University of Zurich[UZH]

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

Hardware Key Management for a Blockchain-based Remote Electronic Voting System

Andreas Knecht
Zurich, Switzerland
Student IDs: 11-916-152

Supervisors: Christian Killer, Eder John Scheid
Date of Submission: July 15, 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

Remote Electronic Voting (REV) becomes an increasing topic of interest since citicens are used to complete many other tasks online. But REV systems need to provide a very high level of security, hindering deployment of adequate solutions. The Communication Systems Group at the university of Zurich implements the REV system provotum.

This thesis explores deployment of a USB dongle in the Remote Electronic Voting (REV) system Provotum to improve security. This work proposes a design centered around a set of 6 – mostly mutually exclusive – functions running on a USB dongle that provide additional protection against theft and selling of private keys, as well as against manipulation by malware on the voter's computer. A USB dongle is selected and all functions are implemented in a firmware running on that dongle, as well as additional system components to constitute a Proof–of–Concept that demonstrates end–to–end integration of the functionality. All finite–field cryptography of the Provotum system (except RSA signatures) is changed to Elliptic Curve Cryptography (ECC) to achieve better firmware runtime performance. The implementation overcomes several challenges that are documented in this work and achieves to offer a strong baseline codebase that facilitates future development of further dongle–enabled Provotum functionality. Additionally, since all proposed dongle functions are implemented in firmware, varying levels of a tradeoff between security and usability can be controlled in the system.

Zusammenfassung

Weltweit wird das Thema Online-Voting mehr und mehr präsent, da Bürger es gewohnt sich, allerlei Geschäfte online zu erledigen. Doch Online-Voting-Systeme müssen einen sehr hohen Grad an Sicherheit bieten, mit dem sich bis heute viele Systeme schwertun. Die Communication Systems Group an der Universität Zürich entwickelt das Online–Voting–System (Remote Electronic Voting (REV)) Provotum.

Diese Masterarbeit untersucht die Nutzung eines USB Dongles im REV System Provotum, um die Sicherheit zu erhöhen. Eine Lösung wird vorgeschlagen, die 6 – einzeln ansteuerbare und sich in Funktionalität teilweise überschneidende – Funktionen auf dem Dongle implementiert, die zusätzlichen Schutz gegen den Diebstahl und Verkauf von privaten Schlüsseln, sowie gegen Manipulation durch Malware auf dem Computer des Stimmenden bietet. Ein USB Dongle wird ausgewählt und alle Funktionen werden in einer Firmware für diesen Dongle implementiert. Zusätzliche Systemkomponenten werden entwickelt oder modifiziert, um einen Proof–of–Concept zu erlangen, der die End–to–End–Integration der Funktionalität zeigt. Alle Verwendungen von "Finite–Field"–Kryptographie werden durch "Elliptic Curve Cryptography (ECC)" ersetzt, um die Laufzeit von Kryptographischen Funktionen auf dem Dongle zu verbessern. Die Implementierung der Lösung muss mit einigen Schwierigkeiten kämpfen, die in dieser Arbeit dokumentiert werden, und erreicht am Schluss einen Stand, der als solide Basis für zukünftige Weiterentwicklungen von Dongle-Hardware-unterstützten Funktionen in Provotum. Die Tatsache, dass alle vorgeschlagenen Dongle-Funktionen in der Firmware implementiert sind, erlaubt ausserdem durch die kontrolliete Wahl der verwendeten Funktion eine Balance zwischen Nutzbarkeit und Sicherheit.

Acknowledgments

I want to thank Christian Killer, Prof. Dr. Burkhard Stiller and the CSG for the opportunity to work on this very interesting topic. This thesis adds to a list of a few, all highly interesting, theses and other projects that the CSG has given me the opportunity to work on. I also want to thank Christian Killer for the calls in which he has given me very valuable input. Further, I want to thank Monika for her highly valued input and her patience with me during times when I was highly focused on this thesis. I also want to thank Thomas for his pragmatic view when it came to me dividing my attention between the thesis and other tasks. Additionally, I want to thank Nora, Jonas and everybody else who has supported me during the time of writing.

Contents

Chapter 1

Introduction

Multiple democracies worldwide are introducing or at least testing Remote Electronic Voting (REV) systems in which voters can cast their vote over the Internet from their own personal computer. While in Switzerland mail-in voting is still the standard way of casting votes, REV might reduce cost over traditional voting [1, 2, 3], as well as reduce the work involved for the voter, which could increase voter turnout [2, 4, 5, 3]. This is especially important as the mobility of persons keeps increasing and traditional voting binds voters geographically. On the other hand, since all voting systems used in practice have to provide stringent security and privacy guarantees, widespread introduction of REV systems has not yet occurred in most countries. Ballots must be kept secret to avoid revealing how voters voted. Additionally, an honest and correct tally of the votes must be verifiable and guaranteed even in presence of malicious actors. Another problem is that personal computers on which voters vote in an REV setting are easily infected by malware that might compromise privacy or even alter a voter's vote.

External hardware, such as USB dongles, has long been used in other use cases for adding protection to online interactions against theft of private keys and manipulation by malware. The most prominent example is online banking, but USB dongle powered 2-factor authentication has been adopted by many other websites and online services. Using such hardware for REV might appear to be a straightforward way to achieve similar security gains. But veryfiability requirements in online voting directly oppose privacy requirements and achieving the optimal balance is hard even with the use of USB dongles. Additionally, an REV system requires cooperation of many different actors, none of which trust each other. In contrast, online banking has a simpler setup in that the users trust the bank and its IT infractructure. An additional difficulty in using USB dongles for REV is that dongles are resource constrained and have limited capabilities in processor speed and memory. The runtime of functions must not be too long so that the burden on the usability of the system is too high, while the required algorithms are more computationally expensive than those required for online banking.

Provotum [6, 7, 8, 9] is a remote electronic voting system developed by CGS@Ifi. Provotum uses a public permissioned Blockchain (BC) to store the encrypted ballots and final tally. This distributed ledger allows only authorized parties to add data, but everybody can read the data. Data stored in the ledger is immutable and cannot be removed. Provotum-RF

is already a pretty solid REV. Under reasonable assumptions, the system provides ballot secrecy (non-everlasting), receipt-freeness, individual- and universal verifiability, as well as recorded-as-cast and counted-as-recorded verifiability. A parallel thesis to this one has the goal of improving cast-as-intended verifiability. The baseline version of Provotum for this work is Provotum-RF [8].

This master thesis has the goal of offloading some cryptographic operations of the system to an external USB dongle with processing capabilities in order to improve the system's security. Security benefits include improved protection against theft of keys and situations where voters are coerced or offered money to vote a certain way. Achieving a system that is resistant to such manipulation attempts is a topic of academic study, but all systems implementing solutions also have drawbacks, if only poor usability.

1.1 Project Goals and Contribution

This work starts with outlining the cryptographic operations on which the system is based. A literature study compares similarities of other REV and places special focus on systems that achieve or claim to achieve coercion-resistance, as well as REV that make use of a hardware dongle or similar device. Multiple potential uses of the USB dongle in the Provotum system are discussed. Some of the proposed mechanisms, despite having clear security benefits, have a potential negative impact on the usability of the system. Other mechanisms don't provide sufficient security improvements to warrant adoption into Provotum. Support for all the identified mechanisms is implemented in the firmware (FW) of the USB dongle, allowing multiple experimental versions of Provotum to test all the features without large modifications to the dongle firmware, and serving as a baseline firmware. The baseline firmware can in the future be extended with more features, modified for improved performance or be ported to different, more secure hardware. Additional Provotum system components are modified to obtain a Proof-of-Concept (PoC) system that demonstrates an end-to-end implementation of the proposed design. Finally, the dongle firmware is evaluated from a performance view, challenges during development documented and future optimization opportunities for the firmware discussed. The discussion also covers the performance impact on the Provotum system as a whole.

Chapter 2

Background and Cryptographic Primitives

First, let's recall cryptographic primitives and algorithms used in the Provotum system. Like all REV Provotum bases its functionality on strong cryptographic algorithms in order to achieve proper functioning of the system with little trust requirements in the individual parts. In the end, this chapter also gives a brief overview over other background topics relevant for this work, namely the USB interface and Blockchain systems.

2.1 Random Numbers

Many cryptographic algorithms require random numbers. [10] gives a good natural language overview on the topic and [11] and [12] give more formal definitions. True random numbers can be generated by physical processes, including coin tossing. Digital applications can draw a stream of true random numbers from specialized hardware drawing randomness from thermal noise or radioactive decay. True random numbers follow no pattern and contain no statistically discernible bias. For many software purposes no true random number generator is available or does not produce output at the required rate. Pseudorandom Number Generators (PRNG) are required. They generate a stream of apparently random numbers deterministically in software. A cryptographic PRNG must satisfy two properties. First, a stream of random numbers or random bytes must be indistinguishable from true random data, i.e. contain no bias nor patterns. Second, it must be infeasible for an attacker to recreate the output stream. Since a PRNG is completely deterministic, the future output depends only on internal state. The input required to construct the initial internal state is called the *seed*. A seed must be sufficiently large and truly random so that an attacker cannot guess it. Additionally information about the internal state must not be leaked to an attacker via the output or other channel.

Note that manipulated PRNGs can be very hard to discover but can completely break encryption: if the random parameters used in some encryption schemes are reproducible by an attacker, obtining the plaintext is straightforward. Hence, accidental or deliberate

weaknesses or backdoors in a PRNG can be fatal. A key derivation function (see below), for example, outputs bytes that are indistinguishable from random without knowledge of the key, but with knowledge of the input key and the parameters, the output stream can be trivially recreated.

2.2 Bounded Attacker

The difficulty assumptions of the following algorithms are based on the assumption that an attacker is computationally bounded in both time and storage, i.e. that the attacker can only successfully solve problems for which a polynomial time and space algorithm exists [13, 12, 14]. If something is claimed to be infeasible, it means that no polynomial time algorithm exists (or has been found to date) that solves the given problem [13, 12, 14].

2.3 Cryptographic Hash Functions

A cryptographic hash function is a deterministic function that takes an arbitrary binary input and outputs a digest of a defined size, for example 256 bits for SHA-256. [15] gives a good overview and [11] a detailed formalized definition. The hash function must be designed so that a small change in the input leads to a large change in the output. Obviously, the image of the function is much smaller than the domain, leading to collisions: multiple inputs exist that produce the same output. However, it must be infeasible for anyone to find two inputs leading to a collision. Additionally, cryptographic hash functions should behave as one-way functions (i.e., informally, given a hash, it should be infeasible to find a message that produces a given hash). This is formalized in the following requirements [15, 11]:

Preimage resistance: given a hash $H(m)$ of a message m it is infeasible without knowledge of m to find m.

Second preimage resistance: given a message m, it is infeasible to find another message m' with the same hash, that is $H(m) = H(m')$.

Collision resistance: it is infeasible to find any two messages m and m' with the same hash ($H(m) = H(m')$).

Furthermore, due to the aforementioned properties, the output of hash functions is not distinguishable from random data if the input is not known.

2.4 Key Derivation Functions

Key Derivation Functions (KDF) do not form a central part in this work but are mentioned a few times. A key derivation function is a deterministic function that takes a secret key

and some additional input and outputs data of a configurable size that is intended to be used as a cryptographic key. [10] gives a natural language overview and [11] a formal definition. The additional input could be a string describing the purpose of the to-be-generated key. Common usage is to use one input key to produce one or multiple output keys that appear to be unrelated. Even with knowledge of all input data except the secret key the output of a KDF must be indistinguishable from random. A commonly used KDF is the Hash-Based KDF (HKDF).

2.5 ECC

Elliptic Curve Cryptography (ECC) uses operations defined on the points of an Elliptic Curve (EC). This document uses the definitions and notation from [11]. Under modular arithmetic and a carefully chosen curve, the points on the curve form a group. The group operation is addition of curve points. By extension, multiplication of points with a scalar value is also defined (i.e. adding a point to itself multiple times). Analogous to finite field cryptography at least one generator element exists, which for ECC is the point G. Some elliptic curves use a generator that generates a subgroup that is smaller than the number of elements in the group, i.e. the order of the subgroup n is less than the order of the full group $|E(\mathbb{F}_p)|$. The resulting cofactor $h = \frac{1}{n}|E(\mathbb{F}_p)|$ should be small.

The big advantage of ECC is that the size of scalars and point coordinates required to achieve equivalent security is much smaller than the scalars used in finite-field cryptography. A 256 bit ECC cryptosystem is equivalent to a finite-field cryptography key larger than 2048 bits [16]. Hence, the cost of computation and transmission is reduced when compared to finite-field cryptography.

Frequently, upper case letters are used to denote points on the curve and lower case letters are used to denote scalars. This work will adhere to the same format. Unless noted otherwise, all following formulas are defined as calculations in the EC group if they involve points. Operations on scalars are implicitly defined as operations modulo n.

Different encodings exist for storage of points in computer memory for efficient computations. For example, Ristretto [17] stores only the x coordinate and exposes a view on the curve where scalars are divided by the cofactor of the curve in order to prevent subtle security weaknesses where unaware applications assume the curve order to be prime.

2.6 ElGamal Cryptosystem

ElGamal is a public-key cryptosystem. Most sources such as [11, 13, 15] give definitions for the finite-field variant of the algorithm. Most finite-field-based algorithms can be transformed into an EC variant by replacing exponentiation with point-scalar multiplication and replacing multiplication with addition. [18] uses this transformation to transform zero-knowledge proofs (including some used in this work) to an EC variant. We use the EC notation from [18] and attribute the basic theory of ElGamal to the original paper

[19]. A user generates a key pair consisting of a private and a public key. The user can distribute the public key freely but must keep the private key secret. Anyone with the public key can then encrypt data for that user by using the public key. Only the user in possession of the private key can decrypt the data. An ElGamal private key x is simply a random number drawn from $\mathbb{Z}_n$ and the calculation of the public key Y is given in Equation 2.1.

$$Y = xG \tag{2.1}$$

Encryption of a message m requires first mapping the message to a point M (see also Section 2.8). Encryption is then given by Equation 2.2 where r is a random number (that must be different for each encryption) drawn from $\mathbb{Z}_n$ and the tuple (C, D) is the resulting ciphertext.

$$(C, D) = (rG, rY + M) \tag{2.2}$$

Decryption then follows Equation 2.3.

$$M = D - Cx \tag{2.3}$$

Applying the decryption from Equation 2.3 yields M. Depending on the mapping of m to M recovery of m is straightforward or non-trivial.

Usage of a parameter r in the encryption makes ElGamal a *probabilistic* encryption system: many ciphertexts are possible for a given message m and public key Y.

2.7 Difficulty Assumptions

The security of the ElGamal cryptosystem (and others) is based on the difficulty assumption of the following three problems [11, 15]:

Elliptic Curve Discrete Logarithm (ECDL): given an appropriately sized elliptic curve group, generator G and Gx it is infeasible to find x.

Elliptic Curve Computational Diffie-Hellman (ECCDH): given an appropriately sized elliptic curve group and generator G, given aG and bG for randomly selected elements a and $b \in \mathbb{Z}_n$, it is infeasible to find abG.

Elliptic Curve Decisional Diffie-Hellman (ECDDH): given an appropriately sized elliptic curve group and generator G, given (aG, bG, cG), it is infeasible to determine if $cG = abG$.

For appropriately chosen elliptic curves, the three problems are all computationally infeasible to solve. With bilinear pairings [11], curves exist where the Decisional Diffie-Hellman problem becomes easy while the Computational Diffie-Hellman problem remains hard. Pairing based cryptography uses different, specific curves and for the curves widely used for regular ECC, ECDDH remains hard.

2.8 Homomorphic Properties

ElGamal has homomorphic properties that are very useful for REV. In a homomorphic encryption system a relation exists on ciphertexts that corresponds to a relation on plaintexts (see Equation 2.4) [13, 15]. We define the notation for encryption of message m under key k resulting in ciphertext c as $E_k(m) = c$.

$$E_k(a) \oplus E_k(b) = E_k(a \oplus b) \tag{2.4}$$

The decryption of a sum of ElGamal ciphertexts yields the sum of their plaintexts. While the finite-field variant of ElGamal has only multiplicative homomorphic properties and a variant called Exponential ElGamal is required to obtain additive homomorphic properties, the EC variant of ElGamal has additive homomorphic properties without modification.

Since we map a message m to a point on the curve M during encryption, this mapping needs homomorphic properties as well [20, 21], i.e. given two messages m_1 and m_2 and their mappings M_1 and M_2, the sum $M_1 + M_2$ must equal the mapping of $m_1 + m_2$. Multiplying messages with the generator point G has these homomorphic properties. But based on the hardness of the ECDL problem, recovery of m from $M = mG$ is not straightforward. If the range of possible values of m is small, as is the case with Provotum, m can be brute-forced from mG.

2.8.1 Re-Encryption

The homomorphic property of EC ElGamal makes it possible to obtain a different ciphertext (C', D') from (C, D) by adding an encryption of 0 under the same public key, as is given in Equation 2.5. We define the notation for encryption of message m under key k with random parameter r resulting in ciphertext c as $E_k(m, r) = c$.

$$E_k(m, r) \otimes E_k(0, s) = E_k(m, r') \tag{2.5}$$

2.9 Digital Signatures

A digital signature scheme, like public-key encryption, uses a pair of related keys: a public and a private key. It allows using the private key to generate a *signature* for a *message* that can be *verified* with the public key [10]. The verification function takes the public key, a message and a signature and returns a boolean value. A signature is only valid for a given message and key pair. Hence, a signature protects the integrity of the message and it ties it to the identity of the signer (or rather the possession of the private key).

2.9.1 Sr25519

The signature scheme used by the Substrate Blockchain used in Provotum is Sr25519 [22, 23], which is a Schnorr signature (directly derived from the Schnorr Proof in Section 2.10.1) on the Curve25519 curve. Points and scalars are encoded in the Ristretto format [17]. The aggregation of values-to-be-hashed and the hash function are defined as a Merlin transcript [24], which is defined as operations on the Strobe protocol framework [25], which internally uses a variant of the Keccak hash function family [26].

The main implementation of Sr25519, which is also used by the Substrate Blockchain, is the Rust project *Schnorrkel* [23]. Schnorrkel makes major breaking changes to the transcript around version 0.3. While it is now at version 0.9.1, the Substrate version used in Provotum-RF and Provotum-HW still uses version 0.1.0.

2.9.2 Blind Signatures

The concept of blind signatures was introduced by Chaum [27]. The goal is that a *sender* can have some data signed by a third party, the *signer*, without the *signer* learning the data. The party verifying the signature is called the *verifier*. Usually the *sender* applies a transformation to the data to be signed: he blinds the data. The *signer* applies the signature to the blinded data and returns it to the *sender*. The *sender* then applies a reverse transformation on the signature to obtain a signature that is valid for the unblinded data. A blinded variant of the RSA scheme is used for Provotum [8]. Many other systems exist, including [28, 29, 30, 31, 32], which might have an application in future work on Provotum in order to overcome problems with identity provisioning in conjunction with panic passwords.

2.10 Zero Knowledge Proofs

Zero Knowledge Proofs (ZKP) play a central role in most REV. Informally, a zero knowledge proof allows one party, the *prover*, to prove a statement to another party, the *verifyer*, without revealing any additional information about the statement. Sources on the theory include [11, 13, 15]. For example, a person can prove knowledge of a private key related to a given public key without revealing the private key. Zero knowledge proofs must satisfy the following requirements:

Completeness: if the statement to be proven is true, the prover can convince the verifier.

Soundness: if the statement to be proven is false, the prover fails to convince the verifier with overwhelming probability.

Zero knowledge the verifier learns nothing beyond the fact that the statement is true.

The last requirement is typically proven by construction of a *simulator*. A simulator fakes the transcript of a normal prover-verifier interaction. If such a transcript can easily be manufactured, this implies that a normal prover-verifier interaction conveys no information apart from the truthiness of the statement to be proven. Furthermore, if a transcript can easily be faked, the proof will only convince the verifier directly participating in the protocol but nobody else.

2.10.1 Sigma Protocols

Sigma Protocols – also frequently spelled Σ-Protocols – are a class of ZKP that are comparatively computationally inexpensive and allow proving statements around discrete logarithms [11, 13, 15]. Provotum exclusively uses Sigma Protocols as ZKP.

The simplest Sigma Protocol (see Figure 2.1) is due to Schnorr [33] and allows proving knowledge of the (EC) discrete logarithm of a number. It is easy to ascertain completeness of the protocol in that the honest prover, in possession of x, can easily calculate z so that the verifying equation holds. Soundness is due to the difficulty of the ECDL problem. A malicious prover, not in possession of x, cannot easily find a z that will convince a verifier. Of course, a malicious prover can hope to anticipate the challenge c sent by the verifier and pre-compute $A = zG - cY$ for some randomly chosen z. Hence, a malicious prover can convince the verifier with small probability. A full transcript can be faked in the same way (by randomly drawing c and z and computing A). Hence, since a simulator exists, the honest execution of the protocol is zero-knowledge.

Prover		**Verifier**
knows x		knows $Y = xG$
$r \leftarrow\$ \mathbb{Z}_n$		
	$A = rG \longrightarrow$	
		$c \leftarrow\$ \mathbb{Z}_n$
	$\longleftarrow c$	
	$z = r + cx \longrightarrow$	
		$zG \stackrel{!}{=} A + cY$

Figure 2.1: Schnorr Proof

Note that this proof is zero-knowledge only under the assumption of an honest verifier. If the verifier maliciously selects c to have some relation to A, the verifier can gain an advantage. Mitigations are having the verifier commit to a value c before the protocol starts (commitment schemes are not covered in this chapter and the reader is referred, for example, to [15]) or using the Fiat-Shamir heuristic (described below).

Figure 2.2 gives a Sigma Protocol due to Chaum and Pedersen [34] that allows to prove the equivalence of two (EC) discrete logarithms xG and xH with different bases.

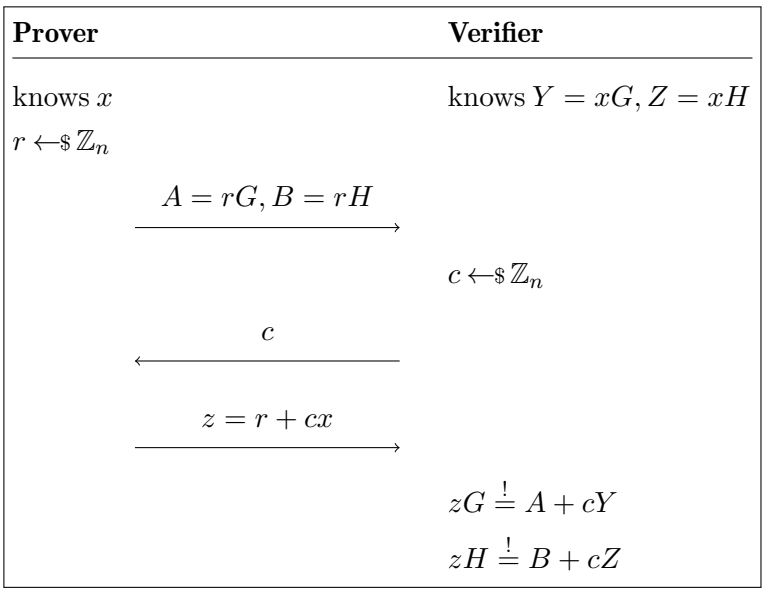

Figure 2.2: Chaum-Pedersen Proof

Note that these protocols are interactive.

2.10.2 Fiat-Shamir Heuristic

The Fiat-Shamir heuristic is a modification of interactive ZKP due to Fiat and Shamir [35] that allows making interactive protocols non-interactive. Instead of receiving a challenge c from the verifier, the prover computes a challenge by hashing the publicly known values and his commitment (e.g. $H(G||Y = xG||A = rG)$ for the Schnorr Proof where $||$ denotes concatenation). Due to the unpredictable nature of hash function outputs it is unfeasible for a malicious prover to find a $c = H(G||Y||A)$ that satisfies the verification equation. Note that A is an input into the hash function to derive c and hence, a malicious prover cannot solve the verification equation for A: changing A changes c.

Two variants of the Fiat-Shamir heuristic exist. One, the *weak Fiat-Shamir* heuristic only hashes the commitments. The *strong Fiat-Shamir* heuristic additionally hashes all publicly known parameters of the statement to be proven. The weak Fiat-Shamir heuristic alloes malicious provers to produce unsound proofs in some cases [36].

2.10.3 Disjunction

Sigma Protocols can be aggregated to prove a conjuction or disjunction of statements. Disjunction is much more important in the context of Provotum. Figure 2.3 gives a disjunctive Schnorr proof proving knowledge of (at least) one discrete logaritm out of two [13, 15]. This is achieved by dividing the challenge sent by the verifier into a sum of two

challenges. Faking the transcript for one of the statements with a random sub-challenge c_2 fixes the other sub-challenge to $c_1 = c - c_2$. Note that in Figure 2.3 neither the prover nor verifier knows y.

Prover		**Verifier**
knows x		knows $X = xG, Y = yG$
$r \leftarrow\$ \mathbb{Z}_n$		
$c_2 \leftarrow\$ \mathbb{Z}_n$		
$z_2 \leftarrow\$ \mathbb{Z}_n$		
$B = z_2G - c_2Y$		
	$A = rG, B \longrightarrow$	
		$c \leftarrow\$ \mathbb{Z}_n$
	$\longleftarrow c$	
$c_1 = c - c_2$		
	$z_1 = r + c_1x, z_2, c_1, c_2 \longrightarrow$	
		$z_1G \stackrel{!}{=} A + c_1X$
		$z_2G \stackrel{!}{=} B + c_2Y$
		$c \stackrel{!}{=} c_1 + c_2$

Figure 2.3: Disjunctive Schnorr Proof

2.10.4 Designated Verifier Proofs

Interactive proofs convince only the verifier participating in the proof. This is due to the fakeability of the transcript. Non-interactive proofs, such as with the Fiat-Shamir heuristic, convince any verifier reading the transcript as long as the hash function is not broken. Designated Verifier (DV) proofs are a variant of non-interactive proofs that only convince a determined verifier [37]. This is achieved by adding a disjunctive clause proving the knowledge of the private key of the intended verifier. As long as the verifier is convinced that the prover does not possess the private key, he can be sure that the other disjunctive clause must be true. Since the designated verifier, however, can fake the proof with possession of his private key any other verifier cannot be sure which clause is true.

2.10.5 Divertible Proofs

Divertible proofs are discussed in [38]. Some proof transcripts are malleable and can be transformed to form another proof transcript that proves a related statement. Of course,

divertible proofs can be a security weakness, but can also be used on purpose for a given use case. Provotum-RF makes use of divertible proofs [8] to cooperatively construct a proof of ballot validity for a ballot that is randomized (re-encrypted with a secret nonce) by a third party.

2.11 Threshold Encryption and Distributed Key Generation

Threshold encryptions schemes are a modification of a public-key encryption system so that no single party holds the key for decryption. A good overview can be found in [15]. Concretely, during key generation each of the participants obtains a private key share. For decryption, each holder of a key share applies a decryption operation to the ciphertext using his key share, yielding a decryption share. Once all the decryption shares are available, they can be combined to form the decryption of the ciphertext.

The terms *threshold encryption*, *distributed key generation*, *decentralized key generation* and *secret sharing* are all used for related and overlapping concepts. Secret sharing does not necessarily only apply to cryptographic keys and some schemes are only aimed at reconstruction of the secret directly. With threshold encryption, after distribution of the key shares, the secret key is never recombined itself. Instead, distributed decryption operations allow reconstruction of a decrypted ciphertext.

In an n-of-n scheme, all private key shares are required to decrypt the ciphertext. This gives the highest protection of the ciphertext, but prevents decryption if one holder of a key share does not participate in decryption. In a t-of-n scheme, only a defined number t out of all n private key shares are required to decrypt the ciphertext. Here, the protection of the ciphertext is slightly weakened, but guarantees decryption is possible as long as t share holders participate in decryption. Provotum-RF uses an n-of-n distributed key generation scheme, which is defined in [8].

t-of-n schemes are often based on Shamir secret sharing [39]. In its basic form, the a private key is generated in a centralized setting, split into n shares and distributed to participants. The centralized private key can then be deleted. Of course, this setup is suboptimal since for some time a single party holds the private key allowing non-shared decryption. Additionally, if the participating parties do not trust each other, they have no method of detecting malicious behavior of the other parties.

The Feldman Verifiable Secret Sharing (VSS) scheme [40] still requires a central *dealer* who needs to be trusted to properly delete the original secret, but the protocol allows participants to check that the dealer generated and distributed the shares correctly.

Pedersen VSS [41] in a fully distributed setting. Each participant takes the role of dealer once in a Feldman VSS. The sum of the polynomials of all the rounds forms the secret sharing polynomial of the protocol. If the Feldman VSS in an individual round reveals a dishonest dealer, the dealer is disqualified and excluded from the protocol. It is evident,

however, that this protocol requires each participant to exchange messages with each other participant in each round of the protocol.

Gennaro et al. [42] show some weaknesses in the aforementioned VSS and propose an improvement. However, all these protocols assume a broadcast channel [43]. If individual channels to each participant are used to broadcast values, malicious actors could send different messages on different channels. In their paper, Kate and Goldberg show a VSS that works over the Internet witout broadcast channels [43] and supply a ready-to-use implementation. Another solution to obtain a broadcast channel is the use of a Blockchain. Values pushed to a Blockchain can be seen by all network participants and not modified retrospectively. Additionally, this reduces the number of individual messages that need to be sent and could be included in Provotum in future work.

2.12 USB

Universal Serial Bus (USB) is a very widely used communication interface standard [44]. Provotum-HW's dongle uses USB to connect to a voter's or sealer's computer. On the application layer, USB defines multiple standard device classes that most operating systems support without installation of additional drivers. These include [44]:

Human Interface Device (HID): this device class allows sending and receiving communication packets in both directions. It is frequently used for keyboards and mice, whose device classes are a subtype of this.

Communication Device Class (CDC): this class allows sending and receiving an undelimited stream of bytes via a virtual COM port.

Audio Device: for audio devices.

Mass Storage Device: implements an interface to abstract mass storage devices such as flash drives.

Provotum-HW uses the CDC to communicate with the USB dongle. Both streaming out of logs and exchange of command and response payloads are implemented over the same CDC channel (see Section 6.1.4). Some devices support multiple device classes over a single physical interface, but the USB software stack the dongle uses does not allow it. Otherwise, the HID class would be better suited for exchange of command and response packets.

2.13 Blockchain

The concept of Blockchain was introduced in [45]. It was first popularized by Bitcoin [46] to implement a cryptocurrency. Blockchain platforms are based on an append-only, immutable data structure: the blockchain, and a consensus algorithm, allowing network

participants reach consensus on a state without requiring trust from each single participant. As long as the majority of participants is honest, consensus on a honest system state is reached. The subject to achieve consensus on can be distribution of coins – as with cryptocurrencies, such as [46] – or results of computation – as with, for example, Ethereum [47].

The immutability property of Blockchain data structures comes from the fact that discrete pieces of data, the *blocks*, are hashed and include the hash of the previous block, forming a chain of blocks [10]. Additionally, the network agrees on the longest chain to form the current system state [46]. Modifying data in a past block requires recomputing the chain of hashes from that point onwards and catching up with the network, which ideally consists of thousands of nodes contributing processing power.

Most REV require an entity called *bulletin board* where the ballots are pushed and immutably stored. A Blockchain platform is perfectly suited for this use case due to the append-only, immutable storage guarantees.

Blockchain systems come with different types of access rights, including:

Private BC: data can only be read and written by authorized parties.

Public BC: data can be read and written by everyone. Trust is achieved due to the consensus algorithm and the immutability property of past blocks.

Public Permissioned BC: data can be read by everyone but only written by authorized parties.

Provotum uses the Substrate BC [48] in a public permissioned mode as a distributed ledger so that authorized parties can push ballots, cryptographic keys and proofs and partially decrypted tallies and everybody can read the state and verify the process and outcome of the elections.

Chapter 3

Related Work

This chapter discusses previous work on REV systems. First, the privacy requirements of REV that are frequently used in literature are introduced. Then, multiple REV systems that claim coercion-resistance are discussed. Finally, REV systems that make use of additional low-power devices (such as USB dongles) are presented.

3.1 Privacy Requirements

In literature a number of terms are frequently used to name privacy requirements for electronic voting systems. Unfortunately, the exact definition of the terms varies slightly between papers. A few authors define and use a formal, mathematical definition (e.g. [4]), which helps in security proofs of proposed systems but is unwieldy for general purpose use.

Ballot Secrecy is the first and most central requirement in a REV. A system that satisfies ballot secrecy guarantees that no one can learn how any voter voted from public information (of course, the voter could tell them personally). Ballot secrecy is a fundamental property of any voting system and must be satisfied in all REV.

Three terms name requirements in the **Verifiability** of elections conducted on REV, i.e. that auditors and the general public can verify that votes have been properly cast and tallied. Namely, **Cast-as-Intended Verifiability** is satisfied if a voter can verify the ballot they are casting correctly represents their intended vote. **Recorded-as-Cast Verifiability** is satisfied if a voter can verify if their ballot was correctly accepted by the voting system. Finally, **Counted-as-Recorded Verifiability** is satisfied if anyone can verify that the election result corresponds to the aggregated votes in the cast ballots. If all three verifiability requirements are satisfied, the term **End-to-End Verifiability** is used.

Coercion Resistance is a requirement in REV that prevents a coercer to force a voter to vote a certain way. This is a hard requirement to satisfy since the voters of an REV are not in a controlled voting booth environment, but using their own personal computer

from home or a similar uncontrolled place. A less stringent requirement is **Receipt-Freeness**, which requires that the system does not at any point contain or produce a proof of how the voter voted. Since voters can simply share their voting credentials with a coercer or vote buyer while remaining passive themselves, this requirement is not strong enough to prevent coercion or vote buying that might arise in real-life elections. **Coercion-Resistance**, informally stated, is satisfied if a voter cannot be forced to abstain from the election nor to cast a random, possibly invalid, ballot, nor to allow the coercer to vote on their behalf.

In many systems, coercion-resistance is achieved by the voter being able to cast multiple ballots (called *revoting*). Only the last ballot is counted [49, 50] or the ballots of each voter are summed up before tallying [51, 52]. These solutions are problematic in Swiss law, because it forbids the casting of multiple ballots by the same voter (article 27b of the federal ordinance on political rights).

Another solution is the use of *panic passwords* (e.g. [53]). Such systems allow registration of multiple passwords, only one of which can be used to cast a valid ballot. Usage of the other credentials casts a ballot that is apparently valid (to any system observer) but can be filtered out during tallying. Since ballots cast using a panic password will never be counted as valid by the tallying process, such systems might not fall under the Swiss law forbidding the casting of multiple ballots. A variant [54] for two-candidate elections uses a tuple of tokens, one of which marks a valid vote and the other doesn't. A vote for both candidates is cast simultaneously and the assignment of the tokens marks which candidate receives the valid vote.

Backes et al. involve a voter's mobile phone in the process and require the voter to encrypt the vote using a human-computable one-time pad (just a modular addition) [55]. This approach is coercion-resistant as long as either the mobile phone or the voter's computer cannot be observed by the coercer. Similarly, Wen and Buckland also have the voter apply a mask to his ballot which the voter can change if she is under coercion [56].

3.2 Voting Systems

Hirt and Sako [57] use homomorphic encryption and a randomizer entity that re-encrypts ballots before voters can cast them in order to achieve receipt-freeness. This forms the basis of the approach with which Provotum-RF achieves the same. Belenios-RF [58] uses a similar approach. A number of systems has been proposed that attempt to go further and achieve coercion resistance.

Juels et al. are the first to introduce the concept of coercion resistance [59] and propose a system, often referred to as JCJ, that they claim is coercion resistant. A coerced voter can use fake credentials to cast a vote. The biggest drawback is that the tallying amounts to quadratic work in the number of ballots. Smith [60] and Weber et al. [1] both propose improvements over JCJ that reduce the quadratic work factor to linear. Both systems, as well as the original JCJ were later shown to have weaknesses [61, 62, 3].

Clarkson et al. implement a variant of JCJ that they call Civitas [62]. Civitas uses the same concepts as JCJ to achieve coersion-resistance, but has some efficiency improvements.

Another variant of JCJ was introduced by Arajúo et al. [3] – AFT. They included a formal proof of the coercion-resistance. As with other improvements of JCJ, the efficiency was improved to support real-world elections.

Meng et al. [63] introduce a coercion-resistant REV that is based on a deniable encryption scheme. The voter can disclose to a coercer parameters used in the encryption that correspond to the real message m and the ciphertext c, but also craft different parameters that correspond to a different message m' and ciphertext c. The coercer has no way of distinguishing an honest opening of the encryption from a dishonest one.

Lueks et al. propose VoteAgain [50] a coercion-resistant voting system based on the *revoting* paradigm that is efficient for large elections. All cast ballots are hidden among a set of dummy ballots. Weaknesses have been discovered for the system [64] including high trust requirements in the central voting authority.

Selections [53] uses panic passwords to achieve coercion-resistance. A voter can register one password and multiple panic passwords during the registration phase. Ballots cast using panic passwords are filtered out during the final tally. For privacy-preserving tallying, a mixnet is employed.

3.3 Hardware Tokens Used in E–Voting

Using external hardware with a limited interface and running a very limited software stack can be beneficial in REV because it provides protection against malware on the voter's computer. Furthermore it can aid in coercion-resistance by managing the voter's credentials, making vote selling more difficult, and simplifying a voter's counter-strategies to be used under coercion.

Another concept offering protection against malware on a voter's computer that needs to be mentioned is *code voting* (e.g. [65]). The voter receives a mapping of candidate names to numerical codes following a random statistical distribution. The voter then enters the code representing her preferred candidate into the computer, whereby the computer cannot learn which candidate the voter voted for nor is able to modify the vote because the codes for the other candidate(s) are not known. Prepared code sheets that are mailed to the voter require trust in the authority preparing the code sheets. Combination with hardware tokens on the voter's side can yield systems that generate codes in a verifiable manner and provide the same protection against malware.

Grewal et al. propose a voting system "Du-Vote" that works under the assumption of voter's PCs by use of a hardware token [66]. The token is equipped with a keyboard and a display and is able to perform cryptographic calculations. The scheme requires the voter to copy codes displayed on her computer via the device's keypad into the device and a result code back into her computer. Usability seems somewhat limited. Furthermore,

Kremer and Rønne discovered several attacks on the scheme [67]. Some attacks can be mitigated, but at the cost of decreased usability for the voter.

Lee and Kim [68] extend the system proposed by Hirt and Sako [57] with a hardware token performing the function of the randomizer. As such, since Provotum uses a randomizer setup very similar to [57], [68] is very similar to this work. But the trust Lee and Kim's system places in the randomizer token is greater. Also they make no attempt at achieving coercion-resistance with the proposed system.

Haenni and Koenig [69] propose a hardware-token-aided REV that makes use of an optical scanner. Voting codes for all candidates are displayed as barcodes on the voter's PC, which might be infected with malware. By scanning her candidate's barcode the voter makes the selection. Since the optical channel of barcode-scanning is one-way, no information about the selected candidate can be leaked onto the PC.

Neumann and Volkamer [70] propose a variant of Civitas which uses a smart card to manage the voter's credentials. However the interactions with the smart card are complicated because the smart card participates in the interactive protocol of registration. They assume a smart card with custom firmware and a smart card reader attached to the voter's PC to interact with it. In contrast, this work uses a USB dongle which is directly attached to the PC. Additionally, the protocol is designed to minimize participation of the dongle in interactive protocols.

Chapter 4

System Architecture of Provotum-RF

This work builds on top of Provotum-RF [8]. Figure 4.1 shows the distinct applications used in the system. The Blockchain used is Substrate [48]. Substrate provides a modular Blockchain platform that allows combining several prototypical building blocks for typical tasks such as consensus finding. Alternatively, the building blocks can be provided in an own custom implementation. The mode of operation is a public permissioned BC, so that only eligible voters can push their encrypted ballots to the BC, but everybody can read the data and verify the associated zero knowledge proofs. Furthermore, Substrate does not need a smart contract to implement custom logic for the Provotum system since the code is directly included in the nodes running the Substrate BC.

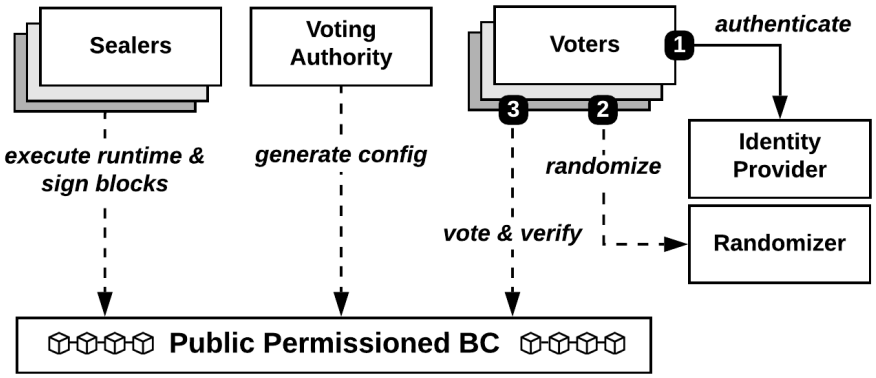

Figure 4.1: Provotum-RF Architecture Diagram [8]

The following is a high level summary of the Provotum-RF system architecture. [8] describes the architecture in high detail. The nodes that make up the distributed ledger of the **public bulletin board** are run on the **voting authority** and **sealer** machines. The **voting authority** can create elections, as well as moving them to the tallying phase. **Sealers** use the DKG algorithm from Section 2.11 to each create a private key share for an election and post the corresponding public key to the **BC**. The encryption algorithm used is Exponential ElGamal [8]. The **voting authority** can then initiate the **BC** to combine the public keys to form an election's public key. **Voters** have to register with the

identity provider in order to receive credentials that allow them to push their ballot to the **BC**. Concretely, they create a key pair and send the blinded public key (using RSA blind signatures as detailed in Section 2.9.2) along with some identity documentation proving their eligibility for the election to the **identity provider**. If the voter eligibility is confirmed, the **identity provider** signs the blinded public key, which enables the corresponding private key to sign a **BC** transaction with a ballot. Ballots are encrypted using the election's public key that can be fetched from the **BC**. In order to provide receipt-freeness, **voters** have to send their ballots to the **randomizer**, which reencrypts the ballot (see Section 2.8.1) with a new secret parameter. The **randomizer** also signs the reencrypted ballot in order to prevent **voters** from forgoing randomization. Once the voting period is over, the **voting authority** moves the election to the tallying phase. The **sealers** each download all the encrypted ballots, homomorphically add them (see Section 2.8), decrypt their share of the final tally and post the result back to the **BC**. Once all the **sealers** have submitted their share, the decryption shares can be combined to produce the final decrypted tally.

Table 4.1: Cryptographic Algorithms in Provotum-RF

Algorithm	**Purpose**	**Definition**
Exponential ElGamal	Encrypt ballots	Section 2.8
N-of-n DKG	Decentralization of ballot decryption	[8]
Schnorr Proof	Proof of possession of private key	Section 2.10
Exponential ElGamal	Re-encryption of ballots	Section 2.8
Re-Encryption Proof	(Designated verifier) proof of ballot re-encryption	Section 2.10, [8]
Disjunctive Chaum-Pedersen Proof	Proof of ballot validity	Section 2.10
ElGamal Threshold Decryption	Decentralized decryption of final tally	Section 2.11, [8]
Chaum-Pedersen Proof	Proof of correct decryption	Section 2.10
RSA Blind Signature	Blind signing of voter credentials	Section 2.9.2, [8]
Sr25519 Signature	Signing of BC Tx	Section 2.9.1, [48]

The above paragraph glosses over the zero knowledge proofs used in the system. Table 4.1 lists the cryptographic operations used in Provotum-RF. During the DKG phase of election setup, the sealers each generate a NIZKP that they are in possession of the corrseponding private key of the published public key. When re-encrypting a voter's ballot, the randomizer generates a designated-verifier proof or re-encryption for the voter. Note that the proof is designated-verifier because otherwise, the proof together with the original ballot and parameters used by the voter would constitute a receipt of the vote, breaking receipt-freeness. The voter and randomizer jointly generate a proof of ballot validity for the re-encrypted ballot (i.e. proving to the system and anyone inspecting the

PBB that the ballot encrypts a 0 or a 1). When the sealers decrypt their share of the final tally, they post a proof of correct decryption to the PBB. Ballot secrecy is granted as long as at least one sealer remains honest. Note that the identiy provider cannot link ballots and users because it only signs blinded public keys. However, if a voter shares the blinding factor, the identity provider can identify the voter's ballots.

Note that the randomizer can be a bottleneck to the system since all voters must randomize their vote. Multiple randomizers can be deployed, but with a growing number of randomizers the probability increases that some randomizers are corrupted. Collusion between a randomizer and voter breaks receipt-freeness of the system. Using USB dongles for randomization alleviates this problem completely since each voter has their ballot randomized locally by their own dongle.

Chapter 5

Design of Provotum-HW

Provotum-HW introduces a USB dongle with its own CPU into the system. This provides an opportunity to perform cryptographic operations on a separate system that is resistant to malware infection. Storage and use of private keys can be delegated to the USB dongle, potentially yielding safer key storage. Since the dongle is low-cost it can be used at many distinct locations of the system.

This section discusses ways to modify Provotum-RF where using a USB dongle could improve the security. At the core of these modifications is the dongle, for which 6 functions are defined that offer varying security improvements but sometimes also usability tradeoffs. All functionality is intended to be implemented at the same time in dongle firmware so that different builds of the Provotum system can explore different choices of dongle functionality and their security and usability tradeoffs. It is even feasible to implement a system that uses different dongle functions for different elections in the same system depending on their importance and sensitivity, for example. Support for one proposed feature, panic passwords, although implemented in the dongle firmware, requires future work to find a solution to a cryptographic problem before end-to-end implementation can proceed.

We start this chapter by discussing the assumptions about an attacker's capabilities and the system running Provotum that motivate all the decisions in the subsequent evaluation of Provotum-HW features.

5.1 Assumptions

In general, we assume that attackers are computationally bounded and that common hardness assumptions, on which the used cryptographic algorithms are built, hold. Table 5.1 lists the assumptions upon which we construct the design of Provotum-HW.

We assume that the election is **sufficiently big**, so that multiple voters vote for each candidate. Otherwise it is trivial to deduct how some voters voted.

Table 5.1: Assumptions for Provotum-HW Design

Assumption	**Description**
Sufficiently big election	Number of voters
Coercion-resistance	Basic demographic assumptions required for CR definition
Duration of coercion	Coercion-free period exists
Capabilities of the coercer	Control of voter PC
Anonymity of the voter's communication channel	No matching of IP addresses
Impenetrability of the USB dongle	No extraction of secrets from dongle hardware
Trust in dongle FW	No trust in dongle software required

The problem with REV systems that provide receipt-freeness is that it is easy for a voter willing to sell her vote by just giving up her voting credentials and remaining passive while the vote buyer casts a vote. This is hard to prevent completely. A system can, however, improve over receipt-freeness according to the following argumentation. Vote sellers can be split into two categories:

1. A vote seller that sells their vote and doesn't care to cast their own vote in the election
2. A vote seller that sells their vote, but will override the ballot cast by the vote buyer and cast their own vote, if the system permits it

If the system permits voters of the second category to override a vote buyer's ballot, the price of a vote goes toward zero because a vote buyer won't know if he is dealing with a voter of the first or second category. This makes vote buying unattractive. Note also that coerced voters mostly belong to the second category because they are not giving up their vote on their own free will. Some coerced voters might belong to the first category due to fear of repercussions from a coercer and mistrust in the REV system, although a well-designed coercion-resistant REV system should protect the coerced voter in their counter-strategy. **Coercion-resistance** can only be attempted if a sufficient number of second category voters exist in the vote seller population.

It is a standard assumption in most literature that a coerced voter or vote seller is not observed by the coercer or vote buyer at all times, i.e. the **duration of coercion** is limited. Specifically, we assume that a time frame exists where a voter can cast their overriding ballot before the voting period ends. We also assume that the registration phase of an election is free from coercion (this is an important assumption of our suggested solution of using *panic passwords*). This is also a standard assumption in most literature.

We assume the coercer has the **capability** to take full control over the voter's computer. For example, the coercer can make the voter install an alternate voting software, which

can read the original voter software's local storage and make arbitrary requests to the rest of the system.

The data that a voter sends to the Internet does not identify a voter. However, with panic passwords and schemes that allow to interact with the voting system multiple times, for example matching of IP addressess can identify voters who interact with the system multiple times. We assume that a voter will use a virtual private network (VPN) connection or other measure to achieve **anonymity** in the face of network observers.

The USB dongle used to implement Provotum-HW is only equiped with a general purpose microcontroller unit (MCU) (see Section 5.5). We assume, however, that it is **impenetrable** by attack methods that exist for hardware, such as probing and decapping. We also assume that the firmware of the USB dongle cannot be infected by malware. Hence, private keys stored on the USB dongle are protected from theft and cryptographic operations are not vulnerable to side-channel attacks. Given the USB interface and application layer communication protocol between the hosts and the USB dongles, it is easy to implement equivalent functionality on specially hardened hardware or a dongle incorporating a Hardware Security Module (HSM). We further assume that modifying the firmware of the dongle is only possible after erasing all data stored on the dongle, including private keys representing voter and dongle identity. Flash memory protection like this is very common in MCUs.

We assume that the voter does not **trust the firmware running on the dongle** and that the dongle must provide proofs of correctness for all operations it performs. But we assume that the voter can be convinced, e.g. by inspecting the dongle hardware, that the dongle's only communication interface is the USB port (i.e. there is no hidden SIM card and antenna on the dongle, allowing the dongle to send a copy of the voter's data somewhere else). Otherwise, the same trust assumptions as with Provotum-RF hold (i.e. that no part of the system should trust any other) but that at least one sealer is honest.

5.2 Overview over Proposed Modifications

The most promising use case for deployment of a USB dongle is key management on dongles instead of voter clients. Keeping voter client private keys on USB dongles adds a strong protection against selling of credentials. Furthermore, if we pair this with the concept *panic passwords*, protection against coercion is added, albeit with a negative impact on usability and a yet-to-be-solved problem with identity provisioning. We also discuss the variant of the dongle encrypting the voter's plaintext vote instead of re-encrypting an already encrypted ballot. The USB dongle can also be used to manage sealer private keys. The usefulness of this is discussed below.

Another modification is to make the voter enter their vote directly into the USB dongle. This prevents man-in-the-middle attacks by malware on the voter's computer.

5.2.1 Managing Voter Credentials on the USB Dongle

Managing a voter's private key with which she signs the ballot's BC transaction on the USB dongle is a measure that adds protection against the selling of the voter's credential. Obviously, the voter can still sell the whole USB dongle, but this might prevent the voter from voting in many future elections, depending on the way dongles are distributed to voters and how replacements can be ordered. Additionally, selling credentials is a punishable offense in most jurisdictions and selling a dongle leaves more evidence than selling a purely digital copy of a private key. The BC transaction with the ballot must then be signed by the USB dongle while the voter still manages the task of broadcasting the transaction into the BC network. In order to register for an election with the identity provider, the voter just forwards communications between the USB dongle and identity provider, while the general flow of signing a blinded public key remains the same as with Provotum-RF. Note that the USB dongle needs to sign the encrypted ballot with some private key unknown to voters, just like the randomizer in Provotum-RF, in order to ensure that the voter really uses a dongle instead of managing credentials herself.

5.2.2 Managing Voter Credentials on the USB Dongle with Panic Passwords

The introduction of the general concept of panic passwords, as introduced by [71], used by Selections [53] (we also describe panic passwords in Section 3.1) allows adding coercion resistance to the system. The design of Selections [53] differs greatly from Provotum and their use of panic passwords is not directly applicable to Provotum. Instead, we attempt to define a basic design of panic passwords that can integrate with Provotum. Note that the design is not complete since challenges with identity provisioning were not tackled in this work. Instead, the panic passwords design attempts to derive the functionality that needs to be implemented on the dongle. We derive the design as follows:

- Since a coercer could observe the PBB after coercion for ballots cast by the same voter identity, a different keypair must be used for each panic or non-panic password. All keypairs must be registered with the identity provider (in a blinded fashion).
- Only the holder of the election's private key (i.e the sealers in a distributed manner) should be able to distinguish between panic ballots and normal ones. One way to achieve this is by definition of a panic token which is an encryption of a 0 or 1, as a boolean indicator whether a panic situation is present, with the public key of the election. When casting a ballot, a panic token is added to the transaction. The voter should possess exactly one panic token encrypting a 0 and the PBB should accept each token at most once, otherwise the voter could cast multiple valid votes.
- The voter must be able to register an arbitrary number of panic credentials. This allows her to pass an arbitrary subset of them to the coercer without a coercer being able to prove or disprove if the non-panic password is among them.

- A a protection against malware, the voter client software and USB dongle firmware should not need to track which panic token corresponds to a 0 or 1. Instead, a distinct password or PIN should be mapped to each token and the knowledge which PIN corresponds to the non-panic token should only be kept inside the voter's head.

The usage workflow of such a system would be as follows:

- The voter can request that the dongle create a given number of key pairs and return the respective public keys.
- The voter registers a number of public keys and an equal number of encrypted panic tokens with the identity provider. She must present a proof that exactly one panic token encrypts a 0. The identity provider signs the public keys and panic tokens individually. In order to maintain voter anonymity in front of the identity provider, blind signatures must be used like with Provotum-RF.
- After the registration phase, which we consider coercion-free, public keys and panic tokens are mapped as pairs against a PIN each and stored this way on the dongle. The pairing is necessary to prevent randomization attacks, which are defined to be among the attacks corercion-resistance must protect against.
- A voter selects the credentials to be used by entry of a PIN into the dongle. A voter that distrusts the dongle can keep a copy of the mapping of PINs to public keys and panic tokens and check that the dongle used the correct credentials for preparing the BC transaction. This can be achieved in a way that does not leak the number of PINs registered on the dongle, e.g. by hiding a list of hashes of mappings among additional random numbers with the same bit length as the hashes.
- To cast a non-panic ballot, the voter enters the non-panic PIN, to cast a panic ballot, the voter enters any of the panic PINs. The PIN and the parameters of the ballot are forwarded to the USB dongle. The dongle returns a signed BC transaction that can be broadcast to the network.
- The PBB rejects transactions that reuse any previously used public key or panic token.

Since we declared the assumption that the voter is not under coercion during the registration step, we might require that the voter show up personally at a city office for registration. Ideally, during the registration phase, the voter should not be required to bring anything more than the USB dongle and identity documents. This facilitates the assumption of a coercion-free registration phase over the alternative of the voter bringing an Internet-connected and potentially malware-infected personal computer.

5.2.3 Integration of Panic Passwords into Provotum

Discarding of ballots containing a panic token is easy if a mixnet is used because after mixing panic tokens are indistinguishable to their previous form before mixing. Then,

all ballots can be decrypted individually and those with panic tokens discarded. The use of panic tokens without a mixnet is limited. Obviously, if individual panic tokens are decrypted by cooperating sealers, it is easy to find out if a coerced voter posted a panic ballot. On the other hand, adding up the panic and non-panic tokens homomorphically and then decrypting the sum preserves the voters' privacy. Hence, one can find out if the number of panic ballots cast is less than the margin of the winning candidate. If that were the case, the election result would still be valid, regardless of how the coerced voters voted.

A problem exists with proving to the identity provider that only one encrypted panic token encrypts a 0, while using a blind signature scheme. The sigma protocol ZKP don't work directly with RSA blinded ElGamal ciphertexts (to prove a property of the unblinded plaintext). As mentioned in Section 2.9.2, many blind signature schemes exist and this problem might be solved with little protocol modifications in future work.

Additionally, such an implementation of panic PINs has a strongly negative impact on the usability of the system from the voter's perspective. Forcing the voter to remember multiple PINs is tedious for the voter and brings the risk of the voter forgetting them. Hence, due to the strong impact on usability and the unsolved problem with the ZKP during registration, end-to-end implementation of this feature in Provotum is neither possible nor desirable. Implementing support on the USB dongle firmware is not very difficult and allows future adoption of the feature without modification of the dongle FW.

5.2.4 Encrypting the Ballot on the Dongle

In Provotum-RF the ballot is encrypted on the voter client. The randomizer re-encrypts the ballot and together, the voter client and randomizer generate a proof of ballot validity: that the re-encrypted ballot encrypts a plaintext of 0 or 1, forming a valid vote. If the voter and randomizer cooperate, a receipt of the vote can be made.

This section describes a modification to the system so that the USB dongle receives a voter's plaintext vote and encrypts it. This simplifies the proof of ballot validity since the dongle alone holds all the random parameters used to encrypt the ballot. Concretely, the divertible proof generated interactively between the voter and randomizer can be replaced by a normal disjunctive Chaum Pedersen proof. Further, for the Provotum addition proposed in Section 5.2.6: entering a vote directly into the dongle, encryption of the plaintext vote must by definition be performed on the dongle.

Note that encryption of plaintext votes on the dongle is only possible because the dongle is directly connected to the voter's computer's USB port and we assume that the voter can ascertain for himself that the dongle is not equipped with some other communication interface (such as a cellular mobile connection) to leak the voter's plaintext vote to the Internet. With an online service like the randomizer in Provotum-RF this is not possible.

The protocol needs to ensure that the dongle uses a properly random parameter r to encrypt the ballot (see Section 2.1). If the dongle can choose r arbitrarily, a malicious dongle can easily select the parameter r in a way that allows a malicious party to obtain the same r and decrypt the ballot.

Process

The voter sends the plaintext vote to the randomizer for encryption. The randomizer encrypts the vote using a random parameter r that it doesn't disclose to the voter. The randomizer generates a ballot validity proof for the PBB, proving that the ballot encrypts a 0 for a no-vote OR a 1 for a yes-vote. The randomizer also generates a designated-verifier proof for the voter, proving that the plaintext given by the voter was correctly encrypted. Further, the randomizer generates a signature over the encrypted ballot to prevent the voter from generating a valid ballot by herself. The (DV) proof of correct encryption is given in Figure 5.1.

Randomizer		**Voter**
knows $M, (C, D), Y$		knows $M, (C, D) = (rG, rY + M)$
knows Q, the voter's public key		knows $r, Y, x, Q = xG$
$s, z_0, C_0 \leftarrow\$ \mathbb{Z}_n$		
	$A = z_0G - c_0Q,$ $B = sG, E = sY$ $\longrightarrow$	
		$c \leftarrow\$ \mathbb{Z}_n$
	$\longleftarrow$ c	
$c_1 = c - c_0$		
	$\longrightarrow$ $z_0, z_1 = s + c_1r, c_0, c_1$	
		$c \stackrel{!}{=} c_0 + c_1$
		$z_0G \stackrel{!}{=} A + c_0Q$
		$z_1G \stackrel{!}{=} B + c_1C$
		$z_1Y \stackrel{!}{=} E + (D - M)c_1$

Figure 5.1: Designated-verifier proof of correct encryption

Prevention Malicious Choice of Parameter r

We can force the dongle to encrypt the vote with a pseudorandom r using a similar approach as the Fiat-Shamir heuristic (see Section 2.10.2). First, the dongle encrypts the ballot with an arbitrary parameter r to obtain (C, D). Then the dongle hashes the encrypted ballot together with a nonce chosen by the voter to obtain another pseudorandom parameter r'. The dongle re-encrypts the ballot with r' to obtain (C', D'). This yields an equivalent ciphertext as if the dongle had encrypted the ballot with parameter rr'. The dongle returns the ballot (C, D) to the voter from which the voter can easily obtain (C', D') through the same process as the dongle and at the same time ensure that the

"randomization" step is properly applied to the ballot. The voter then broadcasts (C', D') to the PBB. Note that the ballot validity proof created by the dongle and later broadcast to the PBB must target (C', D').

Note that even if a malicious party knows the way the dongle chose r, dividing C' by r yields $r'G$, from which r' still cannot be retrieved due to the difficulty of the ECDL problem. Additionally, this scheme prevents the voter from getting a receipt of the vote, equivalently to the Provotum-RF randomizer, since the voter doesn't learn the random parameter used in the encryption of the ballot. Note that the voter nonce is crucial in this construction. If the ballot were hashed without the voter nonce, the parameter r' would still solely depend on r and the raw ballot, which could easily be reconstructed by a malicious party. This makes the voter nonce a sensitive piece of data that an honest voter should delete as soon as the ballot encryption process is complete. If the voter and randomizer dongle collude to leak the voter nonce and r respectively, a receipt of the vote can be formed, similarly to Provotum-RF.

Note that a similar heuristic has to be applied to all random data the dongle uses in publicly pushed values rG – i.e. also random parameters for challenges in ZKP, etc. This is the price to pay for entrusting the dongle with a plaintext vote under the assumption that the dongle cannot be trusted. One way to leak a vote to a malicious party in a ZKP challenge rG would be for the dongle to use a KDF keyed with some parameters to derive r in case the vote is a no–vote and use a KDF keyed with different parameters in case of a yes–vote. A malicious party knowing both KDF keys and their meaning could easily discover the leaked information.

Discussion

The security guarantees provided by this variant match those of Provotum-RF. The voter does not need to trust the dongle since she receives a proof that the dongle performed the encryption correctly. With this variant alone, vote selling is easily possible since the voter manages her own credentials (i.e. constructs the BC transaction). Even though the dongle handles a voter's plaintext vote, it cannot leak it since it has no additional communication interfaces beyond the USB port.

5.2.5 Managing Sealer Credentials on the Dongle

If a malicious party steals all the sealers' secret keys, it can decrypt non-homomorphically-added individual votes. Privacy of the votes is hereby lost. It cannot, however, modify the final tally, since it has to provide a decryption proof. The decryption proof cannot be faked despite knowledge of the private key. Publishing the decrypted votes of all voters would, however, greatly undermine trust in the voting system for future elections, require a costly rerun of the compromised election and potentially put some voters in uncomfortable or dangerous situations (for example, if a high pressure to vote a certain way is present from their peer group, political party, employer or other group they are affiliated with).

Since the sealers are not run by private individuals, care can be taken that they only run on hardened and properly administered machines, making theft of all keys unlikely. Additionally, an air gap could even be used for highly sensitive elections. (The private key would be generated on an offline machine, the corresponding public key transferred to an online machine. After the election the homomorphically-added ballots would be transferred to the offline machine for decryption and the decrypted share and proof back to the online machine for publication on the PBB). Since key generation and tallying only need to be done once per election, this could be an acceptable overhead for some elections.

Nevertheless, it is easy to implement the cryptogaphic algorithms that use the sealer's private keys to run on the USB dongle. Then, theft of all keys requires theft of all dongles, as well as extracting the private keys from the dongles, which we assume is impossible. Note that even though decryption of the tally requires a brute-force portion (because exponential ElGamal is used), that part does not need to be run on an USB dongle. Sealers decrypt the homomorphic sum of ballots to partial decryption shares on their dongles. The shares are posted to the PBB, from where the voting authority can reconstruct the final tally. Since implementing sealer operation analogous to Provotum-RF on the dongle is easy, it has still been added to the feature set of the dongle in Provotum-RF.

The problem that a single sealer can block decryption of the election tally can be mitigated by moving to a t-of-n DKG scheme. However, such schemes have a high message overhead as discussed in Section 2.11, making them an unfit candidate to implement on the USB dongles. Since moving the sealer operation to USB dongles provides no real benefits, keeping sealer operations on powerful machines with optional air gaps and instead implementing t-of-n DKG on those is preferable.

5.2.6 Entering Votes Directly into the Dongle

Votes could directly be entered into the USB dongle. The concrete dongle used for this work has a button, as well as a Near-Field Communication (NFC) antenna. Using the button, votes could be entered via some button-pressing pattern. Entering votes via NFC requires an NFC-compatible smartphone or other device and an app running on that phone.

Entering votes directly into the dongle prevents malware on the voter's computer from changing the vote before sending it to the dongle for randomization or encryption. Since high-profile elections attract government-level adversaries, malware designed to change an election outcome is not quite unthinkable. Note that denial-of-service attacks by malware are still possible since the only way to send the ballot to the PBB is via a the voter's PC's Internet connection.

To ensure that a voter doesn't have to trust the dongle, a designated-verifier proof of the correct encryption should still be verified on the voter's computer (this implies transmitting the plaintext vote from the dongle to the voter' computer, verifying the proof and asking the user if she really wanted to cast the presented vote). Doing that allows malware to find out how the voter voted, but not to modify the ballot's content.

Note also that if a voter mistakenly enters the incorrect vote into the dongle and repeats the process for the correct vote, potential malware on the voter's computer is now in possession of a ballot for both vote answers and can cast either as it sees fit. This could be mitigated in future work although any mitigation approach might further reduce usability.

5.2.7 Randomization

The dongle still implements randomization analogous to Provotum-RF. While this functionality is superseded by the alternatives described in this section, it was still implemented on the dongle for reference. Note that while [8] claims that using the strong Fiat-Shamir heuristic is impossible in its interaction with the randomizer, this is not the true. The dongle implements randomization using the strong Fiat-Shamir heuristic by caching the randomized ballot and generating the challenge in the divertible proof protocol based on the randomized ballot. Of course, the voter computes her response to the challenge based on the encrypted, non-randomized ballot. But the voter doesn't care how the challenge is generated. From the view of the voter, the challenge is only a pseudorandom number sent by the randomizer. When the final proof is generated and diverted to the randomized ballot, the randomized ballot to be hashed in the strong Fiat-Shamir heuristic proof verification matches the ballot hashed during proof generation.

5.3 Decision on the Provotum-HW Feature Set

I have presented a number of possible features for Provotum-HW. Table 5.2 shows a summary of these from a voter's perspective – i.e. how a voter's interaction with the system changes if a feature is used in the system. Table 5.3 shows advantages and drawbacks of each feature. Since some carry a negative usability impact or unsolved problems, exposure of all features on the dongle is the best way of achieving a flexible system in which the non-dongle system components can selectively target a specific dongle functionality. The only exception is ECC which is the only cryptography implemented in the dongle firmware.

Table 5.2: Usage of Provotum-HW Features from a Voter's Perspective

Feature	**Usage by the Voter**
Encrypting ballot on dongle	No difference in usage flow. Dongle has to be present to vote
Voter credentials on dongle	No difference in usage flow. Dongle has to be present to vote
Panic passwords	Dongle has to be present during registration and to vote. Define PINs during registration phase, enter non-panic PIN for voting, panic PIN in coercion situation
Sealer credentials on dongle	Dongle must be plugged into sealer machine
Entering votes directly into dongle	Dongle has to be present to vote. Votes are entered into the dongle by button press

Table 5.3: Assessment of Provotum-HW Features

Feature	**Advantages (over Provotum-RF)**	**Disadvantages**	**Notes**
Encrypting ballot on dongle	Faster than Randomization on dongle, simpler ZKP, no direct security advantage over Provotum-RF, required for "entering votes directly into dongle" (Section 5.2.6)		Care required so that dongle chooses proper parameters r
Voter credentials on dongle	Vote selling requires selling the dongle or over–the–shoulder cooperation with a coercer		
Panic passwords	Coercion-resistance	Negative impact on usability, requires mixnet	ZKP of panic token validity over blinded ciphertext not solved
Sealer credentials on dongle	Protection of private keys delegated to dongle hardware	Protection of private keys delegated to dongle hardware	
Entering votes directly into dongle	Protection against malware MITM	Impact on usability	
Switch to ECC	Higher performance	Requires modification of entire system	

5.4 Modified Decryption Proof

The decryption proof as defined by Provotum-RF contains 4 fields: d, u, v and s where d is the decrypted share. Hence, d should not be part of the proof, but part of the data required as input into the verification function. This modification was adopted in Provotum-HW, leaving the decryption proof with fields: a, b and z (corresponding directly to previous fields u, v and s).

5.5 Selection of the USB Token

Multiple options exist for a hardware token in an REV. A token has to be inexpensive so that wide deployment is economically feasible. Further, the token should be easy to develop for and easy for a voter to use. For a real-world deployment, protection of the cryptographic secrets stored in the hardware is very important and has to be prioritized higher than easy development and cost. In this work, the security requirement has been somewhat relaxed in order to facilitate quick development of a proof-of-concept. Table 5.4 gives an overview over the evaluated hardware platforms.

When it comes to secure, inexpensive end-user hardware, smart cards are probably the first such devices that existed. Smart cards are ubiquitous in the form of credit cards and SIM cards, but are also used for other authentication tasks such as access control to buildings and rooms. The typical physical interface of a smart card requires a smart card reader to interact with it. Earlier smart cards were programmed directly with firmware developed in assembly language, but more recently Java Card has enabled smart cards that are programmable with applications developed in a subset of Java. The Java Card operating system remains on the card and handles management of applications, such as adding and removing them. Of course, cards can be locked before shipping to the end-user to prevent any further modification. Unfortunately, closed-source proprietary software and Non-Disclosure Agreements (NDAs) imposed by distributors often pose a challenge for deployment of smart cards in academic projects.

Another class of end-user security hardware is the 2-factor-authentication token. Such a token is designed to add additional protection to logins by representing a second factor in addition to passwords. Ideally, login is not possible without physical possession of the token. For this use case, the universal 2-factor authentication (U2F) and FIDO2 (the name derives from “Fast IDentity Online”) standards have emerged [72]. 2FA dongles are almost always implemented as USB keys and as such are much more easily deployed by end-users. Some of them are equipped with a HSM to securely store secrets and accelerate cryptographic operations, but others simply use a general-purpose microcontroller. Depending on the thread model and benefit/cost ratio for an attacker, a general-purpose MCU is well-suited for the use case. Available options for 2FA dongles include the Yubikey and Solokey. Yubikey does publish the source code of some of the applications in their ecosystem but not all. Additionally, Yubikey has started to employ HSMs in their product, further moving them away from open source due to NDAs and other contracts.

Solokey is fully open source for both hardware and software. It employs a general-purpose 32-bit ARM MCU.

The Nitrokey is a hybrid solution between a 2FA token and additional custom cryptographic applications. Beyond 2FA logins, it implements e-mail encryption, encrypted storage on the Nitrokey, file-encryption for storage off the Nitrokey and management of cryptographic keys and certificates. It is open source for both hardware and software. Some Nitrokeys are fitted with an HSM where the HSM is specified to be an OpenPGP [73] card fitted to a SIM-card slot on the board. Hence, even the firmware on the smart card adheres to an open interface, although, since it is connected as an external SIM-card, it is not considered to be a direct part of the open-source Nitrokey system. The internal implementation of the OpenPGP card might contain closed-source code.

With Solokey being fully open source with an ARM MCU, the option arises to simply use a development kit for any one of the many 32-bit ARM MCUs. The development process would certainly be very similar if not a bit easier due to physical rails providing easy access to most pins of the MCU. Additionally, most development kits are already equipped with a debug chip, voiding the need for an external debug probe. On the other hand, the form factor and design centered around the USB port are arguments in favor of the Solokey.

The final choice of dongle for this project fell on the Solokey.

Table 5.4: Overview over USB Dongles

Dongle	**Advantages**	**Disadvantages**
Smart Card	Secure hardware	Closed-source & NDA, smart-card reader required
Yubikey	Secure hardware	Closed-source & NDA
Solokey	Open source	No HSM
Nitrokey	Open source, HSM support in some models	HSM only in some models and HSM needs to be bought separately
U2F Zero		Superseded by Solokey
General-purpose MCU dev-kit	Easy development, pin access, on-board debug chip	No HSM, less compact

Chapter 6

Implementation

This chapter gives an overview over the implementation decisions of Provotum-HW and documents the most important aspects. The implementation covers the Solokey firmware and modifications to other parts of the Provotum system. Additionally, parts of the Solokey development setup are described allow future work to reproduce it and to provide a basis for subsequent discussion.

6.1 Solokey Firmware

All Solokeys come with pre-flashed firmware whose feature set specifically covers the U2F and FIDO2 specifications. Since these standards are only concerned with authentication and data signing and does not cover ZKP, for example, the existing Solokey firmware cannot be used to cover the Provotum use case.

The Solokey's full software stack is made up of two components. The firmware and a bootloader. The default firmware allows booting into the bootloader via a special command sent via USB (the `solo-python` Python package is usually used for that). The bootloader can then accept new firmware images via USB and allows updating the firmware on the Solokey. With the normal Solokey variant firmware images need to be signed, but with the "hacker" variant unsigned images are accepted. During development of the Provotum dongle firmware, an attempt was made to keep the firmware compatible with the Solokey bootloader and Solokey tooling to update the firmware, but no satisfactory solution could be achieved. Appendix B discusses the reasons and options for future development.

The MCU on the Solokey is an STM32L432KC manufactured by ST Microelectronics. It is an 80MHz MCU implementing the ARM Cortex-M4 architecture.

ARM Cortex-M4 microcontrollers can all be debugged using the serial wire debug (SWD) interface. The SWD interface requires connection of 4 pins to an external debug probe: SWDIO (the bidirectional data pin), SWDCLK (the clock), GND (ground) and VTREF (a target voltage reference so that the debug probe knows which voltage represents a logic 1 on the target chip). The Solokey hardware layout exposes these 4 pins nicely on the

bottom as test points, to which wires can easily be soldered. Debug probes exist from many manufacturers. I used a Segger debug probe in my setup and Segger Ozone as the debug application. The debug interface allows stepping through code and stopping at breakpoints, as well as read and write access to all of the MCU's memory, allowing easy reflashing of firmware (see Appendix A).

I decided against building the Provotum-HW firmware on top of the open source Solokey firmware and opted to use `mbed-os` [74] (which is also open source) as a base instead. `mbed-os` comes with support for a large number of ARM Cortext microcontrollers, including Solokey's STM32L432KC. It further comes with many hardware drivers and libraries, allowing rapid development of custom embedded firmware (at the cost of higher flash memory and RAM consumption). Specifically, the abstraction level of the USB drivers and libraries on `mbed-os` is higher than using the STM32 software development kit (SDK) directly or building on top of the default Solokey firmware. Figure 6.1 gives an overview over the relationship of used components.

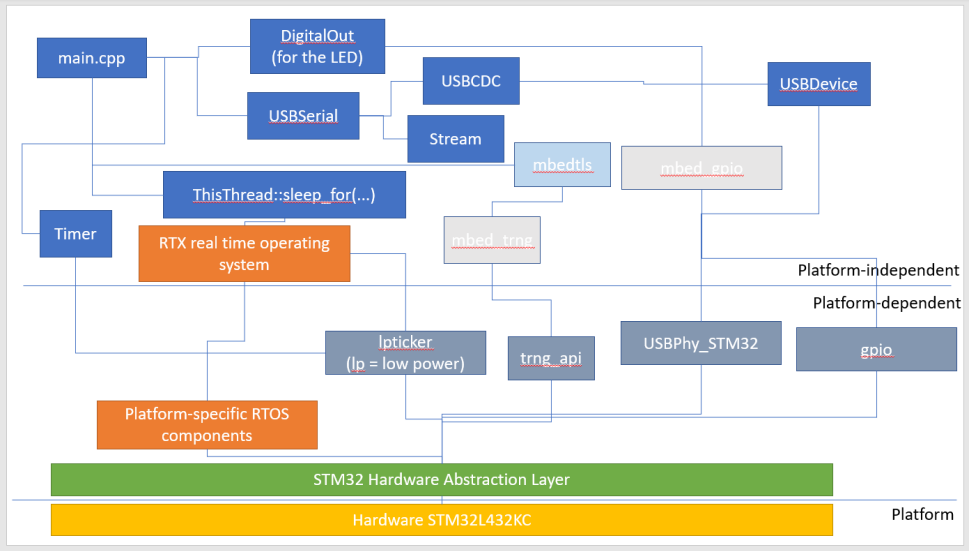

Figure 6.1: Relationship of Used Components of MbedOs

The Provotum-HW Solokey firmware implements USB communication over the USB communication device class (CDC) [44], one of the standardized device classes that work plug-any-play without installing any drivers on most platforms. USB CDC implements a bidirectional stream connection so that both nodes can send arbitrary unstructured bytes over the connection. Communication with the client and sending log messages to the host system are both implemented over the USB connection. Unfortunately, `mbed-os` doesn't support implementation of two USB device classes over a single USB port. Hence, logging and client communication are multiplexed over the same data channel. Details are documented in Section 6.1.4.

Table 6.1 shows the external libraries used for the dongle firmware. The application draws random numbers from the mbedtls CTR-DRBG random number generator implementation, which follows the NIST SP 800-90A standard. The mbedtls implementation instantiates an entropy pool by drawing high-quality random numbers from the hardware random number generator included in the MCU. The driver for the RNG is provided

Table 6.1: External Libraries Used in the Dongle Firmware

Name	Description
mbed-os	Real-time operating system, used for device drivers and higher level abstractions thereof, especially USB
mbedtls	Used for (secp256k1) EC point and big integer operations, SHA-512 and the entropy pool
libsodium	Used for EC operations on Curve25519 elements and encoding in the Ristretto format
strobe	Implementation of the Strobe protocol framework
tinycbor	Used to de- and encode the CBOR payloads of dongle messages
STL	Used for std::vector, std::array, std::shared_ptr, std::string and some functions from <algorithm>

by mbed-os. The entropy pool is reseeded at regular intervals from the hardware RNG. Libsodium's internal function for random numbers is rewired to draw random numbers from mbedtls CTR-DRBG. If the random numbers provided by the RNG hardware in the MCU are truly high-quality, the software entropy pool is not necessary. However, they can improve throughput because not every random byte is drawn from hardware and protect against cases where the hardware RNG returns biased or partially predictable bytes.

6.1.1 Flash Layout

Table 6.2 shows the flash memory layout of the Solokey as they are distributed by the manufacturer. The page size of the chip is 2048 Bytes. In order to support reflashig with different application firmware from the bootloader, the bootloader and bootloader data sections have to be preserved in flash and the application has to fit completely into the flash pages designated for the application (because the bootloader refuses to write to other pages during FW update). Theoretically, a small modification to the bootloader can enable flashing of applications that also make use of the application data pages for code, leaving 234 KB of flash usable for the application. Concrete attempts to make the Provotum dongle firmware cooperate with the Solokey bootloader have failed and options are discussed in Appendix B. Nevertheless the dongle firmware was configured to make use of 232 of the aforementioned 234 KB of flash. The last page of the flash memory is reserved for storage of public keys, panic tokens and signatures thereof.

6.1.2 Development Approach

The vast majority of embedded firmware projects are mostly written in C and some parts in assembly. C++ is sometimes used. Advantages of C++ are nicer abstractions and proper object oriented programming, such as constructors and destructors that are

Table 6.2: Solokey Flash Layout

Section	**Size (in Pages)**	**Start Address, End Address**
Bootloader	10	0x8000000–0x8005000
Application	98 (- 8 Bytes at the end)	0x8005000–0x8035FF8
Bootloader Data	1 (- 8 Bytes at the beginning)	0x8035FF8–0x8036800
Application Data	19	0x8036800–0x8040000

automatically called when objects are constructed or destroyed. Many features, such as exceptions, templates and the standard template library have high cost (mostly in binary code size footprint, sometimes in RAM usage and runtime cost). They should at most be used sparingly. Disadvantages of C++ include difficulties in ensuring that really only a subset of the language is used. For example, if a decision is taken not to use exceptions, all statically linked-in code must be checked that it doesn't use exceptions and some libraries cannot be used. Alternatives, such as templated result classes cannot be used due to their code size footprint. Rust is a new option where cross compilers exist, but it hasn't yet reached a maturity nor popularity level where major firmware developments are started in Rust.

The dongle firmware, including all dongle functions, was written in C++, since the mbed-os platform used also makes heavy use of C++. Dongle functions were first compiled natively for the x64 Windows platform of the author and tested in unit tests using the Googletest framework. This allowed for faster development since repeated flashing of the dongle was not required and dongle functions could be tested without reliance on the USB transport. Additionally, profiling could be achieved in a simple manner on the native platform. In contrast, profiling on an embedded platform is complex and requires high software overhead in the firmware or additional hardware. Additionally, the runtime of the dongle functions is much lower on the native platform, leading to less time spent waiting during tests. An additional advantage was that the implementation could be compared directly to other, already existing, implementations of the same functionality. For example, the Sr25519 signature scheme used by Substrate did not have a pure C or C++ implementation available. The most mature implementation is the *Schnorrkel* project also used by Substrate, which is implemented in Rust. Linking a static library build of Schnorrkel into the native dongle functions test environment allowed signing arbitrary data in Schnorrkel and verifying it in the own custom C++ implementation and vice versa. Some of the unit tests were later also run on the dongle using a custom test runner consisting of only a few lines of code.

For the development of the dongle firmware, it was decided to make use of some limited parts of the C++ standard template library (STL) to facilitate rapid development. The STL is not optimized for embedded software and makes heavy use of dynamic memory. Frequently in embedded software projects, heap memory is not used at all due to complications such as running out of memory non-deterministically, heap fragmentation and non-deterministic allocation times for heap memory. This is especially so in safety critical applications. However, many implementations of cryptographic algorithms, including

those in mbedtls, make use of heap memory to allocate space for big numbers and to parse nested structures. Operations on big integers would be cumbersome to implement and use without dynamic memory. Shared pointers were used extensively to reference-count instances of big integers and elliptic curve points. This decision favored rapid development over minimizing the memory footprint, but as discussed in Section 7.2 the overhead on the runtime is negligible. Additionally, templated types from the STL were carefully used with few or only one concretization to minimize the already considerable impact on the flash memory footprint.

6.1.3 End-to-End Integration of Cryptographic Operations

End-to-end compatibility of all the components of the Provotum system requires that all system components use the same cryptographic algorithms, the same group on which operations are performed, the same elliptic curve (in case of ECC), the same hash functions, etc. The dongle firmware was originally developed with the same underlying cryptographic groups as Provotum-RF, but the performance was too poor on the dongle (see Chapter 7). Hence, modifications were required in all other system components for end-to-end compatibility. Generally, ECC operations used in the system are implemented in the repositories `evote-crypto-ts` and `evote-crypto-rs`, providing equivalent functionality in Typescript and Rust. While, in theory, only modification of these two repositories is required to achieve full systemwide integration of ECC cryptography, in practice the codebase of most components of the system has to be modified because, for example, the field holding the finite-field parameters accociated with each election in Provotum-RF and distributed during communication has been dropped. The only cryptographic context is derived from the used curve and that is fixed for the whole system.

For ECC, the secp256k1 curve is used because it is among those supported by the *elliptic* Javascript library used to implement the required functionality on the client side, as well as the no_std-compatible Rust library *libsecp256k1* used to implement the functionality in Rust. For digital signatures (of BC transactions and artifacts created by the dongle) Sr25519 is used, which uses the Curve25519 curve, because this was already the default for BC transaction signing in Provotum-RF.

6.1.4 USB Command Set

USB commands are sent by encoding the full command and payload in Base64, prepending it with a dash '-' (ASCII character 45), terminating it with a newline character (ASCII character 10) and sending it over the serial channel of the USB CDC. Command responses are also Base64 encoded, dash-prepended and newline terminated. In contrast, log statements do not start with a dash and are not Base64 encoded. Therefore, each line is clearly distinguishable as a command or log statement and log statements are human-readable directly by observing the channel. CBOR [75] is used to serialize the payloads over USB (before the binary CBOR message is Base64-encoded). Payloads are (nested) structs of EC points and/or scalars. All nested structs are flattened all the points and scalars sequentially added to a CBOR message.

Table 6.3 lists the USB commands implemented in the dongle firmware. The table and rest of this section group the commands by the use case they are designed to cover.

Table 6.3: Provotum-HW Dongle USB Commands

Name	**Use Case**	**Design Section**
Utility		
Reboot Bootloader	Reboot into the bootloader for reflashing	
Ballot Randomization		
Randomize Initiate	Ballot re-encryption analogous Provotum-RF	
Randomize Finalize	Ballot re-encryption analogous Provotum-RF	
Encrypting Vote on Dongle		
Encrypt Plaintext Vote	Create ballot to be embedded in a BC transaction by the voting client	5.2.4
Creating Tx on Dongle		
Create Wallets	Create key pairs to create BC tranactions	5.2.1 / 5.2.2
Get Public Key for PIN	Obtain the public key for a previously generated key pair	
Store Signatures	Store identity provider signatures for key pairs already stored on the dongle	
Create Ballot Tx	Create ballot and embed in BC transaction, sign TX	5.2.1 / 5.2.2
Sealer Operation		
Generate Sealer Key Pair	Generate keypair for election	5.2.5
Sealer Partial Decryption	Partially decrypt final tally	5.2.5
Vote on Dongle (With Ballot Encryption Only)		
Vote on Dongle and Encrypt Ballot	Expect user vote by key press & encrypt ballot	5.2.6, 5.2.4
Vote on Dongle (Full Tx Generation)		
Vote on Dongle, Create and Sign Tx	Expect user vote by key press & prepare signed ballot	5.2.6, 5.2.1 / 5.2.2

Ballot Randomization

The Randomize Initiate (see Figure 6.3) and Randomize Finalize (see Figure 6.4) command provide Provotum-RF functionality on the Solokey. Ballots are encrypted by the voting client and re-encrypted on the Solokey. The difference to Provotum-RF is the use of ECC and the use of the strong Fiat-Shamir heuristic. Figure 6.2 shows the messages exchanged in this variant when casting a vote.

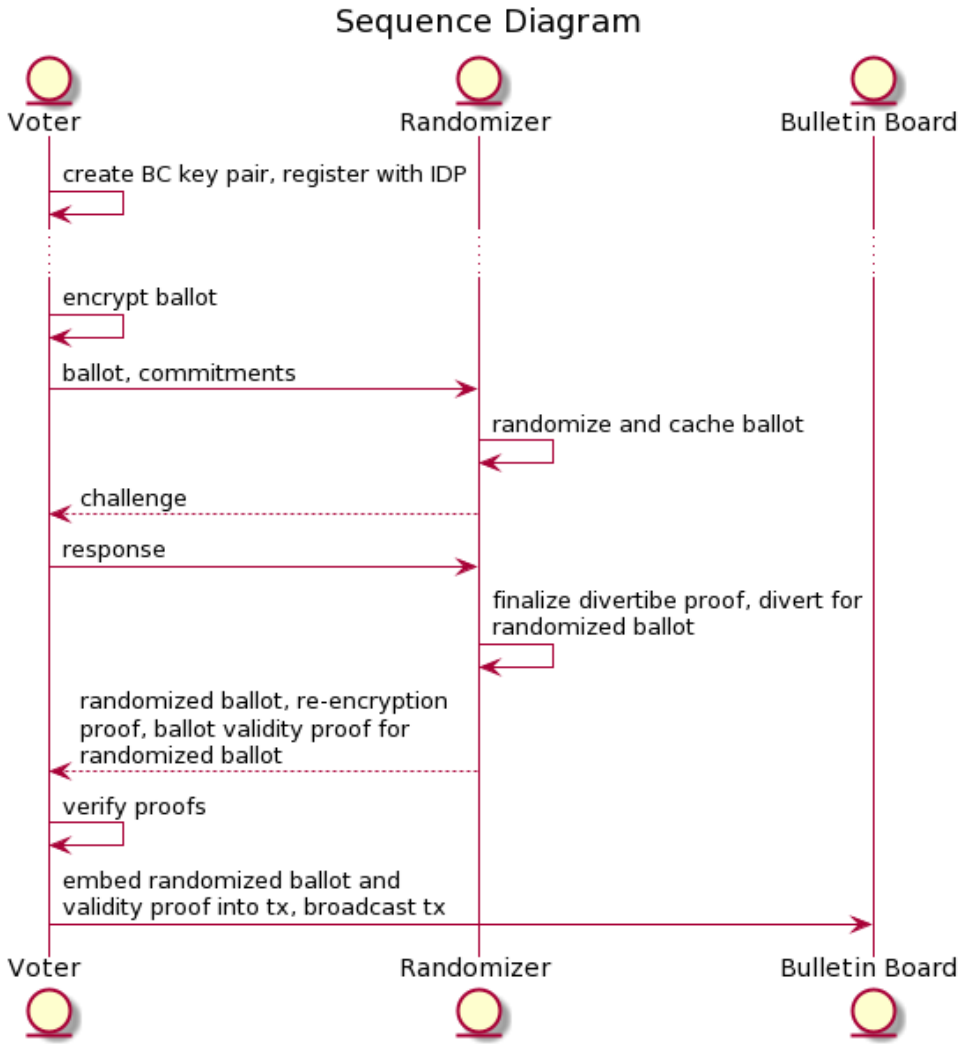

Figure 6.2: Message Sequence Chart for the Randomize Functionality

```
// Request payload
{
    encryptedBallot,
    bcPublicKey, // key of voter's BC account
    ballotValidityProof,
    electionPublicKey,
    voterCommitments,
    voterPublicKey
}

// Response payload
{
    challenge,
    blindCommitments
}
```

Figure 6.3: Randomize Initiate Command

```
// Request payload
{
    proofResponse
}

// Response payload
{
    reEncryptedBallot,
    designatedVerifierProofOfReEncryption,
    divertibleBallotValidityProof,
    ballotSignature
}
```

Figure 6.4: Randomize Finalize Command

Encrypting Vote on Dongle

The Encrypt Plaintext Vote command (see Figure 6.6) is used to encrypt a ballot on the Solokey. The result is an encrypted ballot (and several proofs) that has to be embedded into a BC transaction and broadcast by the voting client. This corresponds to the variant from Section 5.2.4. Figure 6.5 shows the messages exchanged in this variant when casting a vote.

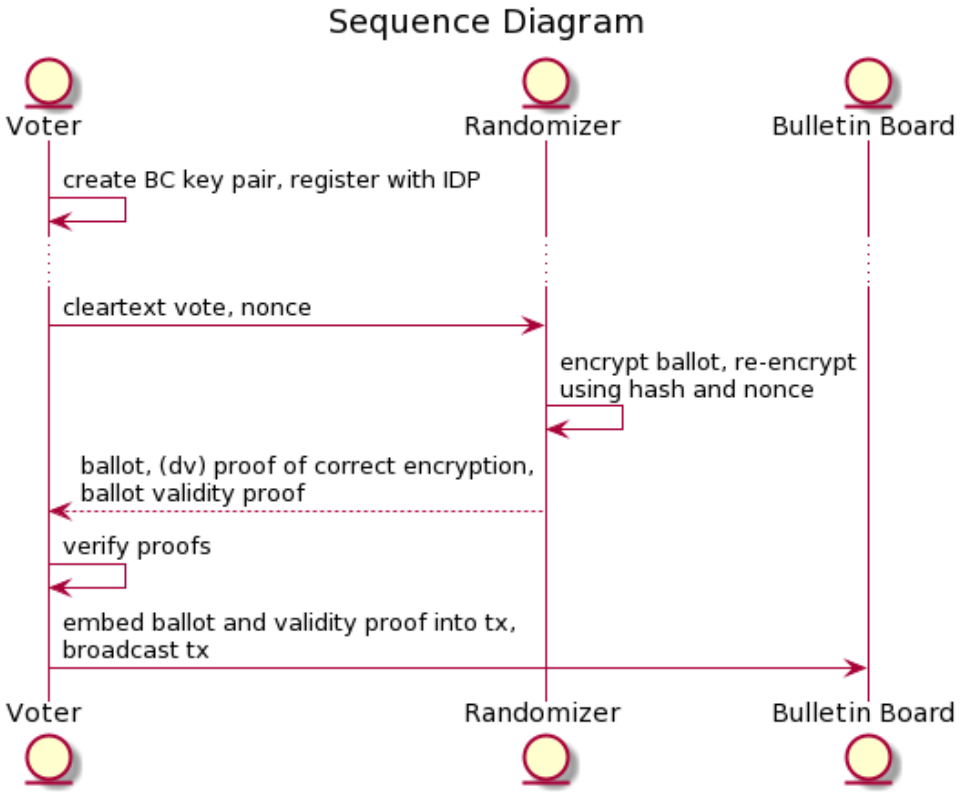

Figure 6.5: Message Sequence Chart for the Encrypt Vote on Dongle Functionality

```
// Request payload
{
    vote,
    electionPublicKey,
    bcPublicKey, // key of voter's BC account
    voterNonce,
    voterPublicKey
}

// Response payload
{
    encryptedBallot,
    designatedVerifierProofOfCorrectEncryption,
    proofOfBallotValidity,
    ballotSignature
}
```

Figure 6.6: Encrypt Plaintext Vote Command

Creating Tx on Dongle

Creating a BC transaction on the dongle that embeds an encrypted vote is realized through the Create Wallets, Create Ballot Tx, Get Public Key for PIN and Store Signatures commands. Figure 6.7 shows the messages exchanged in this variant when casting a vote.

Figure 6.8 lists the payloads of the wallet management commands. This covers the features from Sections 5.2.1 and 5.2.2. If a voting client wants to implement a single credential managed on the Solokey without the use of panic passwords, analogous to Section 5.2.1, the voting client should just use a single default pin such as `0000`. If `pins` contains a list of PINs, a keypair is created for each PIN, allowing an implementation of panic passwords coercion resistance. The implementation of panic passwords in the dongle firmware only requires proper passing of the "panicTokenFeatureUsed" when storing identity provider signatures and panic tokens, as well as embedding of a panic token in the BC transaction if the feature is used (as denoted by the flag stored together with the key pair and identity provider signatures).

Figure 6.9 shows the payloads of the Create Ballot Tx command and its response. Hence, a signed transaction for the PBB containing a ballot that encrypts the voter's given plaintext vote is obtained. If a voting client not implementing panic passwords registered a single credential using a default PIN, it should pass the same pin in the corresponding parameter. Note that the voting client can and should verify that the correct private key was used to sign the transaction by keeping its own mapping of PINs to public keys.

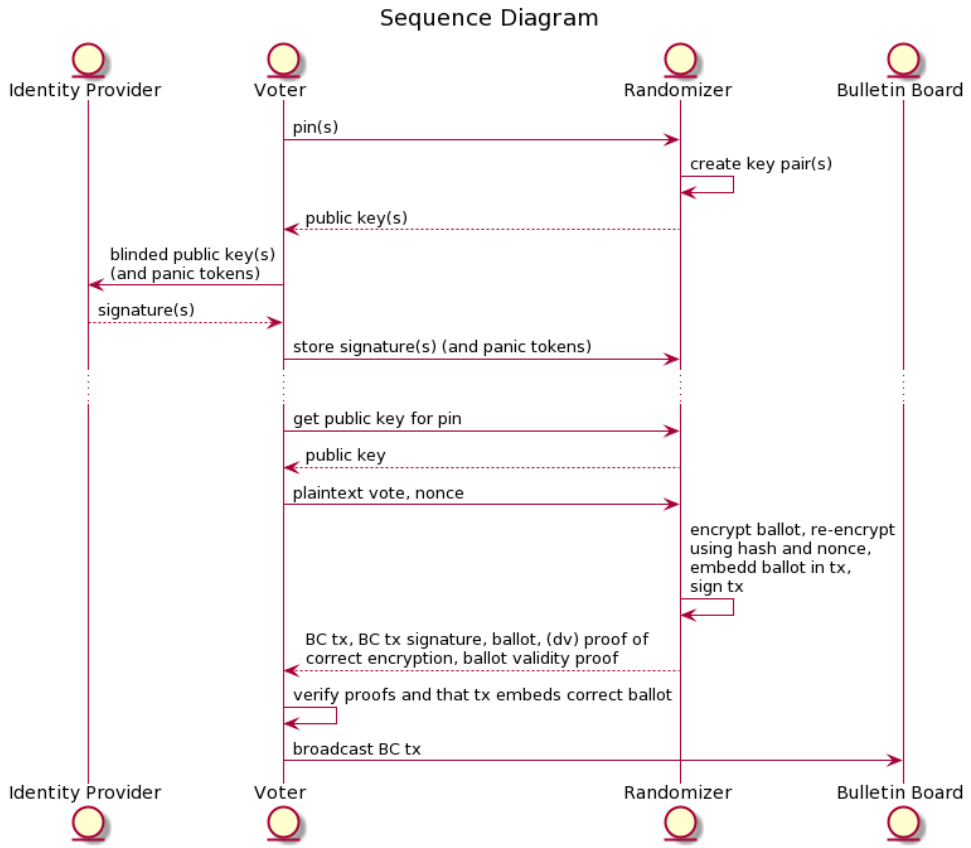

Figure 6.7: Message Sequence Chart for the Create Ballot Tx Functionality

```
// Create public keys request
{
    pins: number[]
}

// Create public keys response
{
    "success" / "fail"
}

// Erase public keys request
{
    // empty, erases all keys
}

// Get public key for PIN request
{
    pin
}

// Get public key for PIN response
{
    publicKey,
    panicTokenFeatureUsed : boolean,
    panicToken, // can be empty
    publicKeySignature,
    panicTokenSignature
}

// Store public key signatures request
// implemented on dongle, integration test
// required
```

Figure 6.8: Wallet Management Commands

```
// Request payload
{
    vote,
    electionPublicKey,
    voterNonce,
    pin, // can be 0000
    voterPublicKey,
    usePanicTokenFeature: boolean,
    txPrototypePart1,
    txPrototypePart2,
    electionId,
    txPrototypePart3
}

// Response payload
{
    intermediateBallot,
    proofOfBallotValidity,
    designatedVerifierProofOfCorrectEncryption,
    ballotSignature,
    ballotTx,
    ballotTxSignature
}
```

Figure 6.9: Create Ballot Tx Command

Sealer Operation

The Generate Sealer Key Pair and Sealer Partial Decryption command are used for sealer operation. Figure 6.10 shows the messages exchanged in this variant when casting a vote.

The Generate Sealer Key Pair command 6.11 is used to generate a key pair for an election if the Solokey is attached to a sealer. This corresponds to the feature in Section 5.2.5.

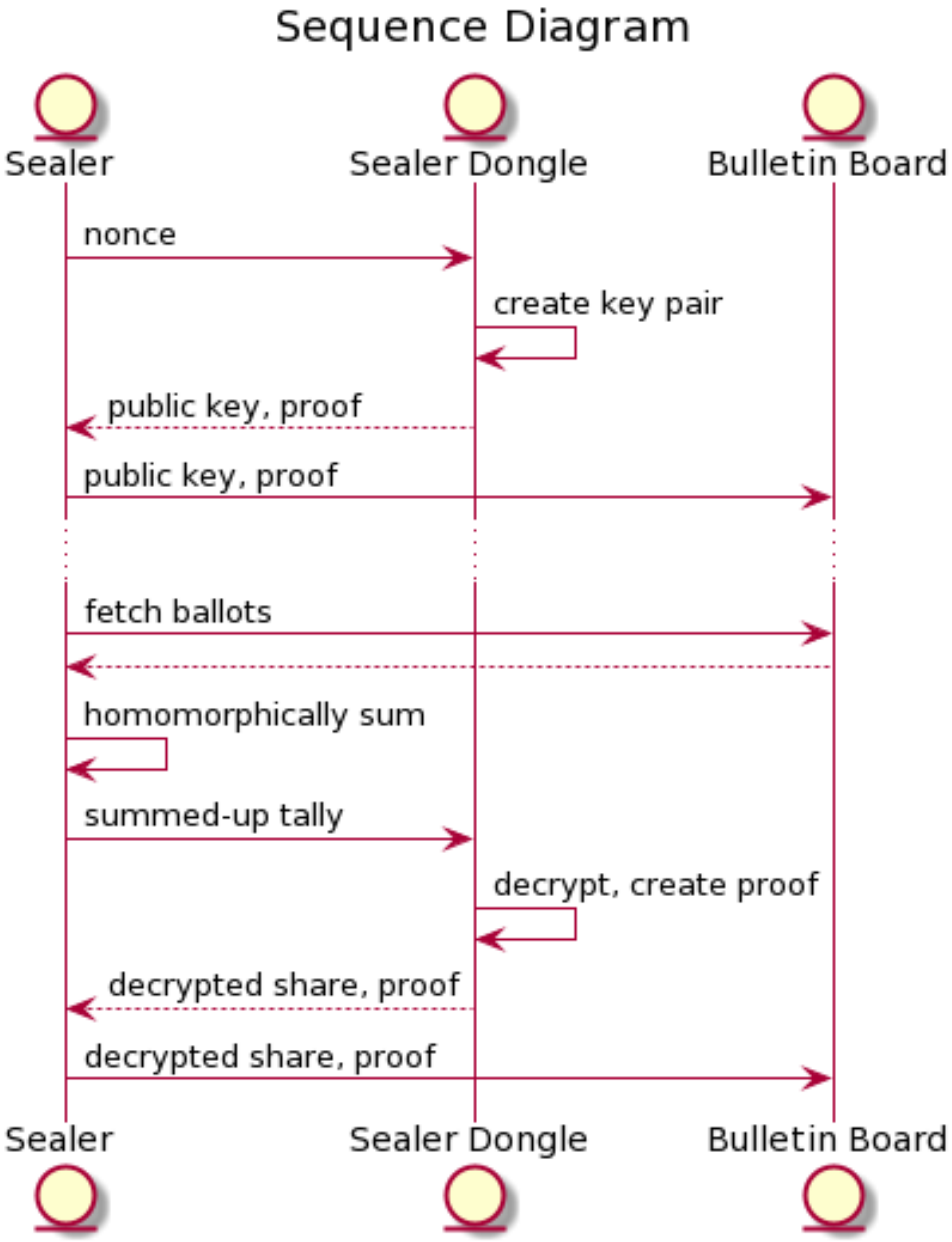

Figure 6.10: Message Sequence Chart for the Sealer Functionality

The Sealer Partial Decryption command 6.12 is used partially decrypt the final tally on a Solokey attached to a sealer.

```
// Request payload
{
    voterNonce,
    bcPublicKey // key of sealer's BC account
}

// Response payload
{
    publicKey,
    proofOfPossessionOfPrivateKey
}
```

Figure 6.11: Generate Sealer Key Pair Command

```
// Request payload
{
    encryptedTally,
    bcPublicKey // key of sealer's BC account
}

// Response payload
{
    partiallyDecryptedTally,
    decryptionProof
}
```

Figure 6.12: Sealer Partial Decryption Command

Vote on Dongle (With Ballot Encryption Only)

The Vote on Dongle and Encrypt Ballot command (see Figure 6.14) is used to allow the voter to enter a vote directly into the Solokey and receive the encrypted ballot. Once the command is received, the dongle waits for a user button interaction for 90 seconds. Short button presses alternate between setting a yes– and a no–vote, indicated by the LED. A long button press confirms the vote and initiates encryption thereof. This corresponds to the combined variant from Sections 5.2.4 and 5.2.6. Figure 6.13 shows the messages exchanged in this variant when casting a vote.

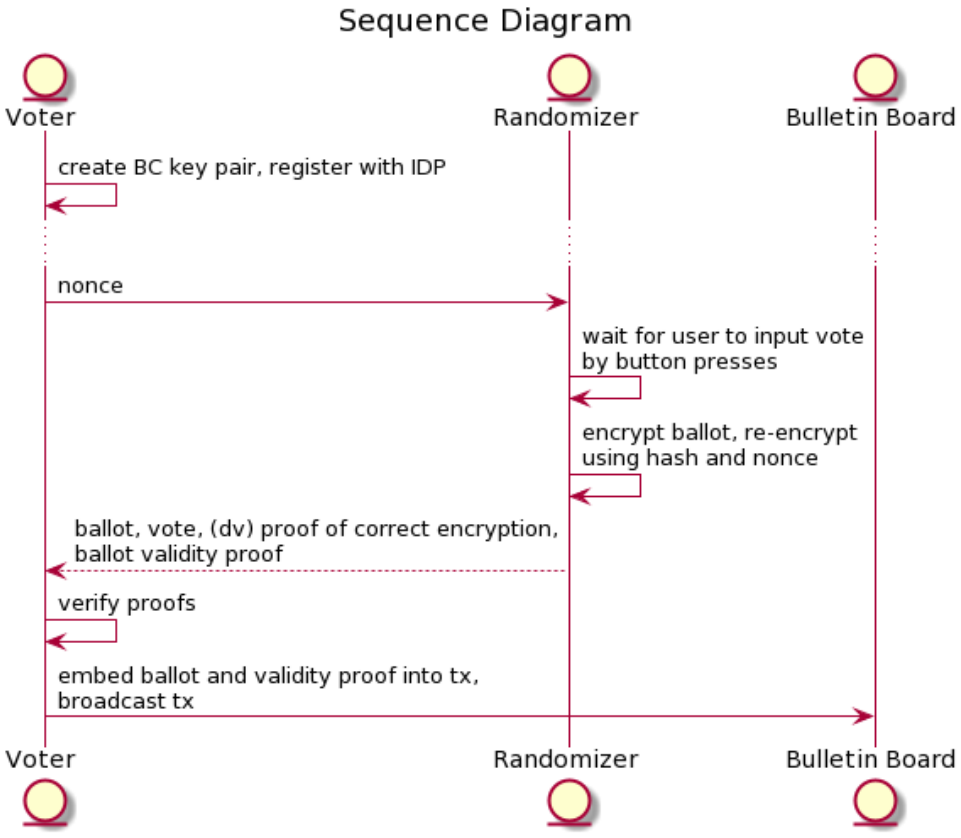

Figure 6.13: Message Sequence Chart for the Vote on Dongle and Encrypt Functionality

```
// Request payload
{
    electionPublicKey,
    bcPublicKey, // key of voter's BC account
    voterNonce,
    voterPublicKey
}
// Response payload
{
    encryptedBallot,
    designatedVerifierProofOfCorrectEncryption,
    proofOfBallotValidity,
    ballotSignature
}
```

Figure 6.14: Vote on Dongle and Encrypt Ballot Command

Vote on Dongle (Full Tx Generation)

The Vote on Dongle, Create and Sign Tx command (see Figure 6.15) is used to allow the voter to enter a vote directly into the Solokey and obtian a signed transaction for broadcasting to the Substrate network. Once the command is received, the dongle waits for a user button interaction for 90 seconds. Short button presses alternate between setting a yes– and a no–vote, indicated by the LED. A long button press confirms the vote and initiates encryption thereof. This corresponds to the combined variant from Sections 5.2.4 and 5.2.1 or 5.2.2. Figure 6.16 shows the messages exchanged in this variant when casting a vote.

```
{
    electionPublicKey,
    voterNonce,
    pin, // can be 0000
    voterPublicKey,
    usePanicTokenFeature: boolean,
    txPrototypePart1,
    txPrototypePart2,
    electionId,
    txPrototypePart3
}

// Response payload
{
    intermediateBallot,
    proofOfBallotValidity,
    designatedVerifierProofOfCorrectEncryption,
    ballotSignature,
    ballotTx,
    ballotTxSignature
}
```

Figure 6.15: Vote on Dongle, Create and Sign Tx Command

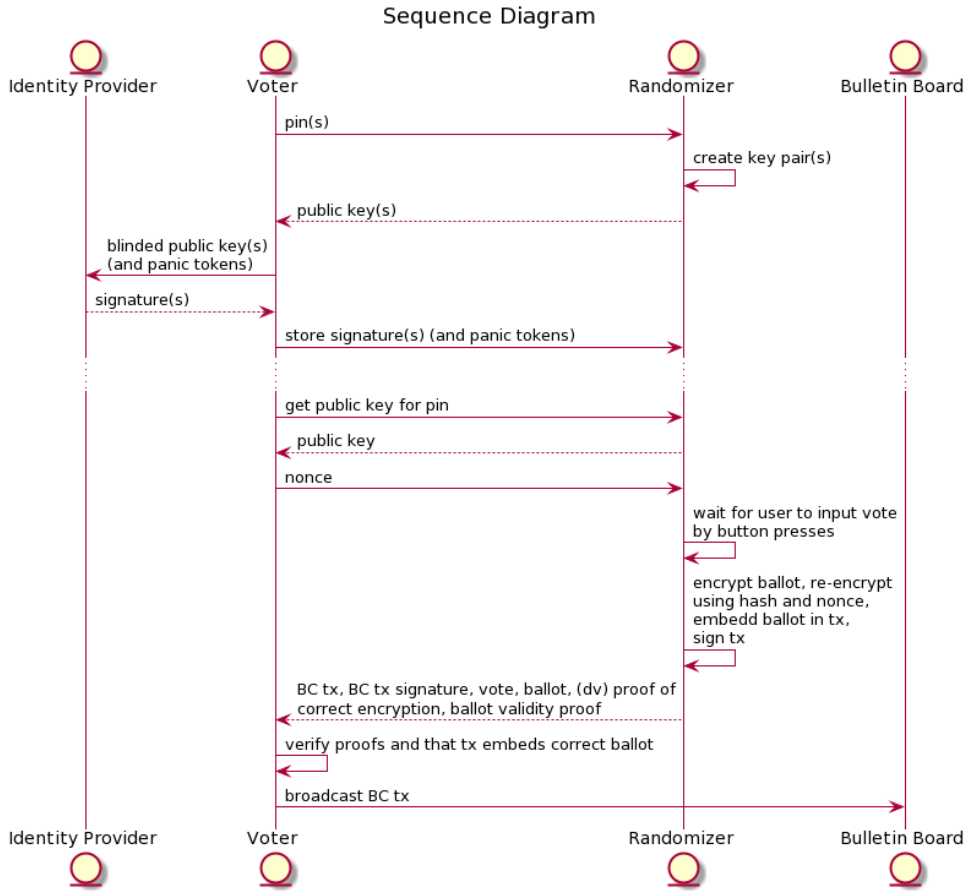

Figure 6.16: Message Sequence Chart for the Vote on Dongle and Create Tx Functionality

6.2 Integration of Provotum Components

Besides the dongle firmware, the Rust Provotum cryptography library and the Provotum Substrate chain codebase were modified to accept the new ECC-based ZKP. The fields of the proofs do not differ from Provotum-RF except that they contain EC points or 256-bit scalars instead of 2048-bit scalars. One exception is the proof of decryption, whose number of fields was decreased from 4 to 3 (see Section 5.4). Hence, the functionality of the modified component remains the same and only the inner workings change. A build of this system component successfully accepts dongle-signed transactions and successfully verifies the proofs of the submitted data, achieving an implementation of end-to-end integration of Provotum-HW features. A Node-JS-based "client" application was used as the link between the dongle and the Substrate BC. It implements its functionality in the form of integration tests, uses the `serialport` library to communicate with the dongle over the CDC interface, exercises all dongle functions and interacts with the Substrate BC, hence offering a fast way to validate dongle functionality and check for regressions during development.

It was decided to consider integration into the codebase of the other Provotum components (i.e. the sealer, identity provider, voting authority and voter web-application) as out of scope for this project since the amount of implemented code was already high and to focus

instead on documentation of pitfalls encountered during this work so that future work can extend on it with less effort. Additionally, since the "client" (integration test) application contains all the necessary code for dongle interaction, it can be used as a template for further integration.

Table 6.4 lists the dongle functions, the required integration modifications to the remaining components of Provotum for full production integration, as well as the level of interaction implemented in the end-to-end PoC.

Table 6.4: Integration Overview of Provotum Components

Use Case / Function	**Integration Required**	**Integration Completeness**
Randomize Ballots	• USB communication with the dongle must be implemented in the voting client. • The voting client must be adapted to stop using the randomizer web service and use the dongle instead. • Some minor modifications to the messages passed. • The divertible proof must now make use of the strong Fiat-Shamir heuristic.	Complete in standalone client demonstrator.
Encrypt Votes on Dongle	• USB communication with the dongle must be implemented in the voting client. • The voting client must be adapted to stop using the randomizer web service and let the dongle encrypt votes instead. • Verification of correct dongle behavior must be implemented.	Complete in standalone client demonstrator.
Create Ballot Tx on Dongle	• USB communication with the dongle must be implemented in the voting client. • The voting client must be adapted to stop managing its own BC key pair and signing its own transaction. • Verification of correct dongle behavior must be implemented.	Complete in standalone client demonstrator. Txs (signed on the dongle) are accepted by the PBB and all proof verifications pass.

Panic Passwords	• The Identity Provider needs to verify the validity of panic tokens, sign all public keys and tokens and fix the pairing of tokens and public keys. • The Substrate transaction verification mechanism needs to prevent panic token reuse. • A mixnet should be employed for tallying.	No implementation attempted beyond support in the dongle firmware.
Dongle Signs Artifacts it Produces	• The PBB should verify the signature and reject unsigned artifacts.	Implemented in dongle firmware. The standalone client demonstrator deserializes and verifies the signature from the dongle response. The PBB code does not check the signature.
Sealer Cryptography on Dongle	• USB communication with the dongle must be implemented in the sealer application. • The sealer application must be adapted to stop managing its own key pair and defer these operations to the dongle instead.	Complete in standalone client demonstrator.
Entering Votes Directly into Dongle	Same as “Encrypting Votes on Dongle” and “Create Ballot Tx on Dongle”.	Complete in standalone client demonstrator. If a dongle-signed tx is returned, it is accepted by the PBB and all proof verifications pass.

ECC	• All system components must be switched to ECC.	Implemented in dongle firmware. The evote-crypto-ts repository has been extended by all required operations based on ECC. The standalone client demonstrator makes use of the new implementations in evote-crypto-ts and is fully ECC–compatible. The evote-crypto-rs repository has been extended and all operations used by the PBB have an ECC implementation. The PBB implementation makes use of the new implementation and is fully ECC–compatible.

Chapter 7

Evaluation

This chapter describes the evaluation performed on Provotum-HW. The evaluation focuses purely on the operation of the dongle and its ability to offer the required functionality. Since the dongle is a very resource-constrained device (it has 256 KB of flash memory, 64 KB of RAM and an 80 MHz CPU), operations that may run efficiently on other Provotum system components may run unacceptably slowly on the dongle. On the other hand, if an operation runs sufficiently efficiently on the dongle, it is generally guaranteed that it will also perform well on other system components. Since – excluding switching to ECC – Provotum-HW does not make any fundamental changes to the provotum system (such as switching to a mixnet), the performance of the remaining system components beside the dongle is generally equivalent to Provotum-RF. Table 7.1 lists the evaluation activities performed on Provotum-HW.

Table 7.1: Evaluation Activities on Provotum-HW

Activity	**Goal**
Timing the runtime of dongle functions	Assess agreeability with potential usage by voters. In conjunction with profiling: assess optimization potential/success
Profiling	Identify longest running functions and assess optimization potential
Discovery of large functions	Identify largest functions to assess largest-gain optimization opportunities to reduce flash-memory footprint
Regularly printing the maximum stack usage of each task	Identify the risk of stack overflows, identify the RAM requirements of the firmware
Regularly printing the maximum heap usage	Identify the RAM requirements of the firmware

7.1 Timing the Runtime of Functions

A very basic evaluation consists of timing the runtime of dongle functions, such as encrypting or randomizing a ballot. Of course, this does not provide a high level of granularity, i.e., it does not reveal exactly what subfunctions use a lot of time. Profiling is used for higher granularity. Nevertheless, timing the function runtime gives a quick idea whether the total runtime of a function is within a range that would be acceptable for most users.

Table 7.2: Runtime of Dongle Function Unit Tests

Test	**Runtime with FF Cryptography (ms)**	**Runtime with ECC (ms)**
Randomize	134308	69080
Encrypt Ballot	45913	28665
Create Ballot Tx	46022	28686

Table 7.2 shows the runtime of the dongle function unit tests running on the dongle. Note that the reported runtimes include the functionality that is supposed to run on the dongle, as well as the verification of the produced artifacts (which usually runs on the client or in the PBB nodes), roughly doubling the runtime of each test. But even with all runtimes halved, this data reveals that the functions run unacceptably slowly if finite field crypto is used. Profiling (see Section 7.2) revealed that no optimization could bring this within an acceptable range, hence motivating the change to ECC. The runtimes with ECC, while still long, are more in an acceptable range for voters using the dongle. It is interesting to note that randomizing a ballot takes 69 seconds while encrypting a plaintext vote and even creating a ballot Tx each take only 29 seconds. Since these functions provide other advantages, such as protection against malware MITM attacks and vote selling, it only follows that their usage is highly recommended.

Equivalent unit tests were also written in Typescript (see Table 7.3) that exercise all dongle functions, as well as the USB transport and verify the produced artifacts in Typescript code. This ensures that both the dongle and client implementation are correct, that the transport and serialization of parameters works and offers a better measurement of the runtime of dongle functions on the dongle since the verification part is not run on the dongle but in Typescript where the runtime is negligible compared to the dongle.

Table 7.3: Runtime of Dongle Functions Exercised by the JS Test Client

Test	**Runtime with ECC (ms)**
Randomize	41739
Encrypt Ballot	12462
Create Ballot Tx	12696

7.2 Profiling

Profiling was achieved using gprof, the GNU Profiler, on a native build of the dongle functions (see Section 6.1.2). Table 7.4 shows the most expensive functions when using finite-field cryptography sorted in descending order.

Table 7.4: Most Expensive Functions with FF Cryptography

Function	**Runtime (Percent)**	**Runtime (s)**
mpi_mul_hlp	51.4	1.1
mpi_montmul	6.07	0.13
mbedtls_mpi_shift_r	5.61	0.12
mpi_sub_hlp	3.74	0.08
mbedtls_mpi_add_abs	2.8	0.06

This data shows that with finite field cryptography, the combined execution of all dongle functions spends 51% of the time in the mpi_mul_hlp function. This function is part of mbedtls and is used in the multiplication of big integers. The next two functions belong to the profiling framework itself. The next most expensive function of the dongle functions only runs for 6% of the time and also belongs in the domain of big integer multiplication in mbedtls. This indicates that little potential for optimization exists since the mbedtls implementation is quite optimized already and the big integer multiplications are required for the dongle functionality. Code review revealed no way to drastically reduce the number of executed multiplications. This motivates switching to ECC as the best way of achieving acceptable runtime performance.

Table 7.5: Most Expensive Functions with ECC

Function	**Runtime (Percent)**	**Runtime (s)**
_mcount_private	46.15	0.36
__fentry__	15.38	0.12
mbedtls_mpi_mul_mpi	8.97	0.07
mbedtls_mpi_cmp_mpi	7.69	0.06
mpi_mul_hlp	5.13	0.04
mbedtls_mpi_copy	3.85	0.03
myFree	3.85	0.03

Table 7.5 shows the runtime of ECC dongle functions sorted in descending order. Here, the most expensive functions are from the profiling framework itself and all other functions contribute to under 10% of the runtime each. The first 3, again, correspond to big integer multiplication. Among the next few functions are functions related to copying data and

memory allocation. Optimizing in that domain could yield some improvement, but the profiling numbers suggest a potential for optimization under 10%. As discussed in Section 7.5, other optimizations, such as switching to a different elliptic curve, have a higher potential for large performance improvements. Even though shared pointers were used, profiling reveals that the runtime impact of reference counting is at a rounded 0% of total runtime, making the switch to another memory model not a worthwhile optimization.

7.3 Discovery of Large Functions

The flash memory footprint of the dongle firmware is very high. During development, versions of the firmware started exceeding the 234 KB limit of flash memory usable for the application (see Section 6.1.1). Large functions were identified from the firmware binary using *nm* and *puncover*. The analysis revealed that most dongle functions were among the largest functions due to instructions from inlined C++ STL library functions. Stripping the STL from the firmware was considered but rejected as a first-line solution because it was unclear, what other factors were contributing to the large binary size and it would have slowed development considerably. Compiling the binary with the -fno-inline compiler flag, which prevents the compiler from inlining functions, yielded a significantly smaller binary (i.e. around 130KB instead of 280KB at the time of the measurement) that also fit well into the available flash memory. After disabling inlining, the first 7 largest functions were in cryptography 3rd party libraries. Only in the 8th place was a function developed by me, which used 1980 Bytes of flash, or 0.8% of the available flash memory. Further optimization was hence stopped. Not using inlining has a negative impact on the runtime performance of the application and a future attempt at reenabling it should be made. Concrete steps are discussed in Section 7.5.

7.4 Regularly Printing Maximum Stack and Heap Usage

To evaluate the RAM consumption of the application, regular print outs of the maximum used stack memory per task and the maximum used heap memory were added to the firmware. These reveal that the firmware is well within the limits of RAM on the MCU. The maximum stack usage stats were used to size the stacks for each task optimally (i.e. including a small safety margin, but not too much). Heap analysis revealed that more than 25 KB of heap memory remain free at all times.

7.5 Evaluation and Optimization Discussion

In summary, due to flash memory constraints, inlining was disabled, yielding a runtime performance penalty, but allowing the firmware to fit into flash. Switching to ECC brought the runtime of the dongle functions into an acceptable range. Many optimizations can be attempted in future work to reduce the runtime.

In order to be able to re-enable inlining, the flash memory requirement of the whole firmware has to be reduced. Purging the STL from the firmware is a first step in achieving that. Mbed-os also has a high flash footprint and could be removed. The dongle functionality is single-threaded except for the USB driver and library. Hence, the only considerable part of porting would be the switch to a different USB software stack. It might be a good idea, however, to keep the mbed-os implementation of SharedPtr and use that instead of STL's shared_ptr. The mbed implementation has a lower flash footprint and has no hard-to-remove dependencies on other mbed functionality. Switching to manual memory management offers no useful performance benefit as indicated by profiling.

Switching to a different elliptic curve (currently secp256k1 is used) might provide further performance and flash footprint improvements. If the libsodium implementation of Ristretto encoded points on the Curve25519 curve were used, the dependency on mbedtls could be dropped and libsodium used as the main cryptography library, freeing up the flash used by mbedtls. Unfortunately, libsodium is also not very lean, including 30KB of curve constants. But another mature C implementation of Curve25519 operations with Ristretto points does not seem to exist. As a benefit, Curve25519 is considered to offer higher performance than other curves of the same security level (e.g. [76]). Of course, switching the elliptic curve used, requires the same modification to all parts of the Provotum system. Dropping mbed-os and mbedtls also requires manual integration of the MCU's hardware random number generator peripheral into libsodium.

Chapter 8

Summary and Conclusions

The goal of this thesis was to explore the use of a USB dongle in Provotum, given a baseline version of Provotum-RF, and try to achieve security improvements. This goal was formulated broadly on purpose to allow for an exploratory work that uncovers the potential of the USB dongle in the Provotum system. This work started off with a review of related work. Initial review revealed that the best security improvement achievable by use of a USB dongle might be coercion resistance and that most works using hardware dongles in REV systems do so to increase protection against malware interference and attempt to achieve coercion resistance. It also revealed that most works claiming coercion resistance were later shown to have weaknesses or to not have achieved full CR. Additionally, work related to to hardware dongles frequently proposes systems whose usability is poorer than systems without hardware dongles.

Next, a comparison of external hardware for use in this project was conducted. Adverse factors that spoke against some of the most secure options included lack of open-source and legal restrictions through NDAs. Hence, a simple device, the Solokey, was chosen for a Proof-of-Concept (POC). The idea of the POC is to provide a baseline implementation of a hardware-enabled version of Provotum to demonstrate feasibility, explore the development approach, the available tooling, discover pitfalls and to define an interface and function set that can be based upon in future work, both from an academic and development perspective. From an academic perspective, the formulated thoughts and concepts can serve as a map of already explored areas and areas with most promising exploration left open. From a development perspective, the integration tests and Provotum system integration can serve as regression tests while porting the dongle functionality to a more secure device or optimizing the firmware on the same device. Additionally, documentation of tooling around this work's development can serve to accelerate future work and documentation of pitfalls can serve to avoid similar problems in the future.

Definition of the design of the PoC involved getting a good enough understanding of the system's underlying cryptography and ZKP to allow formulation of own variants of the common Sigma Protocol proofs and ability to propose improvements to the randomization protocol and ZKP used in Provotum-RF. Additionally, algorithms for t-of-n DKG were evaluated during this phase, but it was decided against their use due to the high message overhead of the algorithms.

Concrete implementation followed a brief evaluation of platforms and programming languages, from which mbed-os and C++ were selected for dongle firmware development due to the ability to focus on development of functionality over writing a lot of boilerplate code to bring up the platform. In the course of development, all ElGamal encryptions and ZKP were modified from finite-field cryptography to ECC since it became quickly apparent that the runtime of the algorithms used by Provotum on the dongle is too high if finite-field cryptography is used.

A number of challenges presented themselves during implementation (see Section 8.1), but despite all these challenges and with the help of profiling and multiple analysis tools run on the firmware binary, a firmware binary was obtained that fits into the available flash memory (although just barely), has an acceptable runtime and implements all 6 dongle use cases discovered during the design phase of this project: ballot randomization, encrypting a vote on the dongle, creating a BC transaction on the dongle, voting on the dongle and obtaining an encrypted ballot, voting on the dongle and obtaining a BC transaction and, finally, sealer operation (key management and partial decryption of tallies). These dongle functions provide security advantages over Provotum-RF ranging from increased protection of private keys against theft and protection against intentional selling of private keys to protection against malware on the voter's computer. End-to-end integration was achieved by providing the necessary modifications to the Provotum cryptography libraries (both the one in Typescript and the one in Rust) and the Provotum Substrate chain codebase. An integration test proves that the BC accepts dongle-signed transactions and successfully verifies the contained ZKP.

A proposed feature that needs to be specially mentioned is panic passwords. During the study of related work for this thesis the concept of using panic passwords to achieve coercion resistance in REV systems was discovered. This work includes a proposal of how panic passwords can be integrated into the Provotum system that does not require modification to the high-level architecture (i.e. the system components involved and their interaction) and is compatible with the Provotum ballot format. Completion of the feature requires a way of proving to the identity provider the validity of a set of panic passwords in their blinded form. Future work should attempt to find a solution to this since that would achieve coercion resistance. The existing codebase of the dongle firmware and system integration should provide a formidable baseline for this future work.

8.1 Challenges Encountered During Development

The development part of this work covered a number of different systems, all of which come with their own tooling and require sligthly different development skillsets, some of which I already had and some of which I had to acquire.

I am frustrated, for example, that I didn't achieve proper integration of the dongle firmware with the Solokey bootloader (see Appendix B), but the development of the dongle firmware required acquiring an in-depth understanding of parts of the Solokey's hardware and firmware. This includes understanding of the memory model of ARM Cortex processors, the SWD debug interface, debugging practices for microcontrollers, the

boot procedure of ARM Cortex processors and how a bootloader can chainload an application, clock tree configuration of the concrete MCU used and usage of some peripherals of the MCU. Additionally an in-depth understanding of the gcc compiler and linker for embedded development was required including the use of linker files to control allocation of memory for compiled symbols and control of their exact location and reading the linker-generated map-file to verify symbol placement. Gcc-related tooling such as gprof and nm were used to debug problems with the binary size and firmware runtime.

On the Solokey side required knowledge included how the bootloader and application communicate, how the tooling (i.e. solo-python) interacts with the system and reading the hardware schematic to identify the way of electrical attachment of the button and RGB LED to the MCU.

Of course, the some of the used platforms and libraries such as mbed-os also required some effort to famliarize oneself with their architecture and APIs.

Fortunately, a deep understanding of the Substrate architecture was not required for the PBB modifications, but the Rust programming language required some effort to learn, although it was less than initially expected. Some required information around the Substrate ecosystem was only sparsely documented. Implementation of signatures over Substate transactions, for example, required reading and debugging the Polkadot source code to obtain the exact data that needs to be passed into the signature algorithm and the required configuration of the signature and hashing algorithm. An additional difficulty was the number of breaking changes that some dependencies, such as Substrate and its transitive dependencies, have made in the last year. One example is the Sr25519 (whose definition is exposed in form of its main implementation, the Schnorrkel library), which made breaking changes around version 0.3. Multiple phased-out repositories still hosted on the Paritytech Github link to Schnorrkel v0.1.0 while more recent repositories link to version 0.9.1. Identifying which repositories could safely be used against the Substrate version used in Provotum-RF involved hitting a few pitfalls. Additionally, just getting the Provotum Substrate chain binary to build involved a lot of effort because since its last release some maintainers of transitive dependencies broke the contract of semantic versioning, and a lock–file was not available in the Provotum chain codebase.

Concerning lock–files, figuring out how to handle problems with transitive dependencies and trying to use tooling such as `cargo tree` to visualize dependency trees was challenging in general since in the embedded projects I usually work on all dependencies and transitive dependencies are usually fixed and rarely ever changed.

Another painful experience in implementation was also to disover how much the decision to use C++ and parts of the STL impacted the binary size of the firmware (as discussed in Chapter 7).

Given the challenges faced I think that the codebase developed in the course of this work is a sizeable achievement. The dongle firmware provides a solid foundation for future work for the reasons previously discussed and the lessons I learned from this project will prove highly valuable in future embedded development projects of mine.

8.2 Summary

In summary, this work explores the use of a USB dongle in the Provotum system and presents a PoC system – Provotum-HW – that implements 6 dongle functions that improve security over Provotum-RF. The codebase and outwards interface of the dongle firmware serve as a baseline for future implementation and potentially achieving coercion resistance with panic passwords. The integration tests and Provotum system integration allow testing for regressions while porting the dongle functionality to a more secure device or optimizing the firmware on the same device. Additionally, the documentation accumulated in this work can serve as a basis to accelerate the start of future projects based on this work and help avoid pitfalls during development.

8.3 Future Work

Provotum-HW identifies and implements options to improve security of Provotum by means of an additional hardware dongle, but at the same time discovers new open challenges for future work.

8.3.1 Dongle with Display

If an external (USB other interface) device with a display is employed, it is possible to offer better usability for the vote-on-the-dongle use case. Hence, protection against malware can be combined with almost equal usability to voting on the voter's PC. Additionally, the system can support more complex votes (beyond yes– and no–votes) entered directly into the dongle with equal protection against malware. Alternatively, an the use of a smartphone application communicating with the Solokey via its NFC interface could be explored.

8.3.2 Integration of Panic Passwords

As outlined in Section 5.2.3, panic passwords still require solutions to some problems during identity provisioning. Ideally, no modifictaion to the dongle firmware is required and integration can proceed according to the integration steps detailed in Table 6.4.

8.3.3 Non Malleable Encoding of Transcripts and Hashed Data

Currently, whenever data is hashed, it is just concatenated and fed into the hashing algorithm. A constructed case where this could be a problem is when, given the bitstrings `A=01, B=00, C=010, D=0`, the hashes $H(A||B)$ and $H(C||D)$ are equal (where $||$ denotes concatenation). While concrete attacks on this might not exist in the context of Provotum,

best practices usually require hashed data to be serialized in some canonical form that uniquely determines the elements added to the hashed data, their order and their length. The Strobe protocol framework [25] or Merlin transcripts [24] could be used for that since they are already used for the same purpose in the Sr25519 signature algorithm used by the Substrate BC.

8.3.4 Certificate / Trust Chain Structure for Dongle Keys

Currently, all dongles sign the artifacts they generate with the same, hardcoded key. Ideally, each dongle has a different key and a certificate chain is in place to allow verification of a dongle key back to a single root of trust. Additionally, revocation of dongle keys should be checked so that the manufacturer can revoke the certificates of dongles with discovered security vulnerabilities. Equipping each dongle with a different private key requires definition of a process by which the key is injected into the dongle during manufacturing or generated at the first boot and by which a certificate can be issued for the associated public key.

8.3.5 Securing Deployment of Firmware

Currently, the Solokeys are equiped with the "hacker" bootloader, which allows flashing of unsigned firmware images. By flashing the non-hacker variant of the Solokey bootloader and installing a public key on the dongle, only firmware images can be flashed via the bootloader that are signed with the corresponding private key. Enabling the flash memory protection and debug interface lockout, external access to the dongle's memory and debug capabilities is prevented unless the dongle's memory is fully erased.

8.3.6 Firmware Performance Optimization

Section 7.5 lists a number of potential optimizations by which the performance of dongle functionality could be improved, wait times for voters reduced during ballot preparation and additional flash memory freed for implementation of other future features in the dongle firmware.

Bibliography

[1] S. G. Weber, R. Araujo, and J. Buchmann, "On coercion-resistant electronic elections with linear work," in *The Second International Conference on Availability, Reliability and Security (ARES'07)*, pp. 908–916, IEEE, 2007.

[2] D. Chaum, M. Jakobsson, R. L. Rivest, P. Y. Ryan, J. Benaloh, M. Kutylowski, and B. Adida, *Towards trustworthy elections: new directions in electronic voting*, vol. 6000. Springer, 2010.

[3] R. Araújo, N. B. Rajeb, R. Robbana, J. Traoré, and S. Youssfi, "Towards practical and secure coercion-resistant electronic elections," in *International Conference on Cryptology and Network Security*, pp. 278–297, Springer, 2010.

[4] S. Delaune, S. Kremer, and M. Ryan, "Coercion-resistance and receipt-freeness in electronic voting," in *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pp. 12–pp, IEEE, 2006.

[5] A. Juels, D. Catalano, and M. Jakobsson, "Coercion-resistant electronic elections," in *Towards Trustworthy Elections*, pp. 37–63, Springer, 2010.

[6] R. Matile and C. Killer, "Privacy, verifiability, and auditability in blockchain-based e-voting." University of Zurich, 2018.

[7] M. Eck, A. Scheitlin, and N. Zaugg, "Design and implementation of blockchain-based e-voting." University of Zurich, 2020.

[8] A. Hofmann, "Security analysis and improvements of a blockchain-based remote electronic voting system." University of Zurich, 2020.

[9] M. Eck, "Mixnets in a distributed ledger remote electronic voting system." University of Zurich, 2021.

[10] D. Wong, *Real-World Cryptography*, vol. MEAP Edition Version 11. Manning, 2021.

[11] D. Boneh and V. Shoup, "A graduate course in applied cryptography," 2020.

[12] S. Goldwasser and M. Bellare, "Lecture notes on cryptography," *Summer course "Cryptography and computer security" at MIT*, vol. 2008, 2008.

[13] N. P. Smart, "Cryptography made simple," 2016.

[14] R. Pass and A. Shelat, "A course in cryptography," 2010.

[15] B. Schoenmakers, "Lecture notes cryptographic protocols," 2021.

[16] IETF, "Rfc 4492." https://datatracker.ietf.org/doc/html/rfc4492. Accessed: 2021-07-03.

[17] "Ristretto website." https://ristretto.group/. Accessed: 2021-07-01.

[18] I. Chatzigiannakis, A. Pyrgelis, P. G. Spirakis, and Y. C. Stamatiou, "Elliptic curve based zero knowledge proofs and their applicability on resource constrained devices," in *2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*, pp. 715–720, IEEE, 2011.

[19] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.

[20] "Cryptography stackexchange: Mapping of message onto elliptic curve and reverse it?." https://crypto.stackexchange.com/questions/14955/mapping-of-message-onto-elliptic-curve-and-reverse-it/14966#14966. Accessed: 2021-07-01.

[21] "Cryptography stackexchange: Elliptic curve elgamal with homomorphic mapping." https://crypto.stackexchange.com/questions/32386/elliptic-curve-elgamal-with-homomorphic-mapping/32391#32391. Accessed: 2021-07-01.

[22] "Polkadot wiki: What is sr25519 and where did it come from?." https://wiki.polkadot.network/docs/learn-keys/#what-is-sr25519-and-where-did-it-come-from. Accessed: 2021-07-02.

[23] "Schnorrkel." https://github.com/w3f/schnorrkel. Accessed: 2021-07-02.

[24] "Merlin." https://docs.rs/merlin/1.0.3/merlin/. Accessed: 2021-07-02.

[25] "Strobe protocol framework." https://strobe.sourceforge.io/. Accessed: 2021-07-02.

[26] "Keccak." https://keccak.team/keccak.html. Accessed: 2021-07-02.

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

[28] T. Okamoto, "Provably secure and practical identification schemes and corresponding signature schemes," in *Annual international cryptology conference*, pp. 31–53, Springer, 1992.

[29] D. Pointcheval and J. Stern, "Security arguments for digital signatures and blind signatures," *Journal of cryptology*, vol. 13, no. 3, pp. 361–396, 2000.

[30] P. Horster, H. Petersen, and M. Michels, "Meta-elgamal signature schemes," in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pp. 96–107, 1994.

[31] O. Blazy, G. Fuchsbauer, D. Pointcheval, and D. Vergnaud, "Signatures on randomizable ciphertexts," in *International Workshop on Public Key Cryptography*, pp. 403–422, Springer, 2011.

[32] N. Asghar, "A survey on blind digital signatures," *Dept. Combinatorics Optim., Univ. Waterloo, ON, Canada, Tech. Rep*, 2011.

[33] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of cryptology*, vol. 4, no. 3, pp. 161–174, 1991.

[34] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *Annual international cryptology conference*, pp. 89–105, Springer, 1992.

[35] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Conference on the theory and application of cryptographic techniques*, pp. 186–194, Springer, 1986.

[36] D. Bernhard, O. Pereira, and B. Warinschi, "How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 626–643, Springer, 2012.

[37] M. Jakobsson, K. Sako, and R. Impagliazzo, "Designated verifier proofs and their applications," in *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 143–154, Springer, 1996.

[38] T. Okamoto and K. Ohta, "Divertible zero knowledge interactive proofs and commutative random self-reducibility," in *Workshop on the Theory and Application of of Cryptographic Techniques*, pp. 134–149, Springer, 1989.

[39] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[40] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pp. 427–438, IEEE, 1987.

[41] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Annual international cryptology conference*, pp. 129–140, Springer, 1991.

[42] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," in *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 295–310, Springer, 1999.

[43] A. Kate and I. Goldberg, "Distributed key generation for the internet," in *2009 29th IEEE International Conference on Distributed Computing Systems*, pp. 119–128, IEEE, 2009.

[44] "USB 2.0 Specification." `https://www.usb.org/document-library/usb-20-specification`. last visit May 14, 2021.

[45] D. L. Chaum, *Computer Systems established, maintained and trusted by mutually suspicious groups*. Electronics Research Laboratory, University of California, 1979.

[46] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," tech. rep., Manubot, 2019.

[47] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[48] "Substrate Blockchain Platform." `https://substrate.io`. last visit May 14, 2021.

[49] Ü. Madise and T. Martens, "E-voting in estonia 2005. the first practice of country-wide binding internet voting in the world," in *Electronic Voting 2006–2nd International Workshop, Co-organized by Council of Europe, ESF TED, IFIP WG 8.6 and E-Voting. CC*, Gesellschaft für Informatik eV, 2006.

[50] W. Lueks, I. Querejeta-Azurmendi, and C. Troncoso, "Voteagain: A scalable coercion-resistant voting system," *arXiv preprint arXiv:2005.11189*, 2020.

[51] O. Kulyk, V. Teague, and M. Volkamer, "Extending helios towards private eligibility verifiability," in *International Conference on E-Voting and Identity*, pp. 57–73, Springer, 2015.

[52] P. Locher and R. Haenni, "Receipt-free remote electronic elections with everlasting privacy," *Annals of Telecommunications*, vol. 71, no. 7, pp. 323–336, 2016.

[53] J. Clark and U. Hengartner, "Selections: Internet voting with over-the-shoulder coercion-resistance," in *International Conference on Financial Cryptography and Data Security*, pp. 47–61, Springer, 2011.

[54] C. Spadafora, R. Longo, and M. Sala, "A coercion-resistant blockchain-based e-voting protocol with receipts," *Advances in Mathematics of Communications*, 2021.

[55] M. Backes, M. Gagné, and M. Skoruppa, "Using mobile device communication to strengthen e-voting protocols," in *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pp. 237–242, 2013.

[56] R. Wen and R. Buckland, "Masked ballot voting for receipt-free online elections," in *International Conference on E-Voting and Identity*, pp. 18–36, Springer, 2009.

[57] M. Hirt and K. Sako, "Efficient receipt-free voting based on homomorphic encryption," in *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 539–556, Springer, 2000.

[58] P. Chaidos, V. Cortier, G. Fuchsbauer, and D. Galindo, "Beleniosrf: A non-interactive receipt-free electronic voting scheme," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1614–1625, 2016.

[59] A. Juels, D. Catalano, and M. Jakobsson, "Coercion-resistant electronic elections," in *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, WPES '05, (New York, NY, USA), p. 61–70, Association for Computing Machinery, 2005.

[60] W. D. Smith, "New cryptographic election protocol with best-known theoretical properties," in *Proc. of Workshop on Frontiers in Electronic Elections*, pp. 1–14, 2005.

[61] R. Araújo, S. Foulle, and J. Traoré, "A practical and secure coercion-resistant scheme for remote elections," in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.

[62] M. R. Clarkson, S. Chong, and A. C. Myers, "Civitas: Toward a secure voting system," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pp. 354–368, IEEE, 2008.

[63] B. Meng, Z. Li, and J. Qin, "A receipt-free coercion-resistant remote internet voting protocol without physical assumptions through deniable encryption and trapdoor commitment scheme.," *JSW*, vol. 5, no. 9, pp. 942–949, 2010.

[64] T. Haines and J. Müller, "How not to voteagain: Pitfalls of scalable coercion-resistant e-voting,"

[65] D. Chaum, "Surevote: technical overview," in *Proceedings of the workshop on trustworthy elections (WOTE'01)*, 2001.

[66] G. S. Grewal, M. D. Ryan, L. Chen, and M. R. Clarkson, "Du-vote: Remote electronic voting with untrusted computers," in *2015 IEEE 28th Computer Security Foundations Symposium*, pp. 155–169, IEEE, 2015.

[67] S. Kremer and P. B. Rønne, "To du or not to du: A security analysis of du-vote," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 473–486, IEEE, 2016.

[68] B. Lee and K. Kim, "Receipt-free electronic voting scheme with a tamper-resistant randomizer," in *International Conference on Information Security and Cryptology*, pp. 389–406, Springer, 2002.

[69] R. Haenni and R. E. Koenig, "Voting over the internet on an insecure platform," in *Design, Development, and Use of Secure Electronic Voting Systems*, pp. 62–75, IGI Global, 2014.

[70] S. Neumann and M. Volkamer, "Civitas and the real world: problems and solutions from a practical point of view," in *2012 Seventh International Conference on Availability, Reliability and Security*, pp. 180–185, IEEE, 2012.

[71] J. Clark and U. Hengartner, "Panic passwords: Authenticating under duress.," *HotSec*, vol. 8, p. 8, 2008.

[72] "FIDO Alliance." `https://fidoalliance.org`. last visit May 14, 2021.

[73] "OpenPGP." `https://www.openpgp.org`. last visit May 14, 2021.

[74] "mbed OS." `https://os.mbed.com/mbed-os`. last visit May 14, 2021.

[75] "Keccak." `https://cbor.io/`. Accessed: 2021-07-02.

[76] P. Sasdrich and T. Güneysu, "Efficient elliptic-curve cryptography using curve25519 on reconfigurable devices," in *International Symposium on Applied Reconfigurable Computing*, pp. 25–36, Springer, 2014.

Abbreviations

3V	Three Volt
ARM	Name of a CPU architecture
BC	Blockchain
CDC	Communication Device Class
CPU	Central Processing Unit
CR	Coercion Resistance
CSPRNG	Cryptographically Secure Pseudorandom Number Generator
DFU	Device Firmware Update
DKG	Distributed Key Generation
DV Proof	Designated Verifier Proof
ECCDH	Elliptic Curve Computational Diffie Hellman
ECC	Elliptic Curve Cryptography
ECDDH	Elliptic Curve Decisional Diffie Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
FIDO2	A standard for 2-Factor Authentication
GND	Ground
HSM	Hardware Security Module
HW	Hardware
KDF	Key Derivation Function
LED	Light-Emitting Diode
MCU	Microcontroller Unit
MITM	Man-in-the-Middle
NDA	Non-Disclosure Agreement
NFC	Near Field Communication
NIZKP	Non-interactive Zero Knowledge Proof
PBB	Public Bulletin Board
PIN	Personal Identification Number
PRNG	Pseudorandom Number Generator
RAM	Random Access Memory
REV	Remote Electronic Voting
RF	Receipt-Freeness
RNG	Random Number Generator
RTOS	Real-Time Operating System
SIM	Subscriber Identification Module
STL	Standard Template Library
SWCLK	The clock signal of SWD

SWDIO	The I/O signal of SWD
SWD	Serial Wire Debug
U2F	Universal 2-Factor Authentication
USB	Universal Serial Bus
VPN	Virtual Private Network
ZKP	Zero Knowledge Proof

List of Figures

List of Tables

Appendix A

Installation Instructions

This appendix details the required steps to build and run the software components produced or modified in this work.

A.1 Dongle Firmware

You need Python on your system. You also need an ARM cross-compiler, i.e. a compiler produces a binary for ARM Cortex-M processors even though the compiler runs, for example, on an x64 architecture. You can get arm-none-eabi-gcc at `https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads`.

Clone the git repository recursively

```
1 git clone --recurse-submodules git@github.com:provotum/dongle_fw.git
```

Then you need to install the mbed command line tools. I recommend installing these in a Python virtual environment to avoid cluttering up the global modules:

```
1 python3 -m venv env
2 source env/bin/activate
```

Then

```
1 python3 -m pip install mbed-cli
2 cd dongle_fw
3 python3 -m pip install -r mbed-os/requirements.txt
```

Set the GCC_ARM_PATH or MBED_GCC_ARM_PATH according to Mbed documentation as an environment variable or within the mbed toolchain config to the path of your ARM cross-compiler installation.

Compile the firmware with

```
1 mbed compile -t GCC_ARM -m SOLOKEY --profile develop
```

The firmware is then located in BUILD/SOLOKEY/GCC_ARM-DEVELOP/mbed.bin (or mbed.elf) for loading into a debug application. A hex file can be obtained by running

```
arm-none-eabi-objcopy -O ihex /path/to/mbed.elf /target/path/mbed.hex
```

Flashing the firmware can currently only be achieved over the debug interface since the dongle firmware cannot be properly flashed nor booted by the Solokey bootloader (see Appendix B).

A.1.1 Flashing via the Debug Interface

Solder wires to the Solokey test points on the bottom labeled with GND1, 3V1, SWDIO1 and SWCLK1. Connect the wires to debug probe according to the debug probe's manual. The pin assignment for Segger debug probes is documented at `https://www.segger.com/products/debug-probes/j-link/technology/interface-description/`. Figure A.1 shows the connection to a Segger debug probe. Use a debug application compatible with the given debug probe. For Segger probes, Segger Ozone can be and has been used in this work. Note that the dongle still needs to be plugged into the USB port for power and communication with the application client.

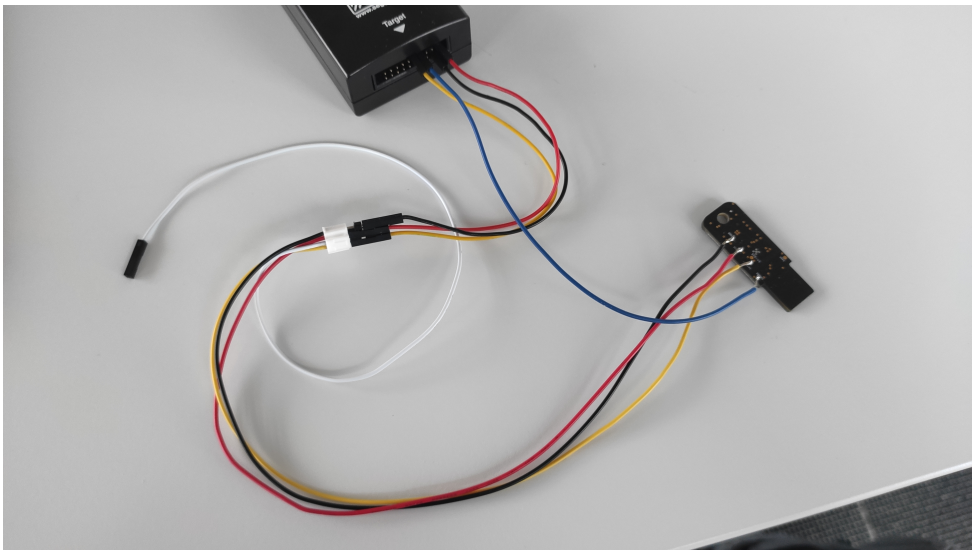

Figure A.1: Solokey Debug Wire Attachment

A.2 Native Build of Dongle Functions

You need cmake, make or ninja and gcc/g++.

Clone the git repository recursively

```
git clone --recurse-submodules git@github.com:provotum/dongle_native_tests.git
```

Then build with cmake

```
cd dongle_native_tests
mkdir build
cd build
cmake ..
make
```

Then run the test executable

```
./test
```

A.3 Client Dongle Tests

You need node (tested with version 14), npm, typescript.

Clone the git repository

```
git clone --recurse-submodules git@github.com:provotum/pv-hw-client.git
cd pv-hw-client
```

Install dependencies

```
npm i
```

Make sure the dongle is plugged in and identify onto which serial port device the CDC interface is mapped. Set an environment variable with the port path, for example

```
export PROVOTUM_SERIAL_PORT=/dev/ttyACM0
```

Depending on your setup, you also have to ensure proper access rights of your user to the serial device. Then compile and run the tests

```
tsc -b
npm test
```

A.4 Provotum Chain Binary

Install the Rust compiler with a compatible nightly build. The following version worked for me (installation instructions assume Ubuntu or Debian):

```
sudo apt-get install clang libclang-dev
curl https://sh.rustup.rs -sSf | sh -s -- -y
source $HOME/.cargo/env
rustup install nightly-2021-02-11
rustup default nightly-2021-02-11
rustup uninstall stable
```

Clone the git repository

```
git clone --recurse-submodules git@github.com:provotum/ProvotumChain.git
cd pv-hw-client
```

To run the unit tests, run

```
cargo test -- --nocapture
```

To build the chain binary, run

```
cargo build
```

Appendix B

Fixing Flashing of the Firmware

Flashing of the dongle firmware is currently not possible using the Solokey bootloader. Since this affects the possibility to flash and test the dongle firmware by oneself, this section documents the attempted solutions and proposes future fixes to resolve the issue.

Care was taken that the application does not occupy flash regions occupied by the bootloader. But the application is too big to fit completely into the application flash section of the Solokey flash layout and the bootloader refuses to write those parts of the application that are mapped to the application data flash section (refer to Section 6.1.1 for an overview over the flash layout).

Flashing was attempted using a modified bootloader that does not perform range checks on the written flash pages, but the bootloader crashed during the writing of the new firmware binary indicating some other yet unidentified problem is present. Debugging this was not yet attempted but requires a debug probe attached to the Solokey.

Another approach is to use DFU mode to flash a firmware. The Solokey documentation describes how to enter DFU mode on a fresh Solokey and use it to flash a firmware binary. DFU erases the full memory before flashing, thus requiring flashing of the bootloader and application at the same time. The Python tool `intelhex` can, for example, be used to merge an application and bootloader hex file. However, the Provotum dongle firmware has no implementation to enter DFU mode after Provotum firmware is flashed, preventing repeated reflashing with other firmware. Further, an attempt of doing this did not yield a properly booting configuration.

One simple fix is to simply slim down the Provotum firmware first until it fits completely into the application section of the Solokey flash layout and then attempt flashing using the unmodified Solokey bootloader. However, a debug probe is still required for flashing to allow testing during the slimming–down efforts.

Alternatively, a different bootloader – such as `mcuboot` – could be employed. Divisioning of flash memory could then be arbitrarily chosen and a combined binary of the new application and bootloader flashed once using DFU mode. Subsequent firmware updates could then be achieved via the new bootloader. But the learning curve of the new bootloader might equally steep as for the Solokey bootloader.

If no firmware updates are to be supported the appliation can simply be compiled to take up the entire flash memory. Flashing it once to a fresh Solokey could be achieved via DFU mode. The application might even implement a function allowing reboot into DFU mode for subsequent reflashing.

Another alternative is to simply port the functionality away from the Solokey onto a HSM–enabled device or the STM NUCLEO-L432KC Dev-Kit, equipped with the same processor but with an onboard debugger. Since this new device does not ship with the Solokey bootloader anway, no attempt at integration has to be made and the full flash memory of the device is available for application code. Reflashing is fairly simple for everyone due to the onboard debugger (STM tooling will have to be installed for reflashing).

Appendix C

Contents of the CD

The contents of the CD are all aggregated in a single ZIP archive. Inside the archive the following directory tree exists:

```
/
├── thesis.pdf ..... The thesis in PDF format
├── midterm.pdf ..... The midterm presentation slides in PDF format
├── dongle_fw ..... The dongle firmware source code
│   ├── mbed-os ..... A copy of the mbed-os RTOS
│   └── external ..... Copies of external libraries depended upon
├── dongle_native_tests ..... The native source code of dongle functions and tests
│   └── external ..... Copies of external libraries depended upon
├── evote-crypto-ts ..... The crypto library used by Typescript code
├── evote-crypto-rs ..... The crypto library used by Rust code
├── pv-hw-client ..... The Node client interacting with the dongle and BC
└── ProvotumChain ..... The Provotum Substrate chain binary source code
```

All codebases are also pushed to Provotum's Github.