A formal security analysis of Blockchain voting

NIKOLAJ SIDORENCO, Aarhus University, Denmark
BAS SPITTERS, Aarhus University, Denmark

Voting and blockchains are intimately connected. Voting is used in blockchains for consensus, governance, and decentralized organizations (DAOs). Conversely, blockchains can be used to serve as a bulletin board for smaller elections. For all these applications the stakes are high, both financial and societal. Moreover, for blockchains, the adversarial model is complex: the adversary has complete knowledge of the system and full access to the network.

Here we focus on the use of blockchains for smaller elections; so-called boardroom voting. We consider one such protocol: the Open Vote Network (OVN), which provides private voting on a blockchain. Such private voting could also be applied to a DAO, improving the state of the art where voting for DAOs is often pseudonymous instead of private.

The OVN protocol has previously been implemented as a smart contract on Ethereum. It comes with an informal security argument. We propose a more complete analysis, which we, moreover, formalize in the Coq proof assistant. We aim to connect our formalization of this protocol with a smart contract in the ConCert framework in the Coq proof assistant. This contract comes with a formal proof that the tally is computed correctly.

1 INTRODUCTION

Digitization is facilitating many aspects of our society, and voting is no exception. However, in voting the stakes are higher than in many other applications. Moreover, elections require trust for their outcome to be accepted. Blockchains provide a trusted bulletin board, and as such, they have been used as part of voting protocols. Conversely, online voting also serves an important role in the blockchain ecosystem itself. Online voting is used in some modern blockchains for making fundamental changes to the system — so-called blockchain governance; see [18] for an overview.

Online voting systems have had buggy implementations or questionable security guarantees. Blockchains can solve some of these issues but also raise a number of new questions; see [25] for a description of the many issues with blockchain voting. Two critiques of blockchain-based solutions are their security and the quality of their implementation. We will provide a road to addressing these issues by focusing on a particular blockchain voting protocol: the Open Vote Network (OVN) [17].

The OVN allows a small group (of N participants) to vote anonymously. To perform the vote, all participants must first agree on a public encryption key. The encryption key in the OVN

Authors' addresses: Nikolaj Sidorenco, Aarhus University, Dept. of Computer Science, Denmark, sidorenco@cs.au.dk; Bas Spitters, Aarhus University, Dept. of Computer Science, Denmark, spitters@cs.au.dk.

has a unique property: when multiplying N values encrypted with the key, one can efficiently uncover the unencrypted product. Moreover, recovering an unencrypted product of M values is only feasible if $N = M$. The OVN has been implemented as a smart-contract on the Ethereum blockchain [20]. Such a smart contract implementation raises two questions: 1) Is the protocol implemented correctly as a smart contract?, and 2) Is the protocol cryptographically secure?

A provably correct implementation is already available in ConCert [3]. ConCert is a framework for implementing smart contracts as programs in the Coq proof assistant [29]. It allows one to transform ('extract') these programs into several smart contract languages. ConCert was used to prove functional properties of an OVN implementation. Tools like ConCert, however, cannot reason about cryptographic security.

We will consider how to prove cryptographic security. We will do this in the computational model, which is more precise than the symbolic model. In this model, one often uses a 'game hopping' style to reduce the security of a protocol to well-established cryptographic assumptions. To do this we need a programming logic to deal with probabilities. A number of tools support this [1, 5, 30]. We use SSProve [1], which provides tool support for a modular game-hopping variant, called State Separating Proofs [6].

The formal security of voting protocols has been analyzed before. Notably, the Helios protocol has been analyzed in the symbolic model [4]. Moreover, the ballot secrecy property of Helios has been formalized in the computational model in EasyCrypt [9]. The procedure verifying the ballots has also been implemented in Coq and proven secure in the symbolic model [15]. This restricted part of the Helios protocol was then extracted to executable code.

The procedure for verifying a ballot in Helios is a Σ-protocol similar to one of the phases of the Open Vote Network. In the present work, we provide a formally verified and provably cryptographically secure implementation with executable code for all procedures of the voting protocol, in addition to the verifying procedure.

With this paper, we address some of the security concerns about blockchain voting [25] by providing a means to formally verify the security properties of the OVN. Functional correctness of the OVN has already been proved in the ConCert framework [2].

1.1 Our Contributions

In this work, we investigate the security of the Open Vote Network. To be concrete:

- We highlight a gap in the original proof.
- We formally specify the requirements for honest voter privacy.
- We provide the first machine-checked security proof of the Open Vote Network.

1.2 Outline

Section 2 presents the necessary background for this work. Section 3 defines our model of a public ledger and our formalization of the OVN. Section 4 compares our work with previous work on smart contract extraction. Our code is available from https://github.com/SSProve/ssprove.

2 PRELIMINARIES

2.1 SSProve

State-separating proofs (SSP) is a methodology used for structuring game-based cryptographic proofs. A similar, but more informal, methodology is used in the Joy of Cryptography textbook [26]. The methodology allows game-based definitions and proofs to be composed in a modular way. Until recently, the SSP methodology lacked the necessary tool support to be applicable in the formal verification of cryptography. This lack of tooling was recently addressed by SSProve [1] which formalizes the methodology in Coq. SSProve defines a (probabilistic) relational Hoare logic for proving equivalences between cryptographic games. In addition to the relational logic, SSProve defines a set of laws about the high-level composition of protocols. With these laws, it is possible to define cryptographic protocols and games which can easily be composed in a formal setting.

SSProve defines protocol composition through its notion of packages. A package is a set of imports, exports, and memory locations. Imports are signatures of procedures (expected input types, output type, and a unique identifier). The package must contain implementations of all procedures specified in the set of exports. A package where the set of imports is empty is called a *game*.

Packages can be composed in two ways: sequentially (denoted $P_1 \circ P_2$) or in parallel (denoted $P_1 \parallel P_2$). Sequentially composing $P_1 \circ P_2$ requires that the exports of P_2 are a subset of the imports of P_1. The sequential composition $P_1 \circ P_2$ is itself a package where the procedures of P_2 are inlined into P_1. Any imports of P_2 are added to the imports of the package. Parallel composition $P_1 \parallel P_2$ constructs a new package combining the exports and imports of the two packages.

Packages are implemented in SSProve using an imperative language which includes sampling from distributions, accessing memory locations, and calling procedures. Table 1 shows a package named **SAMPLE** exporting one procedure *Main* with no imports. *Main* samples uniformly from the set $\{0, 1\}$, stores it in memory location *loc*, and returns it.

Security is defined as playing against an adversary trying to distinguish two games:

Definition 2.1 (Advantage). *For an adversary $\mathcal{A}$ and two games, G_1 and G_2, the advantage is given by:*

$$\alpha(G_1, G_2)(\mathcal{A}) = |\Pr[true \leftarrow \mathcal{A} \circ G_1] - [true \leftarrow \mathcal{A} \circ G_2]|$$

```
package: SAMPLE
mem:  loc: secret
```

```
Main():
  x <$ uniform {0,1}
  put loc = x
  y ← get loc
  ret y
```

Table 1. Example package

To reason about the distinguishing advantage between two packages, SSProve provides the following two lemmas:

Lemma 2.2 (Triangle inequality). *Let G_1, G_2, and G_3 be games. The distinguishing advantage is given by:*

$$\alpha(G_1, G_3)(\mathcal{A}) \leq \alpha(G_1, G_2)(\mathcal{A}) + \alpha(G_2, G_3)(\mathcal{A})$$

Lemma 2.3 (Reduction lemma). *Let G_1 and G_2, be games and let M be an arbitrary package. The distinguishing advantage is then given by:*

$$\alpha(M \circ G_1, M \circ G_2)(\mathcal{A}) \leq \alpha(G_1, G_2)(\mathcal{A} \circ M)$$

By combining the SSP methodology with SSProve, it is possible to give graphical representations of proofs that have a clear semantic in a formal logic.

2.2 ConCert

ConCert is a smart contract certification framework in the Coq proof assistant. It models a blockchain abstractly, as an immutable append-only ledger. It models account-based blockchains such as Ethereum, Tezos, and Concordium.

Smart contracts are modeled as functions in Coq's gallina functional language. The blockchain model is executable in Coq, thus facilitating property-based testing, in addition to formal proofs. The smart contracts in gallina can be transpiled in a verified way to a number of industrial smart contract languages, including LIGO and rust.

A number of paradigmatic smart contracts have been verified in ConCert. This includes Escrow, crowdfunding, token standards, a DAO, a decentralized exchange [23], and an interpreter for a DSL for finance. The modeling is fairly precise, but it does not deal with resource (gas) issues. Fortunately, the problems around the infamous DAO have already been modeled in ConCert [24].

2.3 The Open Vote Network

The Open Vote Network (OVN) is an online voting protocol based on zero-knowledge and an online ledger [17]. The OVN is a decentralized voting protocol without trusted parties. The protocol achieves this by applying zero-knowledge proofs to detect any deviation from the protocol. These zero-knowledge proofs are publicly available in a decentralized manner through the use of a blockchain. The verifiability of the votes depends on brute-forcing the inverse of a certain function, this limits the number of participants for which the OVN is feasible. Moreover, the OVN does not try to address issues with national elections, such as voter coercion, so it should only be used for small-scale elections.

We now give an account of the originally proposed protocol.

2.3.1 The OVN Protocol. The protocol proceeds in two rounds followed by a tallying process. First, one fixes a cyclic group G, with generator g, for which the Diffie-Hellman problem is hard. Each of the (n) participants P_i selects a random secret key $x_i \in Z_{|G|}$ and fixes a vote $v_i \in \{0, 1\}$.

Round 1: Every participant computes their public key $y_i = g^{x_i}$ and a zero-knowledge proof of the relation between y_i and x_i. All public keys are put on the public ledger and verified by the participants. After all these verifications, each participant computes a reconstruction key:

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

Round 2: Every participant computes their ballot as $g^{c_i x_i} g^{v_i}$ and a zero-knowledge proof that $v_i \in \{0, 1\}$ is a correct vote. Both the ballot and the zero-knowledge proof are published to the ledger.

Tallying: After each ballot is on the ledger and every zero-knowledge proof is checked to be valid, the votes can be tallied. To tally the votes we compute the product of all votes.

$$\prod_{i}^{n} g^{c_i x_i} g^{v_i} = g^{\sum_i^n v_i}$$

The product of all votes hides the tally in the exponent. To recover the vote the exponent must be brute-forced. This requires making a table of all multiples of the generator. The size of the table is limited by the number of participants.

Tallying the votes can be done by anyone. Every vote is published to the ledger, which is publicly available. The tallying process requires no information private to the participants of the vote. Hence, the only barrier to computing the tally is the computational requirement of brute-forcing the exponent.

2.3.2 Security Proofs. Hao et al. give the following security requirements for the OVN:

Definition 2.4 (Maximum ballot secrecy). *Each ballot is a ciphertext indistinguishable from random. Thus, revealing nothing about the vote it contains.*

Definition 2.5 (Self-tallying). *After all ballots have been cast, anyone can compute the result without external help.*

Definition 2.6 (Dispute-freeness). *The result should be publicly verifiable. Anyone can check whether all parties adhered to the protocol when casting their ballot.*

For this work, we focus on the maximum ballot secrecy property. The self-tallying property has already been proved [2]. Dispute-freeness is given by the use of zero-knowledge proofs in the protocol.

For maximum ballot secrecy, Hao et al. give a proof for the OVN [17, Theorem 1]. This proof has some interesting subtleties which are uncovered in a formal treatment as we will discuss in the following sections.

3 MODELING THE OVN

To give a formal treatment of the ballot secrecy of the OVN, we need a model of the execution environment.

Our formalization of the OVN consists of three components: First, we have the functionality that any (honest) party should perform during the protocol. Second, we assume access to an ideal functionality for zero-knowledge proofs. Third, the adversarial behavior is encoded inside the execution environment.

All of these components are given as SSP packages and implemented inside SSProve.

We start by giving an account of the different components:

Zero-knowledge. Zero-knowledge protocols [12] are a cryptographic primitive that allows a prover P to convince a verifier V of the veracity of a public statement, without revealing anything beyond that fact. In the OVN protocol by Hao et al. zero-knowledge is implemented as a Σ-protocol, which is made non-interactive by the Fiat-Shamir transformation [11]. We follow the presentation of Hao et al. and define our zero-knowledge protocols based on the Fiat-Shamir transformation. The zero-knowledge functionality is given by the SSProve package in Table 2.

To fully implement the **ZK** package two things are needed. First, the Fiat-Shamir transformation depends on a random oracle. Inside the **ZK** package, the random oracle is exposed through the Query interface. Second, we must supply implementations of Commit, Verify, and Response.

package: $\mathbf{ZK}_{\mathbf{real}}$
mem:

```
ZKVerify(h, a, e, z):
  e ← Query h a
  ret (Verify h a e z)

RUN(h, w):
  assert R h w
  a ← Commit h w
  e ← Query h a
  z ← Response h w a e
  ret (h,a,e,z)
```

Table 2. Real Zero-Knowledge Package

package: $\mathbf{ZK}_{\mathbf{ideal}}$
mem:

```
ZKVerify(h, a, e, z):
  e ← Query h a
  ret (Verify h a e z)

RUN(h, w):
  assert R h w
  (a, e, z) ← Sim h
  ret (h,a,e,z)
```

Table 3. ideal Zero-Knowledge Package

Security of the zero-knowledge functionality is defined as the ability to simulate the output of Run without access to the witness. We define this in SSProve as the advantage of an adversary to distinguish the real protocol from the simulated protocol:

Definition 3.1. *For any adversary $\mathcal{A}$ and random oracle O, the zero-knowledge advantage is given by:*

$$\alpha(ZK_{real} \circ O, ZK_{ideal} \circ O)(\mathcal{A})$$

With ZK_{ideal} as defined in Table 3.

Here, we utilize an idealized version of the two zero-knowledge protocols used by the OVN. These idealized versions are unimplemented **ZK** packages for which we assume that Definition 3.1 holds.

```
package: P_i^b
mem:  sk_loc_i: secret
      ck_loc_i: public
      vote: bool
```

```
Setup():
  sk <$ uniform  S
  put sk_loc_i := sk
  zkp ← SCHNORR.RUN ( g^x , x)
  ret ( g^x , zkp)

Construct(l : ledger):
  let key := (compute_key l i)
    in
  put ck_loc_i := key
  ret ()

Vote():
  x ← get sk_loc_i
  c ← get ck_loc_i
  if b then
    let vote := (c^x · g^v) in
    zkp ← CDS.RUN  g^x, c^x, v
    ret vote
  else
    let vote := (c^x · g^¬v) in
    zkp ← CDS.RUN  g^x, c^x, ¬v
    ret (vote, zkp)
```

Table 4. Honest Party Package.

The first protocol is the Schnorr zero-knowledge protocol [27] proving knowledge of (x, y) such that $y = g^x$. The Schnorr protocol is given by the **SCHNORR** package which is the **ZK** package with specific implementations of Commit, Response, and Verify. The second protocol is the CDS protocol [10] proving knowledge of (x, y, v) such that $v = (g^x, g^{xy} \cdot g) \vee v = (g^x, g^{xy})$. The CDS protocol is given by the **CDS** package which, again, is the **ZK** with specific implementations.

Honest Party Functionality. The core of the formalization consists of the single-party functionality. An honest party has three procedures encapsulated in a package, as seen in Table 4.

Each instantiation has its own private memory location to store its secret key. This ensures that composing multiple parties together in a protocol cannot compromise the secrets of any party.

The `compute_key` specified in the honest party package is the function computing the reconstruction key. A ledger is a finite map from natural numbers (representing the ID of the voter) to tuples of (public key, zero-knowledge proof). Compute_key ensures that all zero-knowledge proofs are valid and then computes:

$$\prod_{j=0}^{i} l[j] / \prod_{j=i}^{n} l[j]$$

where $l[j]$ represents the public key of party j on the ledger.

The Scheduler. The honest party functionality on its own cannot offer any security guarantees. To reason about the OVN, we have to reason about the public ledger and the times at which the honest party is called.

The **Schedule** package is used to model both the public ledger and to decide when each party of the protocol gets to run. In particular, the **Schedule** package is used to limit and reason about the adversary's power attacking the secrecy of any individual vote.

The adversary is given the capability to control all other parties (except one) of the protocol. This is modeled by letting the adversary pass a ledger state to the scheduler on which the remainder of the protocol will execute. The adversary is limited in three ways; first, the scheduler forces at least one other party in the protocol to submit an honestly constructed public key to the ledger. This honest public key is always given to party j which is known to the adversary. Second, the attacked party is run last. This party can therefore see the public keys of all other parties before submitting their public key. Last, the procedures of the attacked party are run in the correct order.

The limitation that the attacked party can see the other public keys might seem restrictive. Indeed, if all honest parties who want their ballot to be secret had to wait for the public key of all other parties to be available, the protocol would deadlock. Fortunately, the restrictive modeling is only an artifact of the modularity of the proof. As we will discuss later, this limitation is due to the restrictions in the rules about programs producing random samplings in probabilistic programming logics.

Alternatively, the modeling could be changed such that all parties are run inside the **Schedule** package. With this, **Schedule** contains all necessary information to reason about the random samplings and the order in which the parties are run. We chose the current representation to achieve a more modular proof where one only needs to reason about the execution of a single party.

```
package: Scheduler
mem:
```

```
Run(l : ledger):
  pk ← Setup
  x <$ uniform  S
  let y := g^x
  zkp ← SCHNORR.RUN (y, x)
  let l[j] := (y, zkp)
  let l[i] := pk
  Construct l
  ballot ← Vote
  ret ballot
```

Table 5. Adversarial Scheduler Package.

3.1 Security Proof

From Definition 2.4, we need to show that the vote produced by an honest party, subject to the scheduling, is indistinguishable from random. The game in Table 4 states a slightly different property. The game given by comparing P_i^0 to P_i^1 proves that no adversary can determine which of the two choices was voted for. If one were to prove that the vote was indistinguishable from random, P_i^0 would have to construct a zero-knowledge proof that the ballot was constructed from a vote $v \in \{0, 1\}$, which might not be the case.

Expressing our refined game within SSProve, we want to bound the following advantage:

$$\alpha(\textbf{Scheduler} \circ P_i^1 \circ ZK_{real}, \textbf{Scheduler} \circ P_i^0 \circ ZK_{real})$$

for any adversary $\mathcal{A}$.

By inlining the procedures from the composed packages, we obtain the code in Figure 1. The notation $l[i] := x$ denotes updating the ledger (a finite map) on key i to value x. For simplicity of notation, we assume that the ledger is mutable and the notation $l[i] := x$ mutates the variable l. Of course, a ledger is not mutable, we come back to this issue in Section 4.1.2.

By observing the inlined code, we see that the two packages are equivalent on all lines, except the ones which compute the ballot. The advantage for the adversary can thus be bounded by the advantage distinguishing $c^x \cdot g^v$ from $c^x \cdot g^{\neg v}$. Since x is sampled uniformly, the distinguishing advantage can only be zero if there exists an element x' such that $c^x \cdot g^v = c^{x'} \cdot g^{\neg v}$ and x' is equally likely to be sampled from the distribution as x.

```
Run(l : ledger):
  sk <$ uniform  S
  put sk_loc_i := sk
  zkp ← SCHNORR.RUN ( g^x , x)
  pk ← ( g^x , zkp)
  xj <$ uniform  S
  let y := g^x j
  zkp ← SCHNORR.RUN (y, xj)
  let l[j] := (y, zkp)
  let l[i] := pk
  let key := (compute_key l i) in
  put ck_loc_i := key
  x ← get sk_loc_i
  c ← get ck_loc_i
  if b then
    let vote := (c^x · g^v) in
    zkp ← CDS.RUN  g^x, c^x, v
    ret vote
  else
    let vote := (c^x · g^¬v) in
    zkp ← CDS.RUN  g^x, c^x, ¬v
    ret (vote, zkp)
```

Figure 1. $Scheduler \circ P_i^b \circ ZK_{real}$

This line of reasoning, however, poses an issue. When comparing the two programs with one program computing with x and another with x', we see that the ballots are equal, but the public keys (g^x and $g^{x'}$) would be different.

The value of c however, is computed from a bijective function on x_j. So rather than using x we can use x_j to compare the distributions. To do so we need to apply the DDH assumption, which is defined in Table 6.

The proof is given by the following steps: First, we alter the code from Figure 1 to use the DDH^1 package to sample the random values. Next, we show that c is indistinguishable from random. This step requires finding the coupling f such that

$$f(\text{compute\_key l i}) = x_j$$

However, simply proving that f exists is not enough. For the proof to be valid, we must supply the specific f, which will depend on the content of the ledger. Computing f from a fixed ledger is straightforward: For each product in compute_key, we simply perform the inverse operation. This

```
package: DDH^b
mem:
Sample():
  x <$ uniform  S
  y <$ uniform  S
  z <$ uniform  S
  put secret_loc1 := x
  put secret_loc2 := y
  put secret_loc3 := z
  if b then
    ret g^x, g^y, g^xy
  else
    ret g^x, g^y, g^z
```

Table 6. DDH Package

step forces us to let party i submit their public key last. If party i did not wait, then there would be no way to construct the coupling in our model of the OVN.

After replacing c with a random value, we replace DDH^1 with DDH^0. With DDH^0 we compute the ballot as $g^z g^v$, from which it is easy to find a value z' for which $g^z g^v = g^{z'} g^{\neg v}$. Since the computed value depends on z, none of the state variables, nor the public key, are changed by the operation. Lastly, we replace DDH^0 with DDH^1 and conclude the proof.
From this proof, we can conclude the following lemma:

Lemma 3.2 (OVN ballot secrecy). *For any adversary $\mathcal{A}$ the ballot produced by an honest party subject to the scheduling has maximum ballot secrecy with advantage:*

$$\alpha(\textbf{\textit{Scheduler}} \circ P_i^1 \circ ZK_{real}, \textbf{\textit{Scheduler}} \circ P_i^0 \circ ZK_{real})$$
$$= 2 \cdot \alpha(DDH^0, DDH^1)(\mathcal{A} \circ \textbf{\textit{Scheduler}})$$

Lemma 3.2 was formalized inside SSProve by utilizing both the theorem about packages and the probabilistic program logic. An overview of the proof is given in Figure 2. All steps where the adversary has no advantage were proven using the program logic. The steps where the adversary gains advantage use the package theorems.

4 VERIFYING THE OVN SMART CONTRACT

Annenkov et al. [2] implement the Open Vote Network inside the ConCert [3] smart contract verification framework. They prove functional correctness (the correctness of the tally) of the OVN implementation. They also provide a formally verified transformation from the ConCert

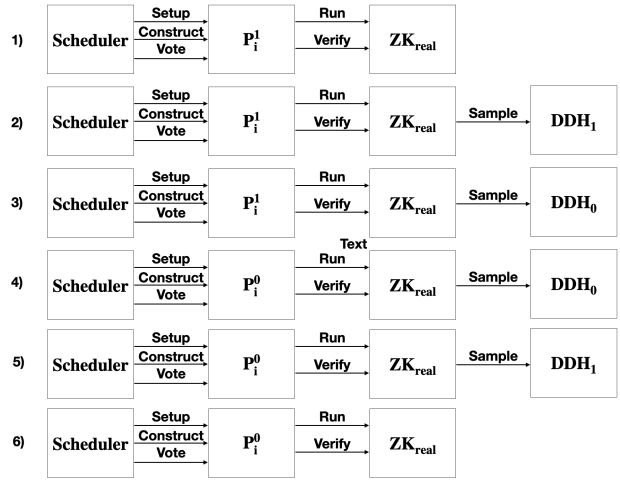

Figure 2. Package-level proof steps of Lemma 3.2.

From programs 1 to 2, the program remains unchanged after inlining giving the adversary no advantage.
From program 2 to 3, we apply the DDH assumption yielding advantage $\alpha(DDH^0, DDH^1)(\mathcal{A})$.
From program 3 to 4, we use the program logic to reason about the ballot distribution. This step yields no advantage.
Going from program 4 to 5, we apply the DDH assumption in the opposite direction yielding the advantage $\alpha(DDH^1, DDH^0)(\mathcal{A})$.
Last, from program 5 to 6, we use the same transformation as from program 1 to 2.
The final advantage is $2 \cdot \alpha(DDH^0, DDH^1)(\mathcal{A})$.

framework to the LIGO smart contract language, which is used on the Tezos blockchain. The OVN is written in gallina, the dependently typed programming language of Coq, and then transformed to a simply-typed program within Coq. Finally, this is translated with a small bit of unverified glue code to the LIGO language.

4.1 The relation between ConCert and SSProve for OVN

We provide an informal comparison between the OVN formalization in ConCert [2] and our SSProve formalization. First, we observe that the two implementations of the voting party are functionally equivalent. We do not prove this fact formally. Next, we give an informal argument that our SSProve model of the blockchain execution from Section 3 is compatible with that of ConCert.

4.1.1 Equivalence of the voting implementations. In the SSProve formalization in Section 3, we described three constituent parts: the single-party functionality, zero-knowledge, and the blockchain model. To relate the ConCert and SSProve version of the OVN, we look at the implementation of a

voting party. The implementation of a voting party in ConCert should be equivalent to the honest party functionality of SSProve.

First, let us consider SSProve's honest party functionality. The package encapsulating an honest party can be divided into two parts: 1) the code responsible for computing the public key and the ballot, and 2) the SSProve-specific code for storing secret variables in state between procedure calls.

For the ConCert formalization of the OVN, the secret variables are passed into the OVN functions. It is the responsibility of the caller to ensure the secrecy of the variables. The ConCert formalization can additionally be split into two parts: 1) functions responsible for computing the public key and the ballot, and 2) the activation functions controlling the blockchain interaction. The activation functions are responsible for calling the functions computing protocol-specific functions at the correct time. For the case of the OVN, the activation functions ensure that the function computing the public key is used in round 1. Conversely, the function computing the ballot should be called if the protocol is in round 2.

To compare the imperative-style SSProve formalization to the state-passing style of the ConCert formalization we give the following theorem (which we have not checked in Coq):

Theorem 4.1. *For any procedure in the honest party functionality with state handling removed, the corresponding (non-activation) function in the ConCert formalization is equivalent if the inputs and the secret variables are equivalent.*

We now turn our attention to how Theorem 4.1 can be proved.

Proof of Theorem 4.1. First, we define the three functions which constitute the core of the OVN:

$$\text{compute\_public\_key}(x : \mathbb{Z}_{|G|}) = g^x$$

$$\text{compute\_key}(l : G\ \texttt{list}, i : \mathbb{N}) = \prod_{j=0}^{i} l[j] / \prod_{j=i}^{n} l[j]$$

$$\text{compute\_vote}(c : G, x : \mathbb{Z}_{|G|}, v : \mathbb{B}) = c^x g^v$$

And slightly refactor the SSProve implementation as seen in Table 7.

With state handing clearly separated from the OVN functionality, we are left with comparing the functions of the SSProve side with those on the ConCert side.

The proof of functional equivalence follows the same pattern for all three functions. The main difficulties lie in connecting different formalizations of finite fields.

4.1.2 Model of blockchain. In the SSProve formalization, the Scheduler is responsible for modeling the blockchain's and the adversary's capabilities. The scheduling is specifically restricted to let party i wait for all other parties before voting. This restriction is purely an artifact of how uniform

```
package: P_i^b
mem:  sk_loc_i: secret
      ck_loc_i: public
      vote: bool
```

```
Setup():
  sk <$ uniform  S
  put sk_loc_i := sk
  zkp ← SCHNORR.RUN ( g^x , x)
  ret (compute_public_key x,
   zkp)

Construct(l : ledger):
  assert (map (fun (pk, zkp)
   -> ZKPVerify zkp) (values l
   ))
  let key := (compute_key l i)
  put ck_loc_i := key
  ret ()

Vote():
  x ← get sk_loc_i
  c ← get ck_loc_i
  if b then
    let vote := (compute_vote
   c x v)
    zkp ← CDS.RUN g^x, c, v
    ret vote
  else
    let vote := (compute_vote
   c x ¬v)
    zkp ← CDS.RUN g^x, c, ¬v
    ret (vote, zkp)
```

Table 7. Refactored Honest Party Package.

sampling works in formal cryptographic frameworks. Specifically, our security proof relies on computing the inverse of the compute_key function. The inverse can only be found if all the arguments of the function are known.

Orthogonal to fixing the scheduling for one party, the Scheduler ensures that the procedures of P_i are called in the correct order. Lastly, the Scheduler is responsible for maintaining the ledger.

A finite map represents the ledger and the Scheduler will only write to a specific key once. This follows the ideal functionality of a blockchain, where blocks are fixed after their creation.

The modeling of a blockchain in ConCert is more precise. In ConCert, each party submits their public keys and ballots to the blockchain, so that they eventually become part of a block and are published to the chain. The OVN in ConCert is therefore based on activation functions. A party can only proceed after all public keys/ballots have found their way onto the blockchain. Both the SSProve and the ConCert OVN formalizations ensure that the different rounds of the OVN are called in order.

5 RELATED WORK

The Open Vote Network was implemented as a smart contract on the Ethereum blockchain by McCorry et al. [20]. A more efficient protocol [28], which moves more of the computation off-chain, allows one to scale OVN to large groups of participants. Neither of these works applies formal methods to ensure that the smart contract behaves as expected.

Smart contract verification. Aspects of Ethereum's smart contract language have been formalized in interactive proof assistants; e.g. [13, 19]. However, these works focus on the verification of the compiler, not on individual smart contracts. We are not aware of smart contract verification frameworks in interactive theorem provers (Coq, Isabelle, F*, EasyCrypt, ...) that could be combined with the frameworks for cryptographic security mentioned in the introduction.

Verification of Helios. Helios is a popular voting protocol. It has been analyzed [4] with the Tamarin [21] protocol verifier, which uses the symbolic model. Tamarin does not consider the implementation of the protocol. Moreover, the Helios protocol does not include a blockchain.

Ballot privacy for the Helios protocol has also been verified [7] in the Easycrypt theorem prover which uses the, more precise, computational model. This analysis was later extended to the Belenios protocol [8]. Easycrypt is similar in scope to SSProve. However, Easycrypt does not allow one to extract executable code and does not provide a framework for the verification of smart contracts.

Rivest [25] proposes 'Software Independence' as an important property for online voting protocols: one should isolate the trusted computing base (TCB) of the voting protocol to independently verifiable tallying of the votes. Of course, these verifiers should be correctly implemented. With this in mind, Haines et al. [16] have extracted verified verifiers for the Helios protocol, using the Coq proof assistant. This extraction is similar in style to our use of ConCert. A difference is that ConCert uses the more modern verified extraction. Haines et al. reasons about Σ-protocols, like we do, but do not verify probabilistic properties. They work in a symbolic model with perfect encryption and where negligible events never happen.

Election Guard. Microsoft's election guard[1] is a popular system for safe offline and online voting which enjoys good properties such as universal verifiability. A formal analysis has been carried out [14]. They use a symbolic model to prove special soundness and honest-verifier zero-knowledge of the verifier in Coq.

The Electis project[2] combines ElectionGuard with the Tezos blockchain as a public bulletin board. No formal security analysis of this combined protocol is available, although one would hope that many of the properties of ElectionGuard carry over, and some are improved by the use of a blockchain.

Cryptographic primitives. The smart contract in ConCert is parameterized by a group and a hash function. In the extracted contracts simple, insecure implementations are used. This can be improved by slightly increasing the TCB and printing the group to the group of the secp256k1 (bitcoin) or ed25519 elliptic curves, which are natively supported in LIGO. For the hash functions, one could use blake2b or sha3. These primitives come from the HACL* library [31] which has been formally verified for correctness in F*. To narrow the gap been F* and Coq, one could use their hacspec [22] specification as a bridge.

6 CONCLUSION

We have provided a complete analysis of a cryptographic smart contract, which is both provably functionally correct and cryptographically secure.

7 ACKNOWLEDGEMENTS

This work was partially supported by the Concordium Blockchain Research Center at Aarhus University and by the AFOSR grant Homotopy type theory and probabilistic computation (12595060).

REFERENCES

[1] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Theo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 1–15. https://doi.org/10.1109/CSF51468.2021.00048

[2] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2021. Extracting Smart Contracts Tested and Verified in Coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM. https://doi.org/10.1145/3437992.3439934

[3] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. 2020. ConCert: A Smart Contract Certification Framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. nil. https://doi.org/10.1145/3372885.3373829

[1] https://www.electionguard.vote/

[2] https://www.electis.io/. Accessed: 28 april 2022.

[4] Sevdenur Baloglu, Sergiu Bursuc, Sjouke Mauw, and Jun Pang. 2021. Election Verifiability Revisited: Automated Security Proofs and Attacks on Helios and Belenios. In *CSF*. IEEE, 1–15.

[5] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. 2020. CryptHOL: Game-Based Proofs in Higher-Order Logic. *Journal of Cryptology* 33, 2 (April 2020), 494–566. https://doi.org/10.1007/s00145-019-09341-z

[6] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. 2018. State Separation for Code-Based Game-Playing Proofs. 222–249. https://doi.org/10.1007/978-3-030-03332-3_9

[7] Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, Benedikt Schmidt, Pierre-Yves Strub, and Bogdan Warinschi. 2017. Machine-Checked Proofs of Privacy for Electronic Voting Protocols. 993–1008. https://doi.org/10.1109/SP.2017.28

[8] Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, and Bogdan Warinschi. 2018. Machine-Checked Proofs for Electronic Voting: Privacy and Verifiability for Belenios. 298–312. https://doi.org/10.1109/CSF.2018.00029

[9] Véronique Cortier, Constantin Cătălin Drăgan, François Dupressoir, Benedikt Schmidt, Pierre-Yves Strub, and Bogdan Warinschi. 2017. Machine-Checked Proofs of Privacy for Electronic Voting Protocols. In *2017 IEEE Symposium on Security and Privacy (SP)*. 993–1008. https://doi.org/10.1109/SP.2017.28

[10] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. 1994. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. 174–187. https://doi.org/10.1007/3-540-48658-5_19

[11] Amos Fiat and Adi Shamir. 1987. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. 186–194. https://doi.org/10.1007/3-540-47721-7_12

[12] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract). 291–304. https://doi.org/10.1145/22145.22178

[13] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 243–269.

[14] Thomas Haines, Rajeev Goré, and Jack Stodart. 2020. Machine-Checking the Universal Verifiability of ElectionGuard. In *NordSec (Lecture Notes in Computer Science, Vol. 12556)*. Springer, 57–73.

[15] Thomas Haines, Rajeev Goré, and Mukesh Tiwari. 2019. Verified Verifiers for Verifying Elections. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. nil. https://doi.org/10.1145/3319535.3354247

[16] Thomas Haines, Rajeev Goré, and Mukesh Tiwari. 2019. Verified Verifiers for Verifying Elections. 685–702. https://doi.org/10.1145/3319535.3354247

[17] F. Hao, P. Ryan, and Piotr Zielinski. 2010. Anonymous Voting by Two-Round Public Discussion. *IET Inf. Secur.* (2010). https://doi.org/10.1049/IET-IFS.2008.0127

[18] Aggelos Kiayias and Philip Lazos. 2022. SoK: Blockchain Governance. https://doi.org/10.48550/ARXIV.2201.07188

[19] Diego Marmsoler and Achim D. Brucker. 2021. A Denotational Semantics of Solidity in Isabelle/HOL. In *SEFM (Lecture Notes in Computer Science, Vol. 13085)*. Springer, 403–422.

[20] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. 2017. A Smart Contract for Boardroom Voting with Maximum Voter Privacy. In *Financial Cryptography and Data Security (Lecture Notes in Computer Science)*, Aggelos Kiayias (Ed.). Springer International Publishing, Cham, 357–375. https://doi.org/10.1007/978-3-319-70972-7_20

[21] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *CAV (Lecture Notes in Computer Science, Vol. 8044)*. Springer, 696–701.

[22] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. 2021. *Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust.* Technical Report. Inria. https://hal.inria.fr/hal-03176482

[23] Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. 2022. Formalising Decentralised Exchanges in Coq. https://doi.org/10.48550/ARXIV.2203.08016

[24] Jakob Botsch Nielsen and Bas Spitters. 2019. Smart Contract Interactions in Coq. In *FM Workshops (1) (Lecture Notes in Computer Science, Vol. 12232)*. Springer, 380–391.

[25] Sunoo Park, Michael Specter, Neha Narulaand, and Ronald L Rivest. 2021. Going from bad to worse: from Internet voting to blockchain voting. *Journal of Cybersecurity* 7, 1 (2021).

[26] Mike Rosulek. 2022. *The Joy of Cryptography.* https://joyofcryptography.com

[27] Claus Schnorr. 1991. Efficient Signature Generation by Smart Cards. *Journal of Cryptology* 4 (Jan. 1991), 161–174. https://doi.org/10.1007/BF00196725

[28] Mohamed Seifelnasr, Hisham S. Galal, and Amr M. Youssef. 2020. Scalable Open-Vote Network on Ethereum. In *Financial Cryptography Workshops (Lecture Notes in Computer Science, Vol. 12063)*. Springer, 436–450.

[29] The Coq Development Team. 2022. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.5846982

[30] Santiago Zanella Béguelin. 2010. *Formal Certification of Game-Based Cryptographic Proofs.* Ph. D. Dissertation. École Nationale Supérieure des Mines de Paris.

[31] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1789–1806.