifiability	Universal verifiability	Eligibility verifiability	Cast-as-intended	Recorded-as-cast	Tallied-as-recorded	Open Source	Installation Guidelines
Academic Protocols														
CSG97 [17]	●	○	○	○	○	●	●	●	○	○	●	●	NA	NA
JCJ05 [33]	●	●	●[1]	○	○	○	●	●	●	○	●	●	NA	NA
HRZ08 [27]	●	○	○	○	○	●	●	●	◐[2]	○	●	●	NA	NA
Academic Implementations														
Helios [1]	◐[3]	○	○	○	○	●	●	●	○	●[4]	●	●	●	●
Belenios [14]	●	○	○	○	○	●	●	●	●[5]	○	●	●	●	●
BeleniosRF [8]	●	●	○	○	○	●	●	●	●[5]	○	●	●	●	○
Civitas [12]	●	●	●[1]	○	○	○	●	●	●[5]	○	●	●	●	●
Selene [51]	●	●	○	○	○	●	●	●	○	●[6]	●	●	NA	NA
OVN McCorry [39]	●	○	○	○	○	●	●	●	◐[2]	○	●	●	●	●
Provotum 2.0 [20]	●	○	○	○	○	●	●	●	●	○	●	●	●	●
ProvotumRF [30]	●	●	○	○	○	●	●	●	●	○	●	●	◐[9]	◐[9]
Commercial Implementations														
nVotes [44]	●	○	○	○	○	●	●	●	○	●[4]	●	●	○	○
Voatz [54]	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Country Based Implementations														
CHVote [25]	●	○	○	○	○	●	●	●	●[5]	●[7]	●	●	●	●
Estonia [28]	●	○	○	○	○	●	●	●	●	●	●	●	◐[8]	○

[1] Needs trusted private credential creators
[2] Needs trusted contract creator
[3] Vote copying possible
[4] Challenge-or-cast
[5] Uses trusted registrar
[6] Used e2e tracking number
[7] Postal mailed verification code
[8] Partly published code
[9] Not published yet

Table 2.1: Comparison of the properties of REV systems, based on [36]

By looking at the two last properties, open-source and installation guidelines, we can see that commercial implementations lack to provide their source code. As a result, commercial implementations do not provide any reproducibility and can not be verified by third parties. On the other hand, academic implementations aim to provide their source code and installation guides. This does, however, not directly imply reproducibility. To provide reproducibility, other requirements like a buildable or executable application are essential. With the use of unprovided or deprecated dependent software components, a buildable or executable application would be impossible to provide [26]. Therefore, it

is essential to provide a reproducible infrastructure such that any other third party can replicate the exact implementation and perform the same experiments.

Chapter 3

Background

A decentralized REV system usually contains a complex combination of multiple software components (e.q. web-applications, server-applications, peer-to-peer-applications). Moreover, different developers or development teams are working on new versions of the protocol. To rapidly deploy new software components, great care needs to be taken to securely integrate code, unify build processes and configure these components. To design such an automated pipeline, many different IT Infrastructure and DevOps approaches need to be evaluated. This section gives an introduction to the main concepts of continuous integration (CI), continuous delivery (CD), infrastructure-as-a-code (IAAC), containerized applications, reproducibility, and distributed ledger technology (DLT).

3.1 Continuous Integration / Continuous Delivery

CI is a software development practice that aims to integrate code from different developers frequently. The integration is done at a centralized point and is followed by an automated build and automated tests. With the help of CI, problems and errors in the integration of code can quickly be detected and resolved. This can lead to an overall more cohesive software [23]. CD extends the idea of CI by automating the deployment process to test and integrate environments but still requires a manual deployment on production servers. Continuous deployment even automates the deployment to production environments [13].

3.2 Design Overview

A CI/CD design always starts with a single source code repository with a high-quality source control management system (e.q. GIT). All developers working on the project should checkout that repository using the same code-base. When a developer commits a new feature, an automated build should be triggered, followed by automated tests. The developer should then get immediate feedback if the build and test process was successful or not [23].

If the infrastructure does offer not only CI but also CD, after the build and test process, the software gets automatically deployed on a test or integration environment. In the case of a container environment (e.q. Docker) an intermediate binary registry is used to store and version the processed build.

Figure 3.1 displays a high-level design of a CI and CD pipeline. The Binary Registry is used to store the binary builds (e.q. Docker Images).

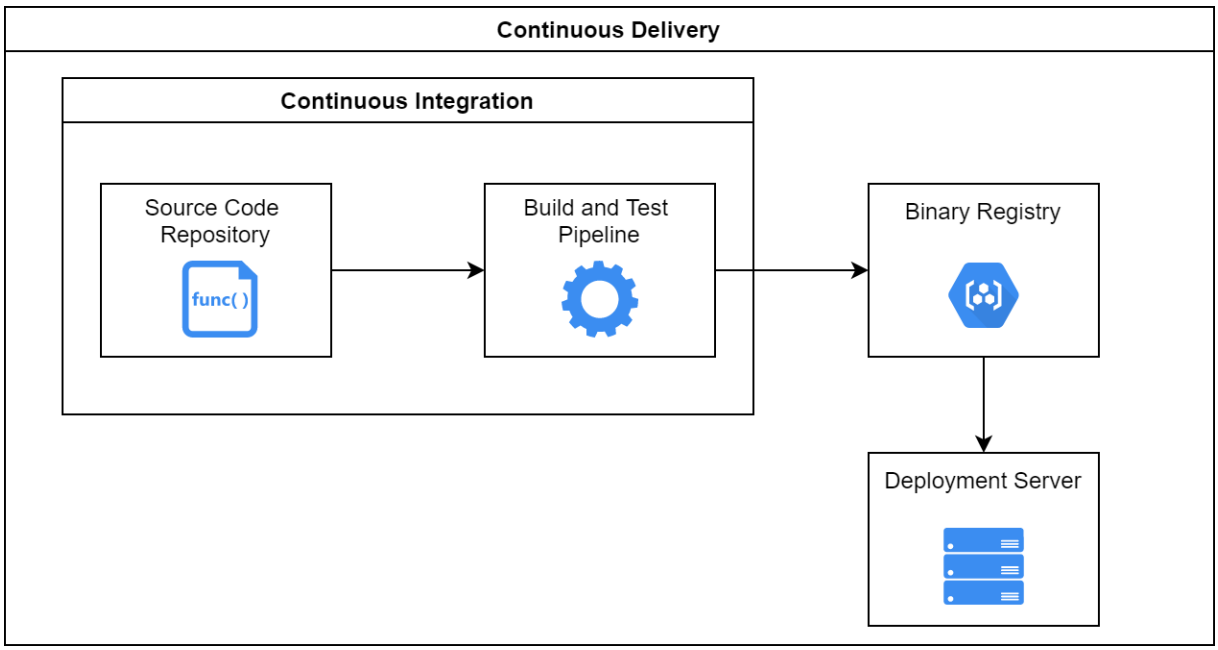

Figure 3.1: Overview of a CI/CD Infrastructure

As a CI/CD pipeline is the link between software development and the running code on servers, it is a high-risk target for attackers. Therefore, much care should be taken by deploying and configuring such a pipeline. Attacks to the pipeline are usually called Software Supply Chain Attacks (SSCA) [29]. There are many different possibilities on how a pipeline could be attacked and manipulated. Often adversaries gain access to the build machine and inject malicious code during build time. This code then gets deployed on test servers and, in the worst case, even on production servers [49]. Other attacks could include stealing information from the build pipeline like signing certificates [29].

The attack surface can be reduced by eliminating unnecessary tools on the building machine and by restricting user access to the machine. Moreover, the pipeline should be run protected from unauthorized network access using firewalls and be isolated from other infrastructural components [49]. More details about the security of the build machine is provided in section 5.1.2

3.3 Infrastructure-As-A-Code

Infrastructure-as-a-code (IAAC) is an approach to automate the deployment and configuration of infrastructure. It uses principles from the domain of Software Development to

automate repeated procedures in infrastructures [42].

To better understand IAAC, let us assume Alice wants to set up a server for her new website. Without IAAC, she would connect to the server, set up users, set permissions, and install software components like a web server or a database. If Alice wants to set up another server for her second website, the whole process of the server configuration needs to be repeated. Doing that manually is error-prone, time-consuming, and inconsistent. The probability is high that she installs different versions of software components that might behave differently.

This is where IAAC comes in. Alice could write a script describing all procedures that need to be executed to set up the server completely. This script can then be run on an arbitrary server to set up a web server for a website. Hence, with IAAC, a server configuration can easily be reproduced.

Within a DevOps infrastructure, IAAC can be used to set up all different components. Such an approach enables the reproducibility of the whole infrastructure. For example, source code management systems, CI/CD pipelines, and deployment servers can be uniformly set up and configured. In addition, with the reproducibility of IAAC, the components can easily be scaled according to the workload.

3.4 Containerized Applications

The famous tagline ”Build Once, Deploy Anywhere” [45] already highlights one of the greatest advantages of containerized applications. The idea is to bundle an application containing all its dependencies centrally and then to run it isolated from other processes on an arbitrary server [49]. With this approach, it is possible to run different isolated applications on the same host server without interfering with each other.

Let us assume Bob wants to deploy his new web service. Without containerization, he would set up a server by installing all needed dependencies like a database and a runtime environment for his application. Bob needs to take care that he installs the right versions of the dependencies on the server to run his application properly. If he wants to deploy another application on the same server, he could potentially run into dependency problems as both applications might use the same packages but different versions.

If Bob uses containerization, on the other hand, he will set up his own CI/CD server, which is responsible for building his containerized applications containing all necessary dependencies. His deployment server would then run the different isolated containers. Thus, dependency problems are solved, and Bob can be sure that he will not run into dependency version problems.

3.4.1 Container vs. Virtual Machine

Containers and Virtual Machines (VM) have one common property. They both try to isolate processes from each other such that they do not interfere with each other [48].

However, it is crucial to understand that they work differently to achieve a certain isolation level. Containers offer weaker isolation than VMs by design [49]. To understand this statement, let's compare the two concepts.

Considering the right architecture on figure 3.2, VMs use a hypervisor on top of an infrastructure (e.q. a physical server) which splits up hardware resources and provides guest kernels to guest operating systems. These operating systems are run in isolated environments, called VMs. Processes from App A are only visible on its underlying guest operating system. The operating system where App B is running on can not see any processes from App A.

In a containerized environment (left side of figure 3.2), the host operating system directly runs on the infrastructure (e.q. physical server or VM). A container system (like Docker) is installed on this operating system. The containerized applications are then run on the container system. The container system is responsible for isolating running applications. It uses Linux cgroups to control resources, Linux namespaces to limit the universe of visible process ids for a container, and changing of the root to control the visibility of the host file system for a container [49]. As such, a root user on the host operating system can see all processes running on it. Hence, processes from App A or App B are visible for this user.

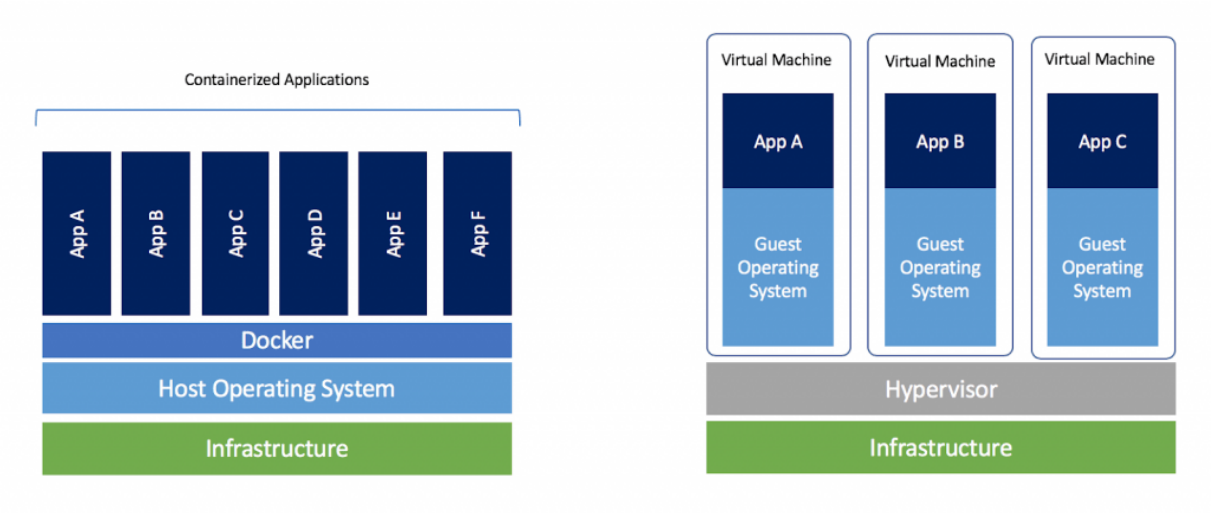

Figure 3.2: Container versus Virtual Machines [22]

Consequently, containerized applications share the same kernel in contrast to VMs. Since a hypervisor has a much simpler job to do in separating virtual kernels than a kernel in separating processes, the isolation in VMs is stronger than in containers [49].

Although containers provide weaker isolation than VMs, they also have important advantages. Firstly, since containers share the same kernel, the overhead is less than in VMs. Containers are therefore more efficient in using resources and performance. Secondly, VMs have long start-up times compared to containers as they need to install a whole operating system at start-up. Thirdly, VMs have fixed allocated resources where resources can be dynamically allocated to containers [49].

3.4.2 Container Security

Containers have less strong isolation between processes than VMs. However, to securely run containerized applications, it is crucial that these applications can not influence each other. For example, let us imagine a case where the isolation between two containerized applications is weak. An adversary that gains control over the first application could potentially attack the second application or even the host machine by breaking out of the container. Therefore, it is of the utmost importance to take countermeasures to container vulnerabilities when running a container environment. Section 5.1 provides a container security analysis for a whole CI/CD environment.

3.4.3 Docker

Docker is an open-source project that provides a lightweight environment to deploy applications into containers [48]. For that, Docker runs a container system and provides all the necessary tools to deploy containers easily. The following list summarizes the most important concepts of Docker.

- Docker image: A docker image is similar to a VM's image. It is a binary in which all the necessary dependencies are already installed, configured, and tested [6]. A docker image can then be shipped to a deployment server which can run a container out of it.
- Dockerfile: A Dockerfile is a script file containing instructions on building a Docker image. With the help of this file, it is easy to rebuild a Docker image from scratch [6].
- Docker registry: A Docker registry provides binary storage for Docker images. It contains hash-based versioning for the images [6]. Deployment servers can pull images from the Docker registry and deploy them as containers.
- Docker container: A Docker container is a deployed docker image. It is a running container on the host operating system.
- Docker daemon: The Docker daemon is a process on the host operating system that does all the work of Docker. It executes all docker processes like building an image or running a container. The Docker daemon has root access on the host operating system [49].
- Docker socket: The Docker socket is the interface to control the Docker daemon [49]. Commands like building an image or running a container can be sent to the socket. Whoever has access to the Docker socket can control the Docker daemon and hence has root access to the host operating system [49].
- Docker CLI: The Docker CLI can be used to request the Docker socket. It provides functionalities like building, running, or tagging images.

3.4.4 Docker-compose

Docker-compose is a tool to deploy multi-container applications [1]. The different containers, including their configurations, are defined in a YAML[2] file. Important configurations include the Docker image used, volume mounts, exposed ports, and definition of environment variables within the container. The multi-container application can then be run by a single command using the YAML file.

3.5 Reproducibility

In science, it is essential that results from scientists can be validated, such that other scientists can draw conclusions. In computer science, simulation experiments are run on an information system (computer, server, etc.) and evaluated by the scientist. If other researchers want to reproduce the same experiment, the same results should be observed again later on. Since conditions like information systems and software dependencies rapidly change over time, the probability is high that an experiment can not be reproduced later. Therefore, scientists should take care of a good reproducibility of their software package when providing them along with a research article [46].

As reproducibility is an important concept in science, a precise definition of it is crucial. Within this thesis, the definition of Goodman et al. [24] is used:

- "Methods reproducibility: provide sufficient detail about procedures and data so that the same procedures could be exactly repeated."
- "Results reproducibility: obtain the same results from an independent study with procedures as closely matched to the original study as possible."
- "Inferential reproducibility: draw the same conclusions from either an independent replication of a study or a reanalysis of the original study."

Combining the sections before, a CI/CD infrastructure deployed by IAAC, including containerized applications, could provide a high level of methods, results, and inferential reproducibility. With IAAC, procedures could be defined to duplicate a whole CI/CD infrastructure. The CI/CD infrastructure can then build containerized applications that contain all dependencies (including versions). As such, the provided code can independently be run by anyone, reproducing the whole infrastructure. Hence, procedures can exactly be repeated (methods reproducibility), independent studies have full documentation (code) on how to reproduce the original study (results reproducibility), and the same conclusion can be drawn in independent replications or from a reanalysis (inferential reproducibility).

[1] https://docs.docker.com/compose/
[2] https://en.wikipedia.org/wiki/YAML

3.6 Distributed Ledger Technology

A distributed ledger (DL) is an append-only data structure that is shared among different nodes in a network [40]. The ledger is a cryptographic backward linked list containing information in the form of transactions in blocks from the network. Each block in a DL contains a list of transactions and the hash value from the former block. Moreover, blocks can only be appended to the ledger. Transactions can have arbitrary information (e.q financial transactions, smart contracts, etc.). This ledger is then distributed and duplicated among all the nodes in a peer-to-peer network. Nodes running the network can sign new transactions, which will be included in a block and appended to the ledger. As there is no central authority in place, nodes need to agree on the ledger's state. To come to such an agreement, a consensus mechanism within the network is used. The most famous consensus mechanisms are: Proof of Work (PoW), Proof of Stake (PoS), Proof of Authority (PoA), or Practical Byzantine Fault Tolerance (PBFT) [60].

The protocol of a DL can be run within a private or a public network and can be permissioned or permissionless. In a public permissionless DL, everyone can run a node that can view the DL and write on it. Often PoW or PoS are used as a consensus mechanism [60]. In a public permissioned DL, everyone can read the state of the DL, but only selected nodes are allowed to write on it. In such a setting, mainly PoA or PBFT consensus mechanisms are employed. In a private DL, only selected participants are allowed to view and write on the DL [47].

Within the context of REV systems, a DL can be used as a decentralized PBB. The properties of a PBB to be append-only, publicly viewable, and restricted write access can be best provided by a public permissiond DL. By design, every DL provides an append-only data structure. Everyone can view a public DB, hence, voters or external parties can verify votes, proofs, and results on the PBB. As a centralized authority normally runs an election, only selected nodes need to have write access to the DL, and a consensus mechanism like PoA or PBFT can be used. In addition, a DL provides a high level of resilience due to its decentralized nature and the use of a consensus mechanism.

Chapter 4

Related Work

As current academic implementations about REV systems mostly focus on the cryptographic voting protocol and trying to provide as much privacy and verifiability as possible, the secure and usable deployment of such a system is missed out often. However, country-based or commercial implementations need to run a REV system in production and therefore need to design a secure system architecture. As commercial implementations normally do not expose their system architecture, country-based implementations normally publish their system architecture to provide a high level of transparency.

The following subsections provide an overview of the system architecture of two country-based REV systems in Switzerland, namely the CHVote [25] and the Swiss Post E-Voting system [56].

4.1 System Architecture CHVote

This description of the system architecture of CHVote is based on [59].

The CHVote architecture is built up as an online and an offline system. The online system provides access for the voter to cast his votes. The offline system is used to decrypt ballots, count the votes, and print the voting material used to provide verifiability over postal mail.

The online system is provided as a centralized deployment, meaning that each software component is run in a classical client-server fashion. The voter can access the voting software over the internet. A DDOS protection software checks the request and discharges potential denial of service attacks in the first step. After that, a reverse-proxy redirects requests to the right CHVote internal service. The reverse proxy also acts as a web application firewall that analyzes HTTP packages and filters out vulnerable requests (like cross-site scripting or SQL injection attacks). Each internal service provides a front-end and back-end where the business logic is implemented (e.q. the PBB). The backend services are connected to internal databases to persist data. In addition, control components run the cryptographic voting protocol and are connected to the backend components through an asynchronous centralized message broker.

The online system infrastructure is based on the platform-as-a-service software OpenShift [1]. It provides a containerized environment including configuration management using IAAC, performance monitoring, server logs, and load-balancing.

The offline components are installed on cantonal local computers. The first component is used to decrypt ballots and to count them. The second component decrypts the printing material and prints them. The voting material is then sent to the voter to provide end-to-end verifiability.

4.2 System Architecture Swiss Post E-Voting

This description of the system architecture of the Swiss Post E-Voting is based on [56].

The Swiss Post E-Voting system is a centralized online voting system that provides access for the voter and administrator over the internet. The deployment is fully separated for each canton, meaning each canton has a redundant test and production environment. Moreover, data centers are geographically distributed among Switzerland.

When voters or administrators access the voting application, their request passes a firewall, a DDOS protection software, and finally reaches the physical reverse proxy. Depending on if the request comes from a voter or an administrator, it gets redirected to the right internal front-end or back-end. This connection is encrypted using SSL, likewise the connection between the voters or administrators client and the reverse proxy. Internal services provide the implementation of business logic and are connected to an internal database. Cryptographic operations (like mixing, deception, etc.) are done using a secure data manager on a local computer offline and disconnected from a network.

[1] https://www.openshift.com/

Chapter 5

System Design

This chapter starts with a container security threat model. The model is then used to develop the CI/CD infrastructure design and the deployment of Provotum 3.0.

5.1 Container Security Threat Model

Containerized Applications gained high popularity in the last few years. It was the launch of Dockers simple CLI, which boosted the adoption of the technology within the developer's community [49]. With Docker, it was possible to build an image containing the application and all its dependencies and run it as a container uniformly and isolated on any infrastructure. Additional tools like the Docker registry made it possible to store versioned images on a centralized server and provide them to deployment servers. Developers realized that Docker could easily be included in their CI/CD infrastructure by building Docker images on a centralized build server and store them on a Docker registry. Deployment servers were able to pull these images and start them.

Because applications in a container run in a different way than in traditional deployment, potential attack vectors do change from a security point of view. One example of that shows the execution of processes within a container. If not specified, a process within a container is by default run as a root user on the host machine. If an adversary takes control over such a process, he might gain root access to the host machine and break the isolation of the container. This example shows the importance of analyzing the potential threats occurring in a containerized environment when designing such an infrastructure. Focusing on deploying a REV system, it is indispensable to analyze these risks to provide a secure deployment.

The analyzed threats in this section are divided into different steps of the supply chain of an image and its container. Figure 5.1 shows the steps of the supply chain. It starts with the Dockerfile, which defines how a Docker image is built. It is followed by the build machine, which is responsible for building a Docker image with the instructions of the Dockerfile. At the third step of the supply chain, the Docker registry is placed to version and store the Docker images. The last step contains the deployment of a Docker container

from a Docker image. The following threads and their mitigations are mainly adopted from [49].

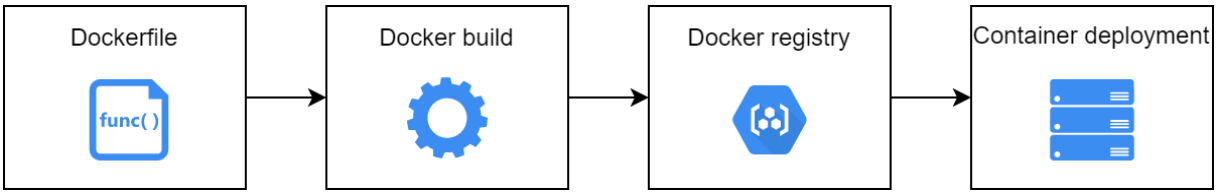

Figure 5.1: Supply Chain of a Docker image

5.1.1 Dockerfile

The Dockerfile builds the first step of the Docker supply chain. Within a Dockerfile, commands are defined on how to build the Docker image. If not configured properly, the resulting image could include malicious code, sensitive data could be exposed, or isolation could be easily broken. Table 5.1 shows the identified Dockerfile threats and possible mitigations.

ID	Title	Threat	Mitigation
DF1	Malicious code in base image	Every new Docker build origins from a base image. Since foreign software is injected from that base image to the project, malicious code could be included in the built image.	Only use official base images from trusted registries. Use base images that are as small as possible to reduce the surface of the attack.
DF2	Unnecessary packages	It often happens that too many unnecessary packages are installed through the Dockerfile that are not necessary to run the software. These packages might have been used to build the project but are not necessary at run-time. These unnecessary packages could have malicious or vulnerable code.	Consider using multi-stage builds. In the first step build your applications with all the necessary packages and in the second stage only copy the necessary packages that are needed at run-time.
DF3	Root user execution	When no specific USER instruction is defined in the Dockerfile, the image is run as root by default. This user might have too much right that is not necessary.	Define a user considering the least privilege principle.

DF4	Edit permissions	If a Dockerfile does not have clean access controls, malicious code could be inserted into it. A good example is the RUN command in the Dockerfile. It allows the execution of arbitrary code after the start of the container.	Manage right accesses for the Dockerfile in your code repository
DF5	Mounting volumes	If sensitive volumes are mounted in the Dockerfile, the container could gain access to sensitive code on the host machine. The worst scenario is if the root folder of the host machine is mounted as a volume. In this case, the container has full access to the host machine's file system.	Do not define mounts in the Dockerfile. Define volume mounts at runtime.
DF6	Sensitive Data	When sensitive data is included in the Dockerfile, it might appear in the source control, in the build pipeline, or the image registry. An adversary could easily gain access to these sensitive data.	Never define sensitive data in the Dockerfile. Use environment variables on a running system to pass sensitive data to a container.

Table 5.1: Dockerfile Threats

5.1.2 Build Machine

A containerized application is built within a build machine. If a build machine is compromised, a different build could be executed, and malicious code could be included. Hence, countermeasures must be taken to protect the build machine from adversaries. Table 5.2 presents the most important build machine threats and their mitigations.

ID	Title	Threat	Mitigation

BM1	Docker Daemon privileges	If an adversary receives access to the build machine and the docker daemon (that is used to build images in the pipeline) has too many privileges (e.q. root access), the adversary automatically has the same privileges as the docker daemon (root) since he can execute arbitrary code over the docker daemon	Consider using non-privileged builds (e.q. using podman [1])
BM2	Manipulating the build	The pipeline could include malicious commands or build steps that would result in a malicious image that might even be trusted by other components like the image registry or the deployment server.	Harden the build machine. Do not install unnecessary tools, keep components up-to-date and restrict the user access to the machine by employing IP-Blocks and firewalls.
BM3	Multiple components	When the build machine is hosted on the same server as other infrastructural components (like the image registry), the isolation between the components could be too weak. In combination with BM1, an adversary could even influence other infrastructural components.	Run a build machine on a server (bare metal or VM) isolated from other infrastructural components.

Table 5.2: Build build Threats

5.1.3 Docker Registry

After building a Docker image at the build machine, it is stored in a Docker registry. The registry provides a hub such that deployment servers can pull the images from it. Moreover, the registry needs to control the permissions, the versioning of the images, and check for vulnerabilities within an image. From a security perspective, the registry is an integral part of ensuring image integrity. Countermeasures should be taken to ensure

[1] https://podman.io/

that only trusted images can be stored and pulled from the registry. Table 5.3 shows the Docker registry threats and their mitigations.

ID	Title	Threat	Mitigation
IS1	Database exposure	Image registries usually use databases to store images and their metadata. If these databases are exposed to the internet, an adversary could gain access to it only with credentials	Do not expose databases to the internet. Only make images accessible through an authenticated channel.
IS2	New occurring vulnerabilities	If the image registry does not periodically scan the stored images for recently found vulnerabilities old images might include malicious code that would not be discovered	Run a vulnerability scanner that periodically scans your images and synchronizes with a public vulnerability database like CVE [2].
IS3	Unsigned Images	When images are not signed during the build process, the origin of the images can not be assured in the image registry. Due to a man-in-the-middle (MITM) attack, an adversary could inject a malicious image into the registry.	Sign images at the build machine and check the signature at the registry.

Table 5.3: Docker registry

5.1.4 Container Deployment

At the last step of the supply chain, the container gets deployed on a deployment server. The deployment server should be properly configured, the right images should be pulled, and admission controls should be enforced. Table 5.4 explains the threats and mitigations for the container deployment.

ID	Title	Threat	Mitigation

[2]https://cve.mitre.org/

CR1	Wrong Image Deployment	Unsigned images and insufficient tag policies could lead to the deployment of a wrong image. If the image is not signed by the pipeline, it can not be guaranteed that the image was not modified by an adversary after the pipeline. If the tagging policy is weak (e.q. no semantic tagging, only using the latest tag, etc.) it is not clear which image gets deployed since tags can be overwritten during the build process.	Sign images at build time and use the semantic versioning functionality of the Docker registry to ensure the right deployment of an image.
CR2	Vulnerable Deployment Configuration	Configurations of container orchestration tools like docker-compose can contain severe vulnerabilities. For example, unwanted host volumes could be mounted into the container or the container could be run with a privileged flag, which gives the container access to all Linux capabilities.	Ensure to only use the privileged flag in rare, special cases. Care about the volumes you mount from the file system. Try avoiding mounting the Docker socket into the container. Never mount a file system from the host into the container that is not needed.
CR3	No Deployment Checks	All vulnerability checks for the image are useless if they are not checked before deployment. If no admission control is done before the deployment, an image with detected vulnerable properties could be deployed.	Enforce deployment checks like checking the signature, check passed through vulnerability scans, or restricted container user permissions.

Table 5.4: Container deployment Threats

5.2 CI/CD Infrastructure Design

This section presents the CI/CD infrastructure design, which is used to deploy all different components of Provotum 3.0. The design of the infrastructure was created with the

following core principles:

- The instances should be set up in cloud infrastructure and configured with IAAC.
- All infrastructural components should be uniformly deployed as Docker containers.
- The threat model of section 5.1 should be considered, and the proposed mitigations should be implemented.
- All connections between the instances should provide TLS for a secure data transfer.
- The infrastructure should be used for future deployments of Provotum.

The core principles are essential to provide a secure and reproducible infrastructure for a decentralized REV system. By having the whole configuration as code, an external party should be able to deploy each component easily and use the code as a starting point for deploying another REV system.

The rest of this section provides an overview of the CI/CD infrastructure, explains the main components in detail, and presents the pipeline for deploying an application.

5.2.1 Overview

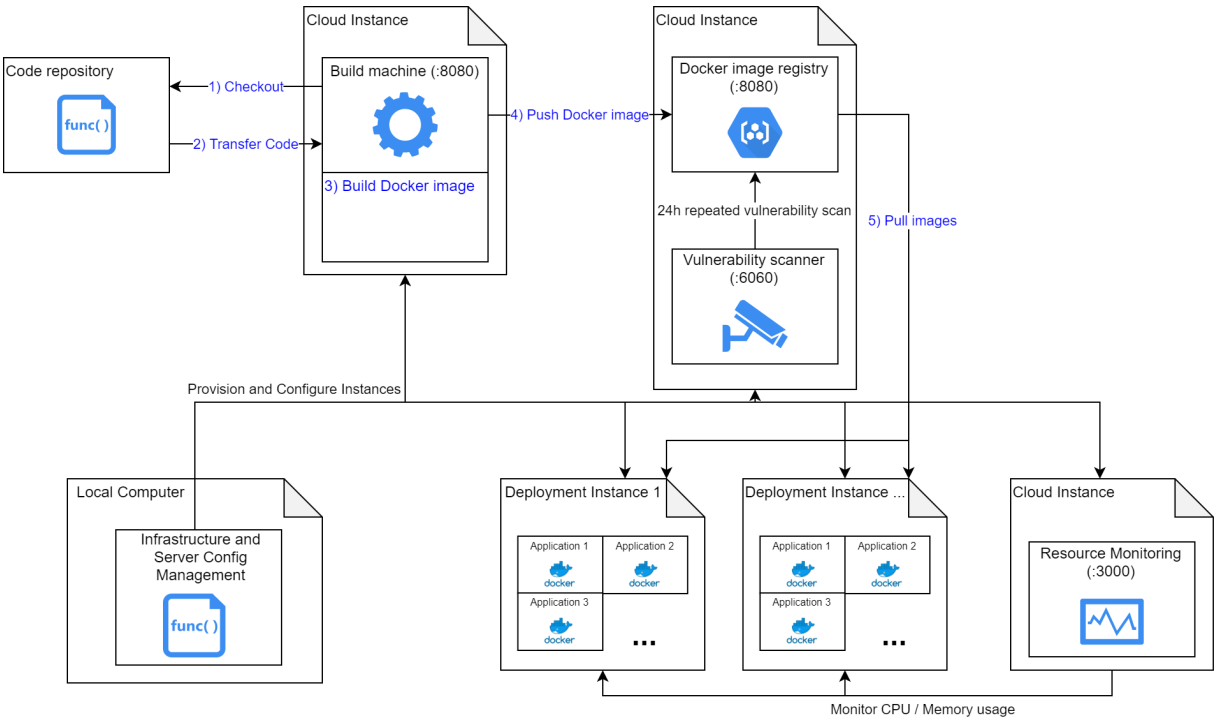

Figure 5.2: Overview of the whole CI/CD infrastructure and their components

Figure 5.2 presents the overview of the CI/CD infrastructure design. The documents represent cloud instances or the local computer. The arrows with labels in blue represent

a usual CD flow. The arrows with labels in black show the interactions between the components which are not directly affected by the CD flow. Deployment instances have an arbitrary number depending on the configuration of a specific deployment. For example, if multiple sealers (authorities in the REV system) are deployed, a new deployment instance is created for each sealer. The applications on the deployment instances also have an arbitrary number depending on the use case.

The main concept behind the design is to build a Docker image from the code repository and store it on a Docker registry. The deployment instances can then authenticate to the Docker registry and pull the required image. Every component of the infrastructure can be provisioned and configured with IAAC at a local computer. The computer can clone a configuration code repository and has all the functionalities at hand to deploy the whole infrastructure.

In addition to the pipeline, a vulnerability scanner and a resource monitoring system are deployed. The vulnerability scanner periodically scans Docker images on the Docker registry, and the resource monitoring system stores CPU and memory usage of the deployed containers. This monitoring can then be used to run different real-world experiments for the deployed components.

5.2.2 Components

Code Repository

The code repository provides the source code for the build pipeline. All applications which will go through the build pipeline are stored in the code repository. Each application contains a Dockerfile with instructions on how to build itself. The Dockerfiles provide multi-stage builds, custom user execution and do not contain any sensitive data. In the first step of the multi-stage build, the application is built, and in the second step, the build and only necessary dependencies are installed within a lightweight base image.

Build Machine

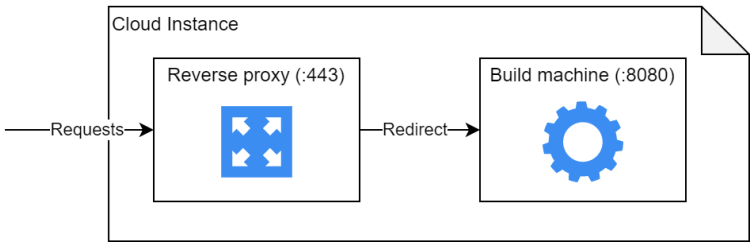

Figure 5.3: Architecture of the build machine instance

The build machine is responsible for building selected code from the code repository and storing them on a Docker registry. For security reasons, it was decided to deploy the build machine on a separate instance. This is since the build machine has access to the Docker

daemon and could therefore potentially influence other docker containers on the same instance. In addition, other software components were reduced to the bare minimum. Figure 5.3 shows the components installed on the instance of the build machine. Only a reverse proxy is used to secure the communication between the admin GUI and the build machine. With the admin GUI, it is possible to set connections to the code repository, configure the build pipeline, and deploy the resulting Docker image to the Docker registry.

Docker Registry and Vulnerability Scanner

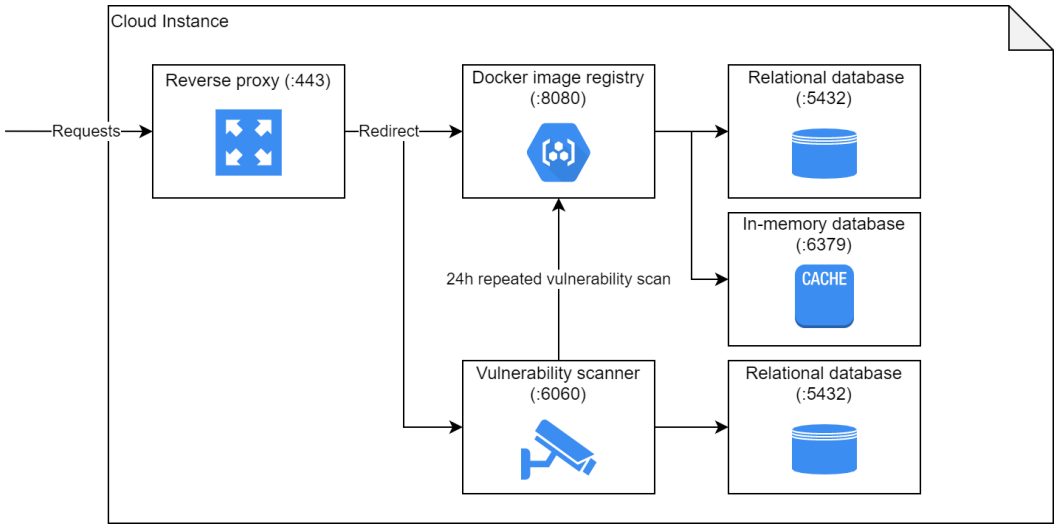

Figure 5.4: Architecture of the docker registry instance

The Docker registry stores and versions Docker images. In addition, a vulnerability scanner provides a periodic scan through all the Docker images stored in the registry. The instance contains a reverse proxy that accepts connections from the outside and redirects them to the right applications. The Docker registry exposes an admin GUI where information about stored images can be retrieved like vulnerabilities, versions, or push/pull histories. Moreover, the Docker registry can be accessed by the Docker CLI, with which images can be pushed or pulled. The Docker registry connects to a relational and in-memory database to store image information. The image binary is stored on the file system of the server.

The vulnerability scanner is directly accessed by the Docker registry and scans requested images. It synchronizes with the CVE database and indexes known published vulnerabilities in its own relational database.

All databases can only be accessed internally through the applications such that no exposures to the internet are provided.

Resource Monitoring

The resource monitoring intends to check CPU and memory usage of running Docker containers of the REV system (deployment instances). The usages are directly monitored

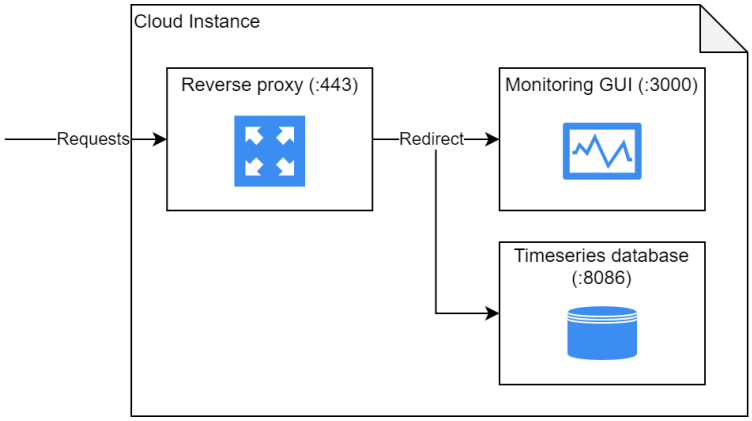

Figure 5.5: Architecture of the resource monitoring instance

at the deployment instances by accessing the Docker daemon and sent to the resource monitoring instance. The data is stored in a time-series database and can directly be accessed through a monitoring GUI.

Deployment Instance

The deployment instances can have a different system design depending on their use-cases. Section 5.3 presents the system design of the deployment of Provotum 3.0. They all have in common that they use a docker-compose file that defines the Docker containers deployed within the instance. The right images get pulled from the Docker registry and run with the defined configurations by running the docker-compose file.

Local Computer

The local computer is used to provision and configure the different infrastructural instances. A configuration code repository can be cloned to the local computer, and scripts can be run to set up all the different servers in the CI/CD infrastructure. The repository is built up such that scripts can be reused for different deployments. For example, the installation of Docker is defined just once and can be executed on different instances, if needed. This principle ensures that the configuration code repository can be easily extended later on if other instances need to be set up.

Every provisioning and configuration of a server is designed to execute the following steps:

1. Define the instances that should be provisioned by the cloud (type of server, resources, operating systems, etc.).
2. Call the API of the cloud to set up the instances.
3. Run a procedural script to install necessary software packages.
4. Copy a docker-compose file and all configurations needed for the applications to run.

5. Start the application by running the docker-compose file.

The first step defines the cloud instances that should be provisioned. The configuration of such an instance is dependent on the cloud provider used. In general, CPU cores, memory size, disk size, operating system, and location of the data center can be configured. In the second step, the configuration is sent to the API of the cloud provider, and the requested instance is provisioned. In the third step, the local computer connects to the provisioned instance and runs a script to set up necessary software packages like Docker. As every CI/CD infrastructure component is run within a Docker container (e.q. build machine or Docker registry), the fourth step can be generalized to only provide a docker-compose file to the instances. This docker-compose file can then be run in the fifth step. After all the steps, the instance should be fully configured, and the applications should be accessible.

5.2.3 Deployment Pipeline

After designing the CI/CD infrastructure, the deployment pipeline can be defined. The deployment pipeline builds a Docker image, stores it on a Docker registry, and deploys it on a deployment instance. Figure 5.2 shows the phases of the pipeline such that source code gets transformed into an executable binary. The following steps are passed:

1. Checkout code: The build pipeline clones the source code.
2. Transfer code: The source code is transferred from the code repository to the build machine.
3. Build Docker image: A Docker image is built in the build machine according to the Dockerfile provided.
4. Push Docker image: The Docker image is pushed onto the Docker registry.
5. Pull images: The Docker image can be pulled from the Docker registry to start a new Docker container.

5.3 Provotum 3.0 Deployment Design

Provotum 3.0 is a decentralized REV system developed by the Communication Systems Group (CSG) at the University of Zurich (UZH) [19]. The system supports a multi-authority scheme using distributed key generation and a randomizer to provide receipt-freeness. The PBB is provided by a Substrate[3] PoA DL. Moreover, a re-encryption mix-net is employed to disconnect voters from their ballots. Hence, elections can be provided in an arbitrary form.

This section focuses on the design of the distributed and configurable deployment of Provotum 3.0. The first subsection briefly presents an overview of the voting protocol.

[3]https://substrate.de

The second subsection explains the design of the deployment of the different application components. Finally, the last subsection illustrates extensions applied to the base implementation of Provotum 3.0 that were needed to deploy the application in a decentralized cloud environment.

5.3.1 Protocol

The protocol of Provotum 3.0 is divided into a setup, a voting, and a tallying phase. Before the setup phase of an election can be started, all authorities need to connect themselves in a peer-to-peer overlay network and start the DL. Different sealers are in charge of validating blocks in the PoA DL. A voting authority also connects to the network but only behaves as an observer of the DL.

Setup: After the protocol is started, the voting authority can set up a new vote providing a vote name and an election question. All sealers then need to create an ElGamal keypair and submit their public part and the proof of knowledge of the private part to the PBB. Afterward, the voting authority can combine the public key shares into a single public election key.

Voting: When the public election key is generated, voters can cast their votes. First, the ballots are encrypted with the public election key. Then, the ballots are sent to the randomizer, which re-encrypts the vote and creates proof of correct re-encryption. The voter is then able to cast the ballot to the PBB.

Tallying: When the ballot box is closed, the sealers start a re-encryption mix-net to shuffle the ballots. They also create a proof of correct shuffling and publish the proofs along with the shuffled ballots on the PBB. These ballots are then partly decrypted by each sealer and again stored on the PBB. The voting authority can finally combine the decrypted shares and tally the votes.

5.3.2 Deployment

Figure 5.6 presents an overview of the different servers deployed for Provotum 3.0. There exist two types of instances. One is a centralized server instance that runs the voting authority and the randomizer. Sealer instances represent the second category of instances. The server instance only exists once, whereas sealer instances can have an arbitrary number. They represent the multi-authority scheme and run a DL with a PoA consensus mechanism. For that, the sealer nodes and the voting authority node connect over a peer-to-peer network. The voting-authority node acts as a boot node, such that the sealers can observe themselves.

In addition, on each instance, a voting CLI is installed to execute functions of the voting protocol (e.q. start a new election).

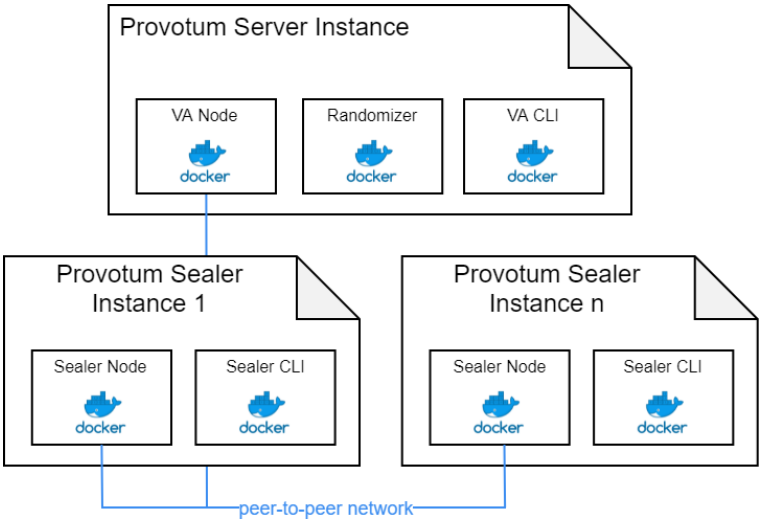

Figure 5.6: Architecture of the deployment of Provotum 3.0

5.3.3 Extensions

As the initial version of Provotum 3.0 was not intended to be run in a realistic distributed setting on different instances, some design changes had to be made. The design changes compared to the original version are described in Table 5.5.

Original Version	**Changes**
It was assumed that only three well-known accounts exist (Alice the voting authority, Bob and Charlie the sealers).	The protocol had to be changed such that multiple sealers could join the network and that they (including the voting authority) could create their own keys.
The test deployment started all applications on a single instance.	The test deployment was designed such that the centralized applications (voting authority and randomizer) were deployed on a single instance, and each sealer was deployed on its own instance.
The PoA DL was started in local mode.	The configuration of the DL had to be changed such that sealers on different instances could observe themselves.
The CLI to access the application only used well-known accounts.	The CLI had to be changed such that self-created accounts could be accessed with it.
The CLI was built as an executable binary.	The CLI binary was containerized that it could be built and stored within the CI/CD pipeline.

Table 5.5: Design changes compared to the original version of Provotum 3.0

The most important design changes were made because well-known accounts were used for the DL. Changes implied that each sealer could generate his own keys and connect to the peer-to-peer network over a boot node. This change opened the possibility to deploy multiple sealers in a decentralized architecture. The implementation details of the decentralized deployment of Provotum 3.0 can be found in section 6.2.

Chapter 6

Implementation

This chapter dives into the implementation of the deployment of the CI/CD infrastructure and the deployment of Provotum 3.0. First, the technologies used for the overall infrastructure are presented. Second, an example of a typical deployment procedure is shown. Third, the steps for the deployment of Provotum 3.0 are explained in detail. Additionally, technical limitations for the CI/CD infrastructure and the deployment of Provotum 3.0 are listed.

6.1 CI/CD Infrastructure Deployment

6.1.1 Technologies

As section 5.2 defined the core design principles for the deployment of each component of the CI/CD infrastructure, it is necessary to decide on an adequate technology stack to implement such requirements. The decision for the following technology stack was inspired by the related work on the architecture of the deployment of REV systems and the background information from chapter 3. The following list presents the technologies used for each component introduced in section 5.2.2:

- **Code repository**: GitHub[1]
- **Build machine**: Jenkins[2]
- **Docker image registry**: Project Quay[3]
- **Vulnerability scanner**: Clair[4]

[1] https://github.com/
[2] https://www.jenkins.io/
[3] https://www.projectquay.io/
[4] https://github.com/quay/clair

- **Local computer (IAAC)**: Terraform[5] (instance provisioning), Ansible[6] (instance configuration)
- **Resource monitoring**: Grafana[7], InfluxDB[8] Telegraf[9]
- **Reverse proxy**: Traefik[10]
- **Container applications, container orchestration**: Docker[11], docker-compose[12]
- **Cloud provider**: DigitalOcean[13]

6.1.2 Deployment Procedure

To deploy all components as IAAC in a cloud environment, the first step is always to provision new server instances. A Terraform script was defined for each component containing instructions for the cloud provider API to set up a new instance. Listing 1 shows an example of a script that sets up the cloud instance for Jenkins on DigitalOcean. The required resources are defined (lines 2-6), the ssh key to connect to the instance is set (lines 8-10), and a first command is provided that will be executed after the provisioning of the instance (lines 12-21). After the startup of the instance is done, an Ansible script is executed (lines 23-25) to install the required software packages (in this example, Jenkins).

The second step contains the configuration of the instance. Listing 2 shows the Ansible script that is executed right after the provisioning of the instance. The script executes different roles that contain tasks that are executed directly on the instance. In this example, Docker, docker-compose, Jenkins, and the reverse proxy are installed.

After these two steps, the instance is fully configured for the given use case. As every software runs in a Docker container, the Ansible script always copies a docker-compose file and necessary configuration files to the instance in the configuration phase. For example, in the previous case of Jenkins, a docker-compose file defining multiple configurations for Jenkins is copied to the instance and is run directly as a Docker container.

6.1.3 Technical Limitations

The deployment of the CI/CD infrastructure also led to some technical limitations. The most important limitation is that build pipelines in Jenkins need to be configured manually. In future work, this limitation could be resolved using Ansible to configure the

[5]https://www.terraform.io/
[6]https://www.ansible.com/
[7]https://grafana.com/
[8]https://www.influxdata.com/
[9]https://www.influxdata.com/time-series-platform/telegraf/
[10]https://traefik.io/traefik/
[11]https://www.docker.com/
[12]https://docs.docker.com/compose/
[13]https://www.digitalocean.com/

```
resource "digitalocean_droplet" "jenkins" {
  count  = 1
  image  = "ubuntu-20-04-x64"
  name   = "jenkins"
  region = "fra1"
  size   = "s-2vcpu-4gb"

  ssh_keys = [
      data.digitalocean_ssh_key.terraform.id
  ]

  provisioner "remote-exec" {
    inline = ["sudo apt update", "echo Done!"]

    connection {
      host        = self.ipv4_address
      type        = "ssh"
      user        = "root"
      private_key = file(var.pvt_key)
    }
  }

  provisioner "local-exec" {
    command = "ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook -u root
    ↪  -i '${self.ipv4_address},' --private-key ${var.pvt_key}'
    ↪  ../../ansible/jenkins.yml"
  }
}
```

Listing 1: Example of a Terraform script

different build pipelines fully. Another minor limitation presents the static task scripts in Ansible for Ubuntu. All Ansible scripts only work on an Ubuntu 18.04+ distribution. These scripts could be executed dynamically dependent on the underlying operating system.

6.2 Provotum 3.0 Deployment

The deployment of Provotum 3.0 followed the same principle as the deployment of other components in the whole infrastructure (like Jenkins). However, some steps needed to be added as server and sealer instances in the Provotum 3.0 design depend on each other. To start the DL, the instances need to share a common custom chain specification file. This file contains information about the initial block in the DL, configuration about the consensus mechanism, the sealer's public validation and finalization address, and the URL for the boot node. For this reason, after the configuration of the different instances, this

```
---

- name: Set up a Jenkins CI server
  hosts: all
  become: yes
  vars:
    - update_apt_cache: yes

  roles:
    - docker
    - docker-compose
    - jenkins-inst
    - role: reverse-proxy
      vars:
        network: files_provotum-jenkins
```

Listing 2: Example of an Ansible script

configuration file needed to be generated and shared among all instances before starting up the peer-to-peer protocol.

This section gives a deeper view into the different steps of the deployment of Provotum 3.0.

6.2.1 Building the Image

Before Provotum 3.0 can be set up, the corresponding Docker images need to be available on the Docker registry. For that, different Dockerfiles are defined within the project of Provotum 3.0. There is a Dockerfile to build a node (voting-authority or sealer), build the randomizer, and build the voting CLI. These components then get built on Jenkins, and the resulting Docker image gets stored on Quay. When the Docker images are stored, they periodically get scanned for vulnerabilities by Clair. After storing the Docker images, they are ready to get pulled from deployment instances.

6.2.2 Provisioning

The provisioning of the different instances for Provotum 3.0 is done with Terraform. The scripts look similar to the one that was used to provision Jenkins in listing 1. One Terraform script is used to provision the server instance containing the voting-authority node, the voting-authority CLI, and the randomizer. Another script provisions the instances of the sealers. This script contains a counter to deploy as many sealer instances as wanted parallelly. After the provisioning, an Ansible script is executed to configure the instances and to generate the common custom chain specification file.

6.2.3 Configuration

When the instance is running, it gets automatically configured with an Ansible script. Depending on if it is a server or a sealer instance, another script gets executed. Since all components of Provotum 3.0 are run in Docker containers, Docker and docker-compose are installed in the first step. Next, validation and finalization keys and addresses are directly generated on the instances with a tool called subkey[14] which is provided by Substrate. Sealers need these keys to validate and finalize blocks in the DL. These keys must be generated locally on the instances and not on the local machine, as they should never be shared with anyone else.

After the key generation, the actual application is installed. For that, a docker-compose file is copied to the instances. The docker-compose file configures the different container applications. Listing 3 presents the docker-compose file for the sealer instance. All environment variables in the docker-compose file are defined in a separate .env file. Line 4-19 corresponds to the sealer application, and Line 21-26 corresponds to the CLI. By looking at the sealer application, we can see the definition of the name of the Docker image, which is stored on the Docker registry on line 6. Line 7-16 shows the command which is executed after the creation of the Docker container. These options correspond to the Substrate options which are needed to start the DL. Specifically, line 8 defines the path to the custom chain specification file. This file is not generated at this point in time as the local machine still needs to collect all public validation and finalization addresses from the sealers and combine them in that file.

In the next step, all instances save their public addresses on the local machine. Then they get centrally combined within the custom chain specification file. Afterward, the file is sent to the instances. At this point, the configuration of the instance is done, and the applications can be started. The sealers and the voting authority automatically connect over a peer-to-peer overlay network. The voting authority acts as a boot node such that the sealers have an entry point to the network. After the connection is made, the validation and finalization key is stored on each node application. With the keys, the sealers start to validate and finalize blocks. The DL is running and is waiting for incoming transactions to record.

6.2.4 The Command-Line Interface

To access the voting protocol, each instance is provided with a CLI to execute commands like generating a new election or casting new votes. The CLI differs between the server and the sealer instance. As the CLI on the server instance can only execute voting authority or voter-related commands, the CLI on the sealer instance can only provide sealer-related commands. To provide an easily accessible interface for all the different CLIs on the instances, an Ansible script is run on the local machine. If we want to start a new election, the ”start election script” can be executed, and Ansible automatically connects to the right instance and runs the right CLI command. The same holds for the sealer

[14] https://substrate.dev/docs/en/knowledgebase/integrate/subkey

```
version: "3.8"

services:
  sealer:
    container_name: sealer-${SEALER_NUMBER}
    image: ${NODE_IMAGE}
    command:
        --chain=./customSpec.json
        --bootnodes ${BOOTNODE}
        --validator
        --name sealer-${SEALER_NUMBER}
        --base-path /tmp/sealer-${SEALER_NUMBER}
        --port 30333
        --ws-port 9944
        --rpc-port 9933
        --execution Native
    network_mode: host
    volumes:
      - ./customSpec.json:/provotum/customSpec.json

  client:
    container_name: client
    image: ${CLIENT_IMAGE}
    environment:
      - VA_URL=${VA_URL}
      - SEALER_MNEMONIC=${SEALER_MNEMONIC}
```

Listing 3: docker-compose for a sealer instance

instances. When the ElGamal key pair needs to be generated for the election, Ansible connects to all sealer instances and executes the CLI command for this functionality. With these scripts, the whole election from setup over voting to tallying can be accessed and tested.

6.2.5 Monitoring

To test runtimes on the distributed deployment of Provotum 3.0, a monitoring system is used to track CPU and memory usages of the different Docker Containers. Telegraf is used as a containerized application to track CPU and memory usages from Docker stats. The data is then sent to the centralized monitoring instance. The influxDB saves the data in time series, and Grafana provides a web application to present the data in a human-readable way. Figure 6.1 shows an example of the CPU usage monitoring during an election.

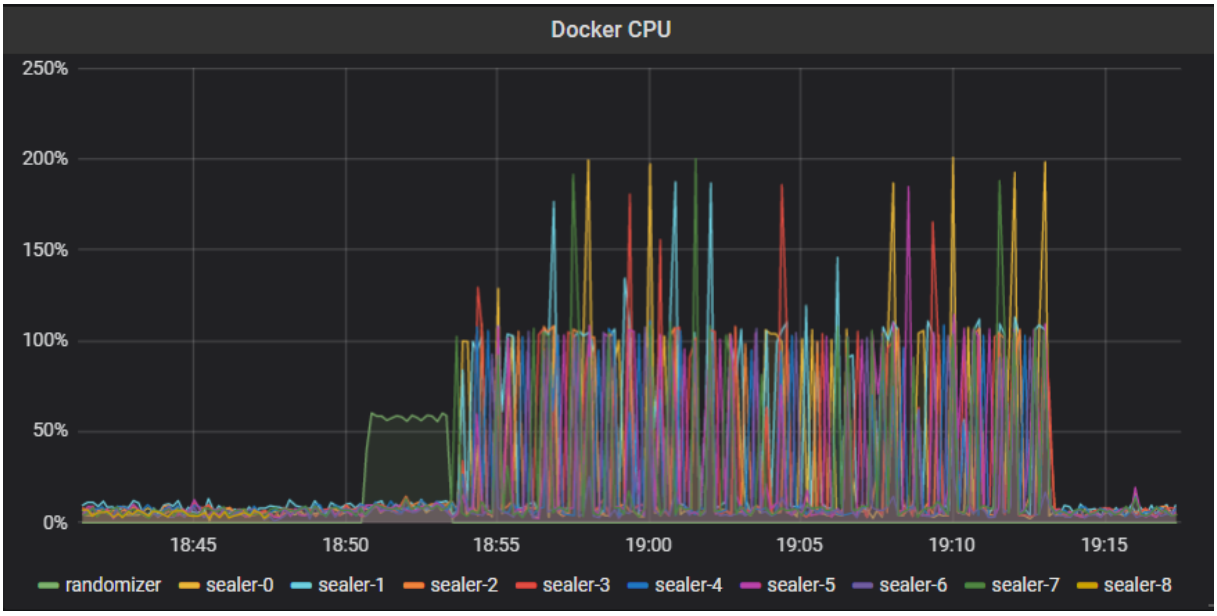

Figure 6.1: Grafana GUI to monitor CPU usages of Docker containers.

6.2.6 Technical Limitations

The deployment of Provotum 3.0 contains only minor technical limitations. However, some aspects of the implementation could be improved. Firstly, the subkey binary is stored in the code repository and then copied from the local machine to the instance. This binary should be built in the CI/CD pipeline and stored on binary file storage. It could then be downloaded directly from the instance. With this improvement, the subkey binary can be easily updated, rebuilt, and deployed. Secondly, the additional step with generating the custom chain specification file needs to be done by user interaction on the local machine. If it were possible to wait until all instances would be provisioned and configured, the custom chain specification file generation could be done automatically after that.

Chapter 7

Evaluation

The evaluation of the whole CI/CD infrastructure and the deployment of the prototype of Provotum 3.0 is divided into two parts. First, the defined core principles for the infrastructure are evaluated. Second, the distributed deployment of Provotum 3.0 are analyzed according to security, scalability, and usability properties.

7.1 Core Principles

As the overarching goal was to provide a modular, reproducible deployment and configuration management system for Provotum 3.0, core principles were defined in section 5.2. As shown in figure 7.1, these principles should enforce a usable, reproducible, and secure deployment for a REV system. Therefore, the subsequent evaluation focuses on the fulfillment of these core principles.

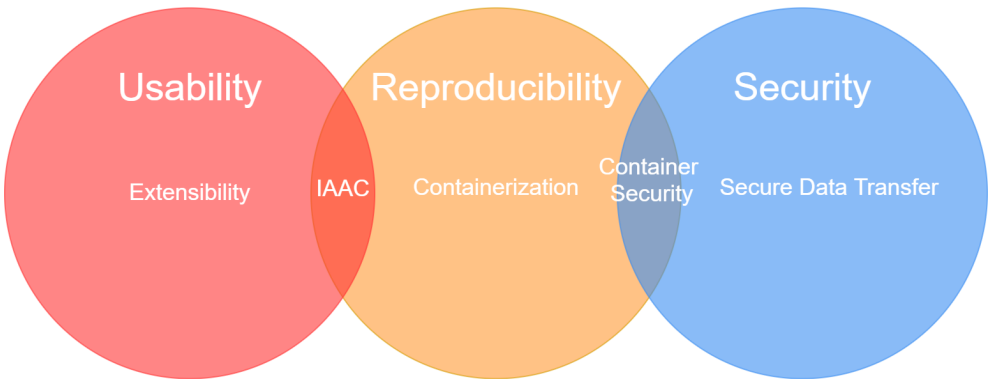

Figure 7.1: Core principles for the CI/CD infrastructure

7.1.1 IAAC and Containerization

According to IAAC and containerization, the following core principles were defined:

- The instances should be set up in cloud infrastructure and configured with IAAC.
- All infrastructural components should be uniformly deployed as Docker containers.

The implementation clearly shows that all components of the CI/CD infrastructure and the deployment of Provotum 3.0 can all be provisioned with Terraform and configured with Ansible. These frameworks provide modular scripts to automate the setup of all software components. Hence, the prototype of deploying the whole infrastructure for a REV system fulfills the first core principle.

By looking at the implementation of the prototype, all components are deployed as Docker containers. For each component, the Ansible script copies a docker-compose file to the instance, which defines a multi-container application. This file is then used to run all applications on the instance. For this reason, the second core principle is also fulfilled.

7.1.2 Container Security

The third core principle emphasis the security of a containerized environment and states the following:

- The threat model of section 5.1 should be considered and the proposed mitigations should be implemented.

The threat model was divided into four important steps of the supply chain of an image and its container. These steps were the Dockerfile, the build machine, the Docker registry, and the container deployment.

Dockerfile

Starting with the Dockerfile, we can see that in Provotum 3.0 deployment, only small and official base images were used, like the Ubuntu or the Debian base image. The Dockerfiles also provide multi-stage builds. In the first step, the application's binary is built and then copied into a smaller image used during runtime. In addition, a user is defined to not execute commands on behalf of the root user. Finally, no sensitive data is stored in the Dockerfile. As Provotum is still in the development stage, all users from the Provotum organization have access to the Dockerfile. Access control of the Dockerfile would only make sense in a productional environment and was not considered in the prototype.

Build Machine

Next, by looking at the build machine, we realize that it is deployed on a cloud instance isolated from other components. This is important because the range of influence of an adversary is restricted if he gains access to the build machine. Moreover, only necessary

software to run the build machine is installed on this instance. The only Docker container (next to the container of the build machine) that is run on the instance is a Traefik reverse proxy to provide TLS and to redirect requests to the build machine. As the hardening of the build machine is strong, non-privileged builds were not considered. The build machine directly uses the Docker daemon to build images in the pipeline.

Docker Registry

The next step of the supply chain is the Docker registry. The Docker registry provides an authenticated access over TLS. The databases are not exposed to the internet and can only be requested from an internal network. Additionally, a vulnerability scanner is installed to scan the Docker images periodically. It also synchronizes its database with the CVE database. Due to the complexity of creating a signature for the image during the build, this feature was not implemented. In a productional environment, the signature of the image would be an important feature to enhance integrity.

Container Deployment

Lastly, the deployment of the container needs to be analyzed. Only necessary volumes were mounted from the filesystem like the custom chain specification file for all the deployments. For Jenkins and the reverse proxy, the Docker socket needed to be mounted to provide the Docker build functionality (for Jenkins) and the observation of recently started containers (reverse proxy). The privileged flag was only used for the Telegraf container such that it can directly read Docker stats. This flag should be used carefully as it provides all Linux capabilities to a container. In a productional environment, this should be analyzed and perhaps not be used. As no signatures were created for the Docker images, no deployment checks were employed. When providing signatures, it is also important to enforce these deployment checks.

7.1.3 Secure Data Transfer

The fourth core principle was related to secure data transfer and was stated as follow:

- All connections between the instances should provide TLS for a secure data transfer.

To achieve end-to-end encryption between different components, a reverse proxy was used on each instance. The reverse proxies listen to virtual hosts and redirect the requests to the right internal application. Certificates for the virtual hosts are resolved using the Let's Encrypt certificate authority[1]. By using the certificates, a secure TLS connection between components can be established. For that reason, the principle is fulfilled for the CI/CD infrastructure and the deployed application. It is important to note that requests

[1] https://letsencrypt.org/

are only encrypted until reaching the reverse proxy. Internal encryption (i.e., redirects from the reverse proxy to the application) is not considered and could be extended in a productional environment using self-signed certificates.

7.1.4 Extensibility

The fifth and last core principle according to the extensibility of the infrastructure was stated as follows:

- The infrastructure should be used for future deployments of Provotum.

As it is difficult to predict which direction future development will take, extensibility is hard to evaluate. However, some important features were implemented to support the future development of the infrastructure. Firstly, all deployment scripts are published on a code repository and are documented in detail. Secondly, the scripts are modularly built up to be reused for another deployment. For example, the installation of Docker or the reverse proxy is defined in a script that can be used for any other deployment. These scripts also allow for parametrization such that they can be reused more easily. Thirdly, the structure of the deployment is the same for each component. Hence, this structure could be reused as an example or a template for another deployment.

7.2 Provotum Deployment

7.2.1 Overview

Next to Provotum 3.0, two other versions, Provotum 2.0 and ProvotumRF, were deployed using the same CI/CD infrastructure. The following table 7.1 presents the main deployment status of all three applications. Provotum 2.0 [20] was deployed on one cloud instance, using three fixed sealers, and is fully functional. This means that the application can be set up such that the voting protocol works out of the box. Next, we have also tried to deploy ProvotumRF [30] within the infrastructure. It provides a dynamic number of sealers that are deployed on distributed instances. However, the protocol is not yet fully functional as some implementation details are missing in the base protocol to provide such a cloud deployment. Finally, Provotum 3.0 was deployed, which provides distributed sealers on separate instances and is fully functional.

Protocol	**Decentralized Deployment**	**Fully Functional**
Provotum 2.0 [20]	✗	✓
ProvotumRF [30]	✓	✗
Provotum 3.0 [19]	✓	✓

Table 7.1: Deployed applications

The remainder of this section presents the evaluation of the deployment of Provotum 3.0.

7.2.2 Provotum 3.0 Deplyoment

The cloud deployment of Provotum 3.0 was evaluated in three different dimensions, namely security, scalability, and usability. The security evaluation was done according to possible container security threats, best practices from reference deployments for REV systems, implementation details of Provotum 3.0, and the underlying trust assumptions. Scalability was tested by measuring the duration of election processes with different parameterizations. Finally, usability was evaluated by the level of possible parametrization, the difficulty to set up such a cloud deployment, and consequently, the reproducibility of the system.

Security

From a container security point of view, the core principle to consider the container security threat model holds for the infrastructure and the Provotum 3.0 deployment. The reason for that is that Provotum 3.0 is built within the CI/CD pipeline of the infrastructure, and the Dockerfile was created regarding the Dockerfile container security mitigations.

By comparing the system design of other relevant REV systems like CHVote [59] or the Swiss Post E-Voting system [56] with the deployment of Provotum 3.0, we can see similarities but also differences. The most pertinent point is that in other REV systems, all communication between the client and the server is encrypted using TLS. Provotum 3.0 uses secure WebSocket connections for CLI requests from one node to another. If, for example, the CLI on the sealer instance requests the voting authority node, a WebSocket connection is established using TLS. Hence, also the deployment of Provotum 3.0 provides secure data transfer over the internet. Another interesting point to consider is the use of DDOS protection or web-application firewalls (WAFs) to filter malicious requests. The deployment of Provotum 3.0 does not consider these components as they can be easily added in a productional environment. For test purposes and the reproducibility of such a distributed system, these components only added a level of complexity and were therefore left out.

Provotum 3.0 uses a DL as the PBB, which specifically needs to be evaluated from a security point of view. Since only selected sealers are allowed to run the protocol of the distributed ledger, the system needs to ensure its integrity. The definition of the authorized sealers is contained in the custom chain specification file. Therefore, the creator of this file needs to be trusted that only valid sealers are allowed to run the protocol. From the implementation of the deployment of Provotum 3.0, we can see that the file is created on the local machine, which is responsible for the configuration of the infrastructure. Hence, this entity needs to be trusted, and in a productional environment, the local machine should therefore be run by a trusted voting authority.

Scalability

To analyze the scalability of the cloud deployment of Provotum 3.0, we need first to identify the bottlenecks of the voting protocol. As the original protocol [19] mainly identified the mixing (shuffle votes, generate and verify shuffle proofs) as the main stateful bottleneck, this evaluation focuses on the mixing step of the distributed deployment.

Experimental Setup: The following runtime tests were conducted on the distributed cloud deployment of Provotum 3.0. We have run tests with 2, 5, and 10 sealers in the network. Each instance was provisioned by DigitalOcean using 2 CPU cores and 2 GB of RAM. Ubuntu 20.04 was chosen as the operating system. In addition, we have fixed the block time to 6 seconds, the mixing batch size to 75, and the rounds of mixing to 3. The fixed block time describes the interval in which new blocks are generated in the DL. The batch size describes how many ballots are shuffled by a sealer in one operation. Each experiment was run 3 times, and the average runtime was calculated. The runtime is measured by the monitoring system of the infrastructure and contains the time from the start until the end of the mixing step. This means it is an end-to-end experiment that includes all steps necessary to mix the ballots. These steps include the shuffling, the generation, the verification of the shuffle proofs, and the complete interaction with the DL (storing, reading ballots, etc.).

Results: The following table 7.2 and figure 7.2 show the results obtained from the scalability experiment. The data is separated into series of 2, 5, and 10 sealers. The table presents the average runtime in seconds for the mixing step for different numbers of cast votes. The same data is plotted in figure 7.2. The axes are formatted in log scales. Hence, a linear graph represents a linear growth in the runtime when the number of votes increases.

Number of votes	2 sealers	5 sealers	10 sealers
100	99s	91s	111s
1'000	1'280s	1'580s	1'750s
10'000	10'002s	11'535s	13'405s

Table 7.2: Mixing times in the distributed deployment

Firstly, we can see that the runtime grows linearly in the number of votes for each data series. This is a good indication that the distributed system could be scaled for a large election. However, we can also see that the mixing for one vote takes approximately 1 second. Secondly, the increase of the runtime with additional sealers is small. The increase from 2 sealers to 5 sealers varies between -9% and 23%. The increase from 5 sealers to 10 sealers varies between 10% and 22%. These numbers, however, should be treated with caution. As they are end-to-end tests, a high variance can occur, for example, due to network delays. Additional results for the runtime of randomizing votes can be found in the appendix.

Discussion: The scalability experiments show some unexpected results. When comparing these results from the decentralized deployment to benchmark results from the original deployment of Provotum 3.0 [19], we can see that the mixing time in the cloud

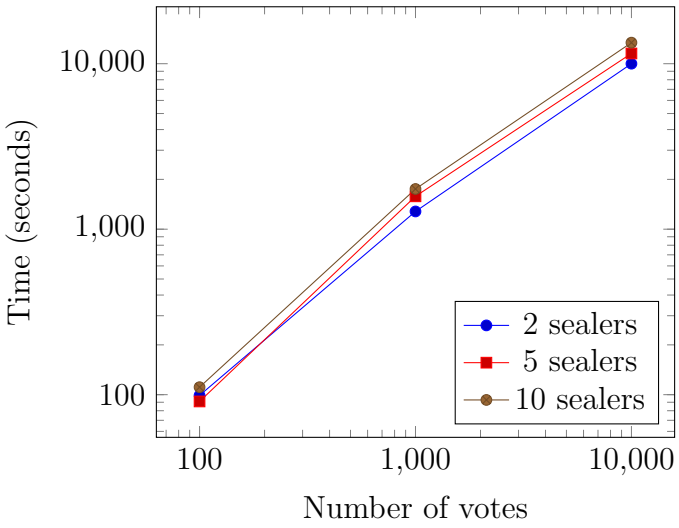

Figure 7.2: Mixing times in the distributed deployment

deployment is around 16 to 26 times slower than the mixing in the original work. The main reason for that lies in the end-to-end characteristics of this experiment. As the original work only measured the runtime of the shuffling, proof generation, and proof validation functions, the overhead from the DL was not considered. In this experiment, as the block time of the DL is 6 seconds, the sealers need to wait until transactions of the previous shuffling were committed. When the shuffling is fast, it will still take 6 seconds until the next sealer can start shuffling. To mitigate this problem, the block time or the batch size could be adjusted. A shorter block time would add transactions faster to the DL, and a larger batch size would increase the number of votes shuffled by a sealer in one transaction. We have tried to adjust those parameters to increase the performance of the mixnet. However, the change of both parameters caused errors during the shuffling process. Another surprising result is that the mixing runtime does not increase linearly with the number of sealers. The sealers should sequentially (one after another) shuffle the batch of ballots. This would implicate that a doubling of the sealers should cause a doubling of the runtime. The reason for that behavior is still unknown and a good starting point for future work.

The previous evaluation shows that the prototype is not ready yet to be tested in a distributed deployment. Some implementation issues still need to be fixed such that a realistic evaluation can be carried out.

Usablity

The usability aspect is evaluated according to the difficulty of how a user can achieve his goal by deploying the REV system. In addition, it is important that the user can parameterize the deployment such that she can adapt the application for a specific use case. The user here is the system admin of the voting authority who controls the local machine and deploys the whole application.

Starting with the principle of IAAC, we can see that all components are deployed using

pre-defined scripts. With that, installing different software components can be abstracted. The user has a much simpler task to do compared to manual provisioning and configuration of the server. With IAAC, it only takes a handful of commands to be executed until the instances are provisioned and configured in the cloud. Next, the design of the deployment focused on a high parametrization of the prototype. Many parameters were assigned with a default value but can be changed in another environment. A good example of that is the domain for the deployment. It is a variable that is predefined such that the whole system can easily be deployed. However, if a user wants to change the domain, she can change it at one point of the script, and all components get configurated with this new domain (including generation of the DNS entries). The most important parameter in the prototype of the deployment of Provotum 3.0, however, is the number of the sealers. This is by default set to two but can be changed to another arbitrary number. The instances are then dynamically set up according to this number. Finally, the different deployment scripts were designed in a modular way. We have used Ansible roles which define tasks to configure a defined software package. This modularity allows for a straightforward analysis of how the prototype works and a good starting point if the user wants to add new functionalities. All in all, the prototype for the distributed deployment of Provotum 3.0 achieves a high reproducibility, such that users can reproduce the exact infrastructure to test the REV system.

Chapter 8

Summary and Conclusions

As the reproducibility crisis also affects domains in computer science and specifically in the domain of remote electronic voting (REV) systems, the main goal of this thesis was to develop and deploy a reproducible cloud infrastructure for such a REV system. In the first step, a continuous integration/continuous delivery (CI/CD) infrastructure was set up to securely build Docker images from the different components of the REV system. Then, a configuration management system was used to automatically provision and configure the deployment instances. As such, the whole deployment could be provided as infrastructure-as-a-code (IAAC) and can therefore be used to reproduce the deployment of Provotum 3.0.

The prototype successfully fulfills the formal definition of reproducibility from section 3.5. The definition states methods, results, and inferential reproducibility. All of these requirements can be fulfilled mainly due to two aspects of the infrastructure. Firstly, all components (infrastructural and application) are defined in Docker images and deployed as Docker containers. With that, dependent software can be installed within a containerized environment and can be run everywhere. Secondly, the whole infrastructure and application are provided as IAAC. This allows to reproduce and configure each cloud instance. With these ideas in mind, methods can be easily reproduced by just executing the code to set up the infrastructure. Docker images can be pulled from a docker registry and directly deployed. By having an identical infrastructure and application deployment, the results of each experiment should also be reproducible. And, if the results are the same, the inferential analysis would also draw the same conclusion.

Nonetheless, during the process to provide such a cloud deployment, also difficulties arose. The many difficulties were to deploy the prototype of a decentralized REV system in a distributed environment, which had not been intended by the prototype's design. It led to some detailed adjustments to the implementation of the REV system, which required in-depth knowledge of the protocol. With this acquired knowledge, prototypes could be designed with a distributed deployment in mind in the future. As such, it would dramatically reduce the complexity of the deployment step. Another difficulty was to design the deployment scripts modularly. However, the work was worth it, such that anyone can reuse these scripts for future deployments.

8.1 Future Work

This section suggests a starting point for future work in the field of the deployment of a decentralized REV system. It provides possible improvements in the REV system prototype, the security, and the usability of the deployment. These suggestions aim to achieve a productional deployment of a decentralized REV system in the future.

8.1.1 Improve Provotum 3.0 Implementations

Section 7.2.2 identified some implementation issues with the prototype of Provotum 3.0 for the cloud deployment. These issues include the configuration of the block time and the batch size for the mix-net. The block time represents the time interval to create new blocks in the distributed ledger (DL), and the batch size defines the number of votes mixed by a sealer in one off-chain operation. Since the mix-net creates the bottleneck concerning the scalability of the prototype, a proper configuration of these two parameters could drastically improve the runtime in such a distributed deployment. Finally, scalability tests could be carried out again, and a more realistic conclusion could be drawn according to a real-world deployment of such a REV system.

8.1.2 Improve the Security of the Deployment

CI/CD Security

Section 5.1 worked out a container security threat model, which was not fully applied to the design of the CI/CD infrastructure. Two important points that we did not implement yet are the signature of the Docker images and the enforcement of deployment checks. The building machine should create the signature for the Docker image to verify the origin of the image later on. Deployment checks can then enforce the verification of the signature. Additional deployment checks (like the rejection of images with detected vulnerabilities) can be also be applied.

Penetration Tests and Static Code Analysis

To ensure the security of the deployment, ethical hackers should execute penetration tests for the whole CI/CD infrastructure and the deployment of the REV system. With the mimic of an attacker, new unknown vulnerabilities could be identified in the running systems. In addition, a static code analysis could also detect implementation weaknesses within the system. With these methods, the distributed deployment could be tested from a security point of view, with the goal of a productional deployment in the future.

Add Aditional Security Components

By analyzing the infrastructure of two productional REV systems in section 4, we realize that some additional components are required for a secure productional deployment. One is DDOS protection to secure the backend components from cyber-attacks. In these attacks, an adversary tries to overload the system with numerous requests. Another component would be a web-application firewall (WAF). Such a firewall is located at the application layer and analyzes HTTP packages from the clients and filters out vulnerable requests. These two components are standard for a productional web application. As such, by focusing on the goal of a REV system's productional deployment, these components are essential to be added.

8.1.3 Improve the Usability of the Deployment

Independent Operating System

Deployment scripts are all coded to configure the system within an Ubuntu 18.04+ operating system. To make these scripts more universal, they should also provide implementations for other operating systems. Consequently, the infrastructure would be more comprehensive and adaptable to other environments.

Automatic Configuration of the Build Machine

The system components of the build machine are deployed using IAAC. However, the pipelines within the build machine need to be configured manually up until now. In the future, these pipelines could automatically be set up when the build machine is deployed. With this improvement, we could also outsource the configuration of the build pipeline to configuration scripts.

Bibliography

[1] Ben Adida. "Helios: Web-based Open-Audit Voting." In: *USENIX security symposium*. Vol. 17. 2008, pp. 335–348.

[2] Monya Baker. "Reproducibility crisis". In: *Nature* 533.26 (2016), pp. 353–66.

[3] Josh Benaloh. "Ballot Casting Assurance via Voter-Initiated Poll Station Auditing." In: *EVT* 7 (2007), pp. 14–14.

[4] Josh Benaloh and Dwight Tuinstra. "Receipt-free secret-ballot elections". In: *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*. 1994, pp. 544–553.

[5] David Yeregui Marcos del Blanco, David Duenas-Cid, and Héctor Aláiz Moretón. "E-Voting System evaluation based on the Council of Europe recommendations: nVotes". In: *International Joint Conference on Electronic Voting*. Springer. 2020, pp. 147–166.

[6] Carl Boettiger. "An introduction to Docker for reproducible research". In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79.

[7] Sergiu Bursuc, Gurchetan S Grewal, and Mark D Ryan. "Trivitas: Voters directly verifying votes". In: *International Conference on E-Voting and Identity*. Springer. 2011, pp. 190–207.

[8] Pyrros Chaidos et al. "Beleniosrf: A non-interactive receipt-free electronic voting scheme". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1614–1625.

[9] David Chaum. "Secret-ballot receipts: True voter-verifiable elections". In: *IEEE security & privacy* 2.1 (2004), pp. 38–47.

[10] David L Chaum. "Untraceable electronic mail, return addresses, and digital pseudonyms". In: *Communications of the ACM* 24.2 (1981), pp. 84–90.

[11] Benoît Chevallier-Mames et al. "On some incompatible properties of voting schemes". In: *Towards Trustworthy Elections*. Springer, 2010, pp. 191–199.

[12] Michael R Clarkson, Stephen Chong, and Andrew C Myers. "Civitas: Toward a secure voting system". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE. 2008, pp. 354–368.

[13] *Continuous integration vs. continuous delivery vs. continuous deployment*. `https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment`. Accessed: 2021-06-01.

[14] Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. "Belenios: a simple private and verifiable electronic voting system". In: *Foundations of Security, Protocols, and Equational Reasoning*. Springer, 2019, pp. 214–238.

[15] Véronique Cortier and Ben Smyth. "Attacking and fixing Helios: An analysis of ballot secrecy". In: *Journal of Computer Security* 21.1 (2013), pp. 89–148.

[16] Véronique Cortier et al. "Election verifiability for helios under weaker trust assumptions". In: *European Symposium on Research in Computer Security*. Springer. 2014, pp. 327–344.

[17] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. "A secure and optimally efficient multi-authority election scheme". In: *European transactions on Telecommunications* 8.5 (1997), pp. 481–490.

[18] Stephanie Delaune, Steve Kremer, and Mark Ryan. "Coercion-resistance and receipt-freeness in electronic voting". In: *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. IEEE. 2006, 12–pp.

[19] Moritz Eck. "Mixnets in a Distributed Ledger Remote Electronic Voting System". MA thesis. 2021.

[20] Moritz Eck, Alex Scheitlin, and Nik Zaugg. "Design and Implementation of Blockchain-based E-Voting". 2020.

[21] *Ethereum Blockchain whitepaper*. `https://ethereum.org/en/whitepaper/`. Accessed: 2021-05-27.

[22] Jenny Fong. *Are Containers Replacing Virtual Machines?* `https://www.openshift.com/blog/build-once-deploy-anywhere`. Accessed: 2021-06-02.

[23] Martin Fowler and Matthew Foemmel. "Continuous integration". In: *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* 122.14 (2006), pp. 1–7.

[24] Steven N Goodman, Daniele Fanelli, and John PA Ioannidis. "What does research reproducibility mean?" In: *Science translational medicine* 8.341 (2016), 341ps12–341ps12.

[25] Rolf Haenni et al. "CHVote System Specification." In: *IACR Cryptol. ePrint Arch.* 2017 (2017), p. 325.

[26] Thomas Haines and Peter B Rønne. "New Standards for E-Voting Systems: Reflections on Source Code Examinations." In: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 391.

[27] Feng Hao, Peter YA Ryan, and Piotr Zieliński. "Anonymous voting by two-round public discussion". In: *IET Information Security* 4.2 (2010), pp. 62–67.

[28] Sven Heiberg et al. "Improving the verifiability of the Estonian Internet Voting scheme". In: *International Joint Conference on Electronic Voting*. Springer. 2016, pp. 92–107.

[29] Trey Herr. "Breaking Trust – Shades of Crisis Across an Insecure Software Supply Chain". In: USENIX Association, Feb. 2021.

[30] Alexander Hofmann. "Security Analysis and Improvements of a Blockchain-based Remote Electronic Voting System". MA thesis. 2020.

[31] Matthew Hutson. *Artificial intelligence faces reproducibility crisis*. 2018.

[32] Hugo Jonker, Sjouke Mauw, and Jun Pang. "Privacy and verifiability in voting systems: Methods, developments and trends". In: *Computer Science Review* 10 (2013), pp. 1–30.

[33] Ari Juels, Dario Catalano, and Markus Jakobsson. "Coercion-Resistant Electronic Elections". In: (2005).

[34] Fatih Karayumak et al. "Usability Analysis of Helios-An Open Source Verifiable Remote Electronic Voting System." In: *EVT/WOTE* 11.5 (2011).

[35] Christian Killer et al. "Ãternum: A Decentralized Voting System with Unconditional Privacy". In: May 2021.

[36] Christian Killer et al. "From Centralized to Decentralized Remote Electronic Voting". In: 2021.

[37] Steve Kremer, Mark Ryan, and Ben Smyth. "Election verifiability in electronic voting protocols". In: *European Symposium on Research in Computer Security*. Springer. 2010, pp. 389–404.

[38] Oksana Kulyk, Vanessa Teague, and Melanie Volkamer. "Extending helios towards private eligibility verifiability". In: *International Conference on E-Voting and Identity*. Springer. 2015, pp. 57–73.

[39] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. "A smart contract for boardroom voting with maximum voter privacy". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, pp. 357–375.

[40] David C Mills et al. "Distributed ledger technology in payments, clearing, and settlement". In: (2016).

[41] Tal Moran and Moni Naor. "Receipt-free universally-verifiable voting with everlasting privacy". In: *Annual International Cryptology Conference*. Springer. 2006, pp. 373–392.

[42] K. Morris. *Infrastructure as Code: Managing Servers in the Cloud*. Safari Books Online. O'Reilly Media, 2016. ISBN: 9781491924396. URL: `https://books.google.ch/books?id=BIhRDAAAQBAJ`.

[43] Stephan Neumann et al. "Towards a practical jcj/civitas implementation". In: (2013).

[44] *nVotes*. URL: `https://nvotes.com/` (visited on 01/02/2021).

[45] Kei Omizo. *Build Once, Deploy Anywhere!* `https://www.openshift.com/blog/build-once-deploy-anywhere`. Accessed: 2021-06-02.

[46] Hans E Plesser. "Reproducibility vs. replicability: a brief history of a confused terminology". In: *Frontiers in neuroinformatics* 11 (2018), p. 76.

[47] Suporn Pongnumkul, Chaiyaphum Siripanpornchana, and Suttipong Thajchayapong. "Performance analysis of private blockchain platforms in varying workloads". In: *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE. 2017, pp. 1–6.

[48] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. "An introduction to docker and analysis of its performance". In: *International Journal of Computer Science and Network Security (IJCSNS)* 17.3 (2017), p. 228.

[49] L. Rice. *Container Security: Fundamental Technology Concepts that Protect Containerized Applications*. O'Reilly Media, 2020. ISBN: 9781492056713. URL: `https://books.google.ch/books?id=J4fiDwAAQBAJ`.

[50] Ronald L Rivest. "On the notion of 'software independence' in voting systems". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366.1881 (2008), pp. 3759–3767.

[51] Peter YA Ryan, Peter B Rønne, and Vincenzo Iovino. "Selene: Voting with transparent verifiability and coercion-mitigation". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 176–192.

[52] Kazue Sako and Joe Kilian. "Receipt-free mix-type voting scheme". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1995, pp. 393–403.

[53] Ben Smyth and David Bernhard. "Ballot secrecy and ballot independence coincide". In: *European Symposium on Research in Computer Security*. Springer. 2013, pp. 463–480.

[54] Michael A Specter, James Koppel, and Daniel Weitzner. "The ballot is busted before the blockchain: A security analysis of voatz, the first internet voting application used in us federal elections". In: *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020, pp. 1535–1553.

[55] Oliver Spycher et al. "A new approach towards coercion-resistant remote e-voting in linear time". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2011, pp. 182–189.

[56] Raffaele Stefanelli, Denis Morel, and Xavier Monnat. "A secure e-voting infrastructure. implementation by swiss post". In: *Second In* (2017), p. 326.

[57] *Universal Declaration of Human Rights*. `https://www.un.org/en/about-us/universal-declaration-of-human-rights`. Accessed: 2021-05-21.

[58] *Verificatum Mix-Net*. `https://www.verificatum.org/`. Accessed: 2021-05-27.

[59] Christophe Vigouroux and Franck Ponchel. *CHVote System Architecture*. `https://gitlab.com/chvote2/documentation/chvote-docs/builds/artifacts/master/raw/design/target/generated-docs/pdf/architecture/system-architecture.pdf?job=artifact:design-docs`. Accessed: 2021-06-14.

[60] Wenbo Wang et al. "A survey on consensus mechanisms and mining strategy management in blockchain networks". In: *IEEE Access* 7 (2019), pp. 22328–22370.

[61] Stefan G Weber, Roberto Araujo, and Johannes Buchmann. "On coercion-resistant electronic elections with linear work". In: *The Second International Conference on Availability, Reliability and Security (ARES'07)*. IEEE. 2007, pp. 908–916.

Abbreviations

CD	Continuous Delivery
CI	Continuous Integration
CLI	Command-Line Interface
CPU	Central Processing Unit
DDOS	Distributed Denial-of-Service
DKG	Distributed Key Generation
DL	Distributed Ledger
DLT	Distributed Ledger Technology
DNS	Domain Name System
GB	Gigabyte
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IAAC	Infrastructure-as-a-Code
MITM	Man-in-the-middle
PBB	Public Bulletin Board
PBFT	Practical Byzantine Fault Tolerance
PoA	Proof-of-Authority
PoS	Proof-of-Stake
PoW	Proof-of-Work
RAM	Random-Access Memory
REV	Remote Electronic Voting
SSCA	Software Supply Chain Attacks
TLS	Transport Layer Security
URL	Uniform Resource Locator
VM	Virtual Machine
WAF	Web Application Firewall

List of Figures

List of Tables

List of Listings

Appendix A

Additional Provotum 3.0 Scalability Results

The following results show the average runtime for randomizing the ballots. The experiments were run with 2, 5, and 10 sealers and are represented in absolute values. The average runtime to randomize one ballot is on average $\approx 0.2s$.

Number of votes	2 sealers	5 sealers	10 sealers
100	20s	19s	22s
1'000	130s	212s	175s
10'000	1'375s	2'019	1'976s

Table A.1: Runtime for randomizing the ballots

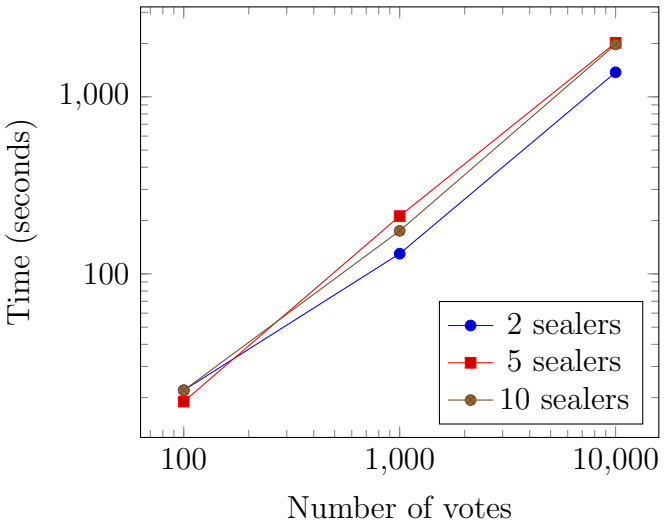

Figure A.1: Runtime for randomizing the ballots

Appendix B

Installation Guidelines

The whole source code of the prototype can be found here: `https://github.com/provotum/provotum-ci-cd`.

The repository contains all necessary documentation to deploy the different components of the CI/CD infrastructure and the applications. The README file at the root contains prerequisites to set up the infrastructure. In addition, the README files in `/terraform/*/` contain specific installation guides for the chosen software component.

Appendix C

Contents of the CD

The thesis is handed in with an archive containing the following items:

- The PDF file of this report.
- The source code of the report in LaTeXformat.
- Images as source and PNG files.
- The source code of the implemented prototype
- Results of the scalability experiments
- The scripts to reproduce the scalability experiments