Verified Verifiers for Verifying Elections

Thomas Haines
Dept of Mathematical Sciences,
Norwegian University of Science and Technology
Trondheim, Norway
thomas.haines@ntnu.no

Rajeev Goré
Research School of Computer Science,
Australian National University
Canberra, Australia
rajeev.gore@anu.edu.au

Mukesh Tiwari
Research School of Computer Science,
Australian National University
Canberra, Australia
mukesh.tiwari@anu.edu.au

ABSTRACT

The security and trustworthiness of elections is critical to democracy; alas, securing elections is notoriously hard. Powerful cryptographic techniques for verifying the integrity of electronic voting have been developed and are in increasingly common use. The claimed security guarantees of most of these techniques have been formally proved. However, implementing the cryptographic verifiers which utilise these techniques is a technical and error prone process, and often leads to critical errors appearing in the gap between the implementation and the formally verified design.

We significantly reduce the gap between theory and practice by using machine checked proofs coupled with code extraction to produce cryptographic verifiers that are themselves formally verified. We demonstrate the feasibility of our technique by producing a formally verified verifier which we use to check the 2018 International Association for Cryptologic Research (IACR) directors election.

KEYWORDS

verifiable e-voting; interactive theorem provers; code extraction

ACM Reference Format:
Thomas Haines, Rajeev Goré, and Mukesh Tiwari. 2019. Verified Verifiers for Verifying Elections. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3354247

1 INTRODUCTION

Electronic voting is discussed regularly as a potential solution to many problems of running elections from cost to usability; also, there is a common conception that it will increase voter turnout rates. Security experts, however, are well aware that electronic voting is a security nightmare. Consequently, there has been much effort to develop usable schemes that produce publicly verifiable evidence attesting to the correctness of their results. A scheme where voters receive assurance that their vote was correctly included in the tally is called end-to-end verifiable. Generally, this property is broken down into three sub-properties: namely, cast-as-intended, collected-as-cast, and counted-as-collected. For each of these sub-properties, the overall scheme must produce verifiable evidence, often utilising cryptographic techniques such as sigma protocols and zero-knowledge proofs, as we describe shortly. Of course, care must be taken that the conjunction of these sub-proprieties actually implies end-to-end verifiability as noted by Küsters et al [36].

The theoretical foundations of such end-to-end verifiable electronic voting schemes are maturing but the propensity of practice towards broken implementations is concerning. For example, although Switzerland has, perhaps, the most rigorous requirements and testing for its electronic voting, we have seen the withdrawal of several proposed end-to-end verifiable systems because, even after much analysis and certification, the implementations were fatally flawed. For the cornucopia of errors in the Swiss post system see Teague et al.'s excellent write up.[1] Some other prominent failures and issues in allegedly end-to-end verifiable systems have included the I-Vote system deployed in the Australian state of New South Wales [33], and the e-voting system used in national elections in Estonia [43]. Many general issues have also been discovered [10, 11, 18] which need to be carefully avoided in any implementation, but most of these issues were at one time present in the Helios end-to-end verifiable e-voting system [1] used by the International Association for Cryptological Research.

Public systems and prototypes have provided such a richness of trivial but critical errors that trust in any system without extraordinarily careful analysis is unwarranted, regardless of any claimed verifiability proprieties. It seems self-evident that there is a critical lack of people with sufficient technical skill in programming and cryptography to check these implementations.

We propose a different approach; by leveraging the various verifiability properties of modern end-to-end verifiable e-voting schemes it suffices to have a correct verifier for the (evidence produced by the) scheme to ensure its integrity, regardless of any flaws in the actual e-voting implementation; Rivest calls this property software independence [40]. We can utilise interactive theorem provers and code extraction to produce these verifiers with a very high degree of confidence, which depends on the strong guarantees of correctness provided by the interactive theorem prover rather than the new and un- or under- studied e-voting system implementation.

Thus our title refers to the formal verification of software for verifying the evidence produced by end-to-end verifiable schemes; specifically, the software that implements the cryptographic primitives and zero-knowledge proofs (ZKP), including sigma protocols and mixnets, as used in an actual election. It does **not** refer to

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11...$15.00
https://doi.org/10.1145/3319535.3354247

[1] https://people.eng.unimelb.edu.au/vjteague/SwissVote.html

the formal verification of the code that actually implements vote-casting, or vote-transmission, or vote-counting. Indeed, according to "software independence", we do not actually need to verify these e-voting implementations at all to ensure the integrity of the election outcome if our verifiers for them are verified.

Formal verification implies that we have a logic-based proof that the software is correct with respect to some logical specification. By their nature, ZKPs generate cryptography-based proofs that are published so that a scrutineer can (cryptographically) verify that these proofs correctly vouch for the published result. Hence, our title states that our verified software will accept the public evidence for the election only if the election was run correctly in the cryptographic sense. To ensure that the claim of correctness is itself correct, we give logic-based proofs that the cryptographic primitives and ZKP implementations satisfy the logical specifications of their relevant security requirements. Thus these two notions of verification beget two notions of proof: formal verification relies on a proof that the code meets its specification; ZKPs generate proofs that are published so that a scrutineer can check that these proofs correctly vouch for the published result.

1.1 Helios and the IACR directors election 2018

We now explain the Helios voting system as used in the 2018 IACR directors election. We apologise to non-experts as we cannot avoid using e-voting and cryptographic jargon in this description. Note, nothing in this section refers to formal verification!

The 2018 IACR directors election considered seven candidates to fill three positions on the board of directors. All members of the IACR were eligible to vote and could vote for as many candidates as they wished, with the three candidates receiving the most votes being elected; this style of voting is called approval voting.

The Helios voting system [1] v4 was used for the election; configured with four authorities, who generated an ElGamal [26] public key such that all four authorities were required to decrypt efficiently. Each voter uses her personal computer to log into the system using credentials she receives by email. Using her own computer, the voter creates seven ElGamal ciphertexts; one for each candidate, encrypting either zero or one in the exponent. Since the vote is in the exponent, the ElGamal cryptosystem becomes additively homomorphic. The voter's computer proves using a sigma protocol—a particular kind of efficient ZKP—that the ballot is an encryption of zero or one. The voter is then offered the chance to audit her encrypted ballot to check that it does indeed contain the vote she intended. If she chooses to audit, she must discard this ballot—but can cast a fresh ballot; this mechanism is called a Benaloh challenge [8]. Once she has an unaudited ballot with which she is happy, she casts it. The Helios website maintains an append-only bulletin board on which the voter's encrypted ballot appears, as well as all other public information and evidence related to the election. After the voting period is over, all the encrypted ballots corresponding to each candidate are multiplied together; so that there is now a single ciphertext for each candidate, encoding the number of votes for that candidate. The authorities then decrypt these (seven) ciphertexts, announce the results and prove, using a sigma protocol, that the announced result is the correct decryption.

Having explained the scheme we now explain its verifiability properties. The scheme enjoys cast-as-intended verifiability courtesy of the Benaloh challenges since the voter can continue to challenge until she is convinced that her own computer is not compromised and is producing ballots according to her wishes. It enjoys collected-as-cast verifiability since the voter can directly check that her encrypted ballot was included on the bulletin board. The more complicated step is the counted-as-collected check. At this stage, there is a published list of encrypted ballots on the board and a published result. It is easy for an arbitrary scrutineer to homomorphic tally the encrypted votes, one simply reruns the (multiplication) computation and checks the result matches the published one. The two remaining steps are to check that the encrypted ballots actually encrypt zero or one and that the tallied ciphertexts are decrypted correctly.

We will now introduce the notation and techniques required to check that the ballots are correctly encrypted and decrypted. Recall that a binary relation R is a subset of the Cartesian product of two sets. Given a set S of statements and set W of possible witnesses, we use R to denote the relationship between S and W. A sigma protocol for relationship R is used to produce a cryptographically correct ZKP proof for a given statement $s \in S$, that the prover knows a witness $w \in W$ such that $(s, w) \in R$, without divulging w itself. Recall that an ElGamal ciphertext over a cyclic group G is a pair of group elements (c_1, c_2) for a given generator g and public key y. We assume that G has prime order q and denote by F the the field of integers mod q.

The sigma protocol for correct encryption for this election uses $(G * G * G * G)$ as S and F as W and is a sigma protocol for the relation R' of four group elements $(g, c_1, y, c_2) \in S$ and one field element $r \in W$ where the pair $((g, c_1, y, c_2), r) \in R'$ if $(g^r = c_1 \wedge y^r = c_2) \vee (g^r = c_1 \wedge y^r = c_2/g)$.

The sigma protocol for correct decryption uses $(G * G * G * G)$ as S and F as W and is a sigma protocol for the relationship R'' of five group elements $(g, y, c_1, c_2, m) \in S$ and one field element $x \in W$ where the pair $((g, y, c_1, c_2, m), x) \in R''$ if $g^x = y \wedge c_1^x = c_2/m$.

To enable scrutiny, the election authority publishes, non-interactive, sigma protocol transcripts for correct encryption and decryption. Thus a scrutineer can verify the election result by checking the following three things. First, the scrutineer checks that the transcripts are valid for all encrypted ballots; this prevents the ballot stuffing attack where a ciphertext encodes more than one vote. Second, the scrutineer reruns the (multiplication) computation and checks that the resulting ciphertexts match the published ones. Finally, the scrutineer checks that the transcripts are valid for the decryption of these combined ciphertexts with respect to the announced result. These three checks suffice to ensure that the ballots were counted-as-collected (which we formally state in theorem `HeliosCorrectResultApproval`).

These three checks cannot be done by hand for millions of ballots, so we need a computer program: a verifier. The verifier must be general enough to cover any election result obtained via Helios. How can we guarantee that this verifier (program) is correct?

1.2 Clarifying our aims: the fine print :)

Our aim is to produce formally verified software for verifying election results in the sense of the three checks mentioned above. In particular, our aim is not to formally verify all the security and privacy guarantees of Helios, nor to verify that the election authority's implementation of Helios is perfect, but rather to guarantee that the cryptographic evidence produced by any Helios implementation and published by the election authority passes the three checks mentioned above, thus guaranteeing that the published results are correct. This is inherently a per election proposition, so we must verify that a particular set of encrypted ballots are correctly constructed and tally to the claimed result.

Since we are producing verified software for verifying election results, which is totally different to producing a verified version of Helios itself, privacy is out of scope and so is any expected behaviour not enforced by the Helios definition of verifiability. Specifically, this means that for some deployed schemes which do not offer end-to-end verifiability, our verifiers can only check those properties which are publicly verifiable.

Nevertheless, our work already contains the primitives required for verifying cast-as-intended and collected-as-cast. We cover all three steps where applicable, though we were only able to verify the last step properly in the case of Helios as used in the IACR2018 election, because both the cast-as-intended and collected-as-cast are individually not universally verifiable for that election. Indeed, we have checked the audited ballots of Helios (for cast-as-intended) as used in the IACR2018, even though, strictly speaking, this check should be performed by the voter before casting.

2 BACKGROUND

Interactive theorem provers are computer programs (tools) which allow a user to encode mathematically rigorous definitions, state desired properties (as theorems), and interactively and formally prove that the definitions imply the theorems. Although they provide some automated proof-search facilities, the theorems to be proved invariably require human guidance, so the tool accepts directions for using a given finite collection of proof-rules, and only accepts a putative proof if the proof-rules are applied correctly. Trust rests upon three pillars: first, the code base for interactive theorem provers is usually very small and has been scrutinised by many experts, typically over several decades; second, most interactive theorem provers produce a machine-readable proof of the claimed theorem and these proofs can be checked either by hand or by a different interactive theorem prover; third, interactive theorem provers typically enjoy extremely rigorous mathematical foundations, which have withstood decades of peer review. Many interactive theorem provers are able to transliterate (extract) correct proofs into ML, Haskell, Scheme or OCaml programs.

The main impediment to using interactive theorem proving and code extraction is the rather steep learning curve involving exotic mathematical logic(s) and the associated proof-rules. Consequently, interactive theorem provers mostly remained in an academic setting [31] [27], and were rarely considered for real life software-engineering. Recent debacles, such as heartbleed[2], have led companies and researchers to focus on avoiding bugs using formal verification, to the point where it is now gaining momentum in mainstream cryptographic development including verification of Google BoringSSL[22] in Coq, HACL* in F* [49] used in Firefox, verification of correctness and security of OpenSSL HMAC[9], verification of elliptic curve Curve25519 [16], and verified side channel security of MAC-then-Encode-then-CBC-Encrypt (MEE-CBC)[4].

Secure electronic voting has been extensively studied since Chaum's seminal work [15]. In modern e-voting schemes, extensive use is made of zero-knowledge proofs (ZKPs), first studied by Goldwasser, Micali, and Rackoff [30]. ZKPs enable cryptographic, rather than formal, verification of the election result without revealing information which adversely affects the privacy of the election. Many ZKPs are of a particularly simple and efficient form known as a sigma protocol; a class first defined and analysed by Cramer in his PhD Thesis [19]. The other main type of ZKPs used in electronic voting are verifiable mixnets [24, 39], which allow encrypted ballots to be secretly but verifiably shuffled before decryption.

Recently, there has been a significant focus on using automated tools to create and check proofs in e-voting. The earliest tool in use in e-voting appears to be ProVerif which was used by Delaune et al [21] to reason formally about privacy; ProVerif was also used to reason formally about cryptographic verifiability by Smyth et al [42]. More recently tools such as EasyCrypt and particular Tamarin have both been used to formally verify various e-voting schemes [14, 34] with the Cortier et al [17] work on (cryptographic) verifiability and privacy of Belenaios being one of the best examples. However, these prior works focus on the privacy or integrity of the (theoretical) scheme itself while we are concerned primarily with the integrity of a deployed implementation of the scheme.

Largely tangential to the existing work mentioned above are the efforts to produce correct counting software for elections. This may seem strange to people more familiar with first-past-the-post elections, but more complex vote-counting methods such as single-transferable-vote and instant-runoff-voting have counting functions which are non-trivial. There have been a series of papers showing that techniques similar to what we suggest can be used to ensure such complex elections are correctly tallied [28, 29, 38].

The definitions we have formally verified are inspired by Smyth et al [41]; specifically, we formally verify that the cryptographic verifier we generate guarantees both the correctness and soundness aspect of the universal verifiability of the scheme. This is the best that can be achieved for a scheme like Helios where cast-as-intended and collected-as-cast verifiability are not universally verifiable. Schemes such as Helios are said to have individual verifiability for cast-as-intended and collected-as-cast since only voters themselves can check these properties hold for their own ballot.

We used the Coq theorem prover [12] which is based upon Coquand's Calculus of Constructions and has been developed over decades. Part of our work rests upon analysing sigma protocols in Coq and extracting efficient implementations, which has been done before by Barthe et al [5]. Almeida et al [2] developed a compiler which accepts an abstract description of the statement to be proved and produces an implementation of a sigma protocol for that statement along with an Isabelle/HOL proof that the sigma protocol is correct. Both of these works were combined and expanded upon by Almeida et al [3]. Thus there is no barrier to using their work for

[2] http://heartbleed.com/

the small subset of e-voting schemes which can be verified using only sigma protocols; however, as far as we are aware, this has not been done to date.

There are three main differences between our work and that of Almeida et al [3]. Firstly, their approach is more general while ours is specific to the kinds of sigma protocols commonly found in e-voting. These specifics allow us to define and prove generic combinations of sigma protocols which are not otherwise available, such as proving you a witness such that it satisfies two distinct statements. Secondly, in their own words, the "*catch is that our verification component is highly specialized for (a specific class of) ZK-PoK and relies on in-depth knowledge on how the protocol was constructed.*". However, since we aim at verifying existing deployed e-voting implementations we need to prove that the deployed sigma protocol is correct, and extract a provably correct verifier for it. Almeida et al's work would give us a correct sigma protocol for the statement but not a verifier for the existing election. Thirdly, while there are some electronic voting schemes which can be verified using only sigma protocols, the majority use verifiable mixnets. Verifiable mixnets, sometimes called proofs of shuffle, have not been formally verified to be secure before in Coq, or any theorem prover, and the ability to produce verified correct implementations of these more complicated ZKPs is entirely non-trivial. That is, while the work culminating in Almeida et al [3] is impressive, it is not applicable to our primary aim of verifying real elections.

2.1 Verification and Code Extraction Via Coq

We now explain how to use the interactive theorem prover called Coq [12] to: encode specifications; verify (functional) programs correct against these encoded specifications; and extract the code corresponding to the verified functional programs. We first describe the differences between classical logic and intuitionistic logic since the latter is indispensable for code extraction. We then describe how the basic Coq proof engine works and briefly describe the connection to functional programs. Finally, using a running example, we describe how to specify, verify and extract code using Coq.

2.1.1 Classical Logic and Constructive Logics. We assume familiarity with classical logic, but list three of its defining features:

(1) excluded middle: every statement is true or false;
(2) non-contradiction: no statement is both true and false; and
(3) non-empty domain of discourse: the values of variables such as x and y are drawn from a non-empty set.

intuitionistic logic elides the law of the excluded middle and demands that for an existential $\exists x.\varphi(x)$ to be true, we must find a witness a from the domain of discourse which makes $\varphi(a)$ true. Thus we cannot assume $A \vee \neg A$, and then proceed by cases on A and $\neg A$. Nor can we proceed by contradiction whereby we assume $\neg A$, show that this leads to a contradiction, and hence conclude that A must hold. Intuitionistic logic is "constructive" because to conclude $A \vee B$, we must construct a proof of A or a proof of B, and to conclude $\exists x.\varphi(x)$, we must construct a witness a as explained above. Consequently, finding proofs in intuitionistic logic is usually harder than in classical logic.

2.1.2 An overview of the Coq proof engine. At all stages of a Coq proof, the proof engine maintains a collection of labelled hypotheses or assumptions $t_1 : \alpha_1, \cdots, t_n : \alpha_n$, one current goal γ_0, and a list of further goals $\gamma_1, \cdots, \gamma_m$ as illustrated below at left.

$$
\begin{array}{c}
t_1 : \alpha_1 \\
t_2 : \alpha_2 \\
\vdots \\
t_n : \alpha_n \\
\hline\hline
\gamma_0 \\
\gamma_1, \cdots, \gamma_m
\end{array}
\qquad
\begin{array}{c}
\alpha_2 \rightarrow \beta \\
\alpha_2 \\
\beta \\
\vdots \\
\alpha_n \\
\hline\hline
\gamma_0 \\
\gamma_1, \cdots, \gamma_m
\end{array}
\qquad
\begin{array}{c}
\forall x, \beta_1(x) \rightarrow \beta_2(x) \\
\alpha_2 \\
\vdots \\
\alpha_n \\
\hline\hline
\beta_1(x)\theta \\
\gamma_1, \cdots, \gamma_m
\end{array}
$$

Ignoring the labels t_i for now, proof construction then either proceeds in a forward or a backward manner using a finite collection of predefined "natural deduction" rules. For example, as shown above centre, if α_1 is of the form $\alpha_2 \rightarrow \beta$, then we may extend the assumptions with β by apply the rule of *modus ponens* which intuitively captures "if $\alpha_2 \rightarrow \beta$ and α_2 then β". Alternatively, as shown in the rightmost figure, if α_1 is of the form $\forall x, \beta_1(x) \rightarrow \beta_2(x)$, then we can pattern-match γ_0 with $\beta_2(x)$ to obtain a substitution θ such that $\beta_2(x)\theta = \gamma_0$, and then replace the goal γ_0 with $\beta_1(x)\theta$, to "backchain" on the implicational assumption instance $\beta_1(x)\theta \rightarrow \beta_2(x)\theta$. Coq will only accept a putative proof if all rules are used correctly, thereby guaranteeing overall correctness.

2.1.3 Proofs as programs and code extraction. The syntax of the basic propositions α and β is user-definable and is based upon a highly sophisticated type-theory which allows all of the logical manipulations mentioned above to be interpreted purely inside a lambda-calculus of terms with the logical formulae as types where $t_1 : \alpha_1$ is now read as "term t_1 is of type α_1". For example, the *modus ponens* rule corresponds to function application: "if f is a function from domain type α_2 to range type β, and t is of type α_2 then $f(t)$ is of type β". By using the type annotations, we can also read $t : \alpha_2$ as "t is a proof of α_2", read $f : \alpha_2 \rightarrow \beta$ as "f is a function that converts proofs of α_2 into proofs of β", and read $f(t) : \beta$ as "$f(t)$ is a proof of β". Thus a successful proof corresponds to a computable function in the underlying lambda-calculus. Coq provides "extraction" facilities to turn such computable functions into actual code in one of programming languages OCaml, Haskell or Scheme.

2.1.4 Program Verification via Coq. Coq also provides a vast array of pre-defined constructs from functional programming such as natural numbers, lists, pattern matching and explicit function definitions. Below, we explain the two ways in which one can produce verified programs via Coq using addition of two natural numbers as an example. We use a format whereby we first give a natural language definition as might be found in a mathematics text, then its encoding into Coq, followed by an explanation of the encoding.

Definition 2.1. The set *mynat* is the smallest set formed using the following clauses:

(1) the term O is in *mynat*;
(2) if the term n is in *mynat* then so is the term $S\ n$;
(3) nothing else is in *mynat*.

```
Inductive mynat : Set :=
  | O : mynat              (* O is a mynat *)
```

```
    | S : mynat -> mynat.  (* S of a mynat is a mynat *)
```

Here, the first line encodes that *mynat* is of type *Set* and the vertical bar separates the two subclauses of the encoding. The terms O and S are known as constructors and anything in between "(*" and "*)" are comments. The first subclause illustrates that the colon can also be read as set membership ∈ while the second clause illustrates that the constructor S is actually a function that accepts a member from *mynat* and constructs another member of *mynat* by prefixing the given member with S. Thus the explicit mention of n in the natural language definition is elided. Clause (3) of the natural language definition is encoded by the declaration `Inductive`. Intuitively, the natural numbers are the terms $O, (S\ O), (S\ (S\ O)), \cdots$.

Definition 2.2 (Specification of addition). Adding O to any natural number m gives m, and for all natural numbers n, m, and r, if adding n to m gives r then adding $(S\ n)$ to m gives $(S\ r)$.

```
Inductive add: mynat -> mynat -> mynat -> Prop :=
| addO: forall m, (add O m m)
| addS: forall n m r, add n m r -> add (S n) m (S r).
```

Here, the notation $mynat \longrightarrow mynat \longrightarrow mynat \longrightarrow Prop$ encodes that *add* is ternary and that it is a "Proposition" which returns either true or else false, but in intuitionistic logic rather than classical logic. Our specification of addition is encoded as a ternary predicate *add n m r* that is true iff "adding n to m gives r", based purely on the only two ways in which we can construct the first argument: either it is O, or it is of the form $(S\ \cdot)$.

There are now two ways to proceed to "extract" the code for an implementation "myplus" of the predicate *add*. The first is to write our own function *myplus* inside Coq and to prove that the function implements the specification of addition. The second is to prove a theorem inside Coq such that the proof encodes the function implicitly. In both cases, the "extraction" facilities of Coq allow us to produce actual code in OCaml, Haskell, or Scheme.

The encoding below is our hand-crafted function *myplus* in which the "where" keyword allows an infix symbol + for *myplus* and ⇒ (not ⟶) indicates the return value of the function:

```
Fixpoint myplus (n m: mynat) : mynat :=
    match n with
    | O => m
    | S p => S (p + m)
    end
  where "p + m" := (myplus p m).
```

Theorem 2.3. *For all natural numbers n, m, r, if $r = myplus\ n\ m$ then add n m r is true.*

```
Theorem myplus_correct:
  forall n m r: mynat, (r = myplus n m) -> (add n m r).
```

The proof of this theorem must be constructed by us, interactively, using the proof-engine described in Section 2.1.2. We do not have to worry about the correctness of the proof of this theorem as Coq will ensure that it is correct. The Coq extraction mechanism turns our function "myplus" into Ocaml, Haskell or Scheme code.

The second way to extract code is to prove the following without first writing our own hand-crafted version of *myplus*.

Theorem 2.4. *For all n, m of type mynat, there is a way to construct an r of type mynat such that add n m r is true.*

```
Theorem myplus: forall n m: mynat, { r | add n m r }.
```

Here, the notation $\{r \mid add\ n\ m\ r\}$ instructs Coq to retain all the type-theoretic (algorithmic) content of the proof. The extraction facility then transliterates this content into Ocaml code in the file "myplus.ml". We omit details since we followed the first approach.

Note the following: everything we do depends upon the specification! If the specification does not capture the intended task properly then we are lost. For this reason, we go to great lengths in the sequel to describe our actual Coq specification in detail. It is only when it has been scrutinised by experts that we can convince others that we are actually verifying what we wanted to verify.

3 CONTRIBUTION

We have made the first strides in combining formal verification and code extraction with end-to-end verifiable electronic voting for real elections, thereby significantly reducing the gap between theory and practice. The current norm in electronic voting is to have multiple layers of documentation ranging from high level descriptions and proofs down to the code; these are supposed to line up but often don't, leaving a gap that has been the source of many errors. Since we reason and prove directly about the implementation of the verifier, we eliminate this gap and enable far greater confidence in the verifiability of the e-voting scheme as deployed. Our contribution consists of four main components, as detailed next.

We provide the logical machinery to easily prove implementations of the sigma protocols commonly used in e-voting correct; which is to say, we prove they satisfy special soundness, honest-verifier zero knowledge and completeness. By doing most of our work in general lemmas and theorems, we provide a general base for quickly proving cryptographic e-voting schemes secure.

We provide a verifier, which is provably correct with respect to universal verifiability, for verifying the publicly available evidence produced by the Helios e-voting system, with the caveat that we do not model the Fiat-Shamir transform. By instantiating our general techniques to the specific case of Helios, as used in the IACR 2018 director election, we produce and extract a formally verified verifier for verifying the integrity of that election.

We have verified the published results of a real election using our provably correct verifier. The verifier takes only a few minutes to verify the entire election.

We show that our machinery extends to formally verifying the implementations of verifiable mixnets to be sound, complete, and privacy preserving. While previous work has used interactive theorem provers to prove that sigma protocols are cryptographically valid in this sense, to our knowledge, mixnets never have. This is not only interesting due to being the first machine verified formal proof of the correctness of a verifiable mixnet but also because, as with the sigma protocols, we can extract the verifier and use it to verify real elections. The second result on mixnets builds on the first result on sigma protocols since the mixnet that we prove correct is a protocol built on top of an underlying sigma protocol. The proof of the mixnet uses the fact that the underlying protocol is a verified correct sigma protocol.

3.1 Details on error prevention

To better communicate how our solutions prevents the errors that commonly occur in e-voting, we consider the allegedly end-to-end verifiable Swiss Post e-voting solution we mentioned earlier. This scheme had formal proofs of security, detailed documentation and had undergone various levels of certification. Nevertheless, when the code was made public, various issues were detected. In the next three paragraphs, we outline these issues and explain how our contribution would have eliminated or strongly mitigated these issues.

One of the more trivial errors in the system was in the sigma-protocol for verifying that the voter knows that one of several possible statements is true. The protocol aimed to follow the disjunctive proof approach of Cramer, Damgård, Schoenmakers [20] where the challenger sends one challenge c and the adversary needs to provide n more challenges such that $c = \sum_{i=1}^{n} c_i$. The verifier must check this equality, as otherwise, the soundness of the proof breaks completely. The verifier from the SwissPost e-voting system did not check the equality due to an implementation error, meaning that their verifier was unsound. This mistake could not be made in our formalisation inside Coq, since our encoding demands that every sigma protocol must prove its soundness.

Another mistake was in the use of the Fiat-Shamir transform [23]. This transform converts an interactive sigma protocol into a non-interactive one; however, if not implemented correctly the transform fails catastrophically [11]. In the SwissPost e-voting scheme, the transform was not implemented correctly in producing the publicly verifiable proofs attesting that the ballots were cast-as-intended. This allows a corrupt voting device to substitute ballots without the voter detecting anything. While the Fiat-Shamir transform is out of scope, our explicit formalisation for sigma protocols makes clear what information needs to go into the transform. If the transform is instantiated using the full transcript up to the point of the challenge in our scheme then these issues are avoided.

The most critical error in the SwissPost e-voting scheme was a fault in the commitment parameter generation. The mixnet they were using relies upon a commitment parameter consisting of several group elements for which the discrete log relation should be unknown. In the SwissPost implementation these parameters were generated with a known discrete log relationship. This error allowed any of the several authorities to replace the entire set of ballots with anything they wished, without detection. We intend to model the parameter generation in Coq in future work; however, even as is, our Theorem 7.8 of security for the mixnet makes clear that the parameter generation is of crucial importance by explicitly listing the possibility of incorrect parameter generation as part of the theorem proved. The original paper specification [6] describing the mixnet also demands correct parameter generation, but by moving it to the code level, we remove the gap, which resulted in this critical bug. Additionally, the structure imposed by Coq of clear and simple preconditions where almost all of the detail is proven correct allows a much more focused and less error prone verification process; since, only the definitions need to be scrutinised as encapsulating the correct concept because the theorem prover verifiers the details of the proof.

3.2 Limitations

While our work is a significant step forward, there are several limitations; some of the limitations can be removed, or reduced, through ongoing development, as we discuss in the future work section at the end of the paper; others are inherent to our approach.

At present we are only targeting the integrity of the election process. However, in any real election, one is also concerned about privacy, which we have ignored. While this is a significant limitation of our reasoning about the overall suitability of the electronic voting scheme, it is a secondary consideration since we are primarily interested in extracting a verifier for the correctness of the counting process. An interesting area of future research is to combine our approach with existing security definitions and formally define and prove the security of an e-voting scheme, as well as extracting the verifier. This would change the verifier extracted since the verifier would have to check certain privacy guarantees as well, for instance that no duplicate ballots appear in the ballot box.

In order to eschew probabilistic reasoning, the definitions we use for correctness, in our work, amount to special soundness. Recall that a zero knowledge proof demonstrates that a statement s belongs to a particular language, and it is common to use R to denote the relationship between statements and witnesses. Special soundness says that if any adversary can produce two accepting transcripts for different challenges then it is possible to extract a witness w from those transcripts efficiently such that $(s, w) \in R$. Bellare and Goldreich give the standard definition of proofs of knowledge in their work "On Defining Proofs of Knowledge" [7]. They define knowledge error, which intuitively denotes the probability that the verifier accepts even when the prover does not know a witness. It has been shown that a sigma protocol satisfying special soundness is a proof of knowledge with negligible knowledge error in the length of the challenge, as stated next.

Theorem 3.1. *A sigma protocol $\mathcal{P}$ for relation $\mathcal{R}$ with challenge length t is a proof of knowledge with knowledge error 2^{-t}.*

While we set clear preconditions which imply that the election outcome was correctly calculated and announced, these preconditions also have to be verified. Specifically, we formally prove theorems of the form, if A and B are true then so is C, where C is the integrity of the election. Clearly if A or B are not true then the proof implies nothing about the integrity of the election. In addition, since most e-voting schemes use non-interactive versions of the appropriate zero knowledge proofs, the Fiat-Shamir transform [23] must be applied carefully to avoid the known pitfalls [11].

A final caveat is that the transliteration module inside Coq to turn a Coq function, such as *myplus*, into executable code is not itself verified. In particular, there is no guarantee that the extracted code is correct with respect to the semantics of the chosen programming language. So, strictly speaking we should verify the actual data inside Coq for a higher level of assurance.

4 CONCEPTUAL OVERVIEW

In this section, we describe and motivate some of the conceptual decisions we made in our work. We believe these are the right decisions, at least for many of the current e-voting solutions, and would encourage others to follow them, but they require motivation.

Using an interactive theorem prover only has value if the prover will not accept incorrect proofs; this requires not only the prover itself to be correct but also that simplifying axioms are avoided. In the contexts of cryptography this often makes definitions and theorems unsustainably complex to the point where it is not clear that the proof captures the intent. In addition reasoning about probabilities inside well established theorem provers is complex and further compounds the complexity. For these reasons we choose to avoid any probabilistic statements or simplifying axioms such as perfect encryption or that negligible events never happen.

It may surprise the reader that we can achieve anything interesting under these conditions. However, we achieve cryptographically significant results by proving theorems which are known to imply probabilistic results; for example we can prove that a given three round protocol is perfectly complete, satisfies special-soundness and is honest-verifier zero knowledge—which is to say that it is a sigma protocol—without referring to any probabilities, and the known implication from special-soundness to soundness rests outside what is proved inside the interactive theorem prover.

Some of the verifiable e-voting schemes rely for verification only on sigma protocols and trivial equivalence; so the verifiers can be reasoned about using only special soundness. Of course, any part of the system not formally proved correct must be carefully considered and evaluated; for instance the sigma protocols are almost inevitably going to be made non-interacting via the Fiat-Shamir transform, which has many pitfalls [11]. The conversion using the Fiat-Shamir transform is outside the scope of our work.

5 BUILDING BLOCKS

We will begin to describe what we have defined and proven in Coq. At the start we will express the definition and theorems in standard notation and in Coq notation, for ease of understanding. But as we progress, we shall elide the Coq encodings for lack of space.

5.1 Algebraic Structures

We now describe the definitions and theorems which we encoded and proved in Coq, interspersed with (slightly simplified) examples of the Coq formalisation. We start our building blocks with basic algebraic structures over which we will define our later results. We limit ourselves to working in cyclic groups of prime order since these cover the overwhelming majority of e-voting systems. Our Coq formalisation of an Abelian group is as follows.

Definition 5.1 (Abelian Group). An Abelian Group is a set G, together with a binary operator $\cdot$, identity element e in G, and unary operator $-$, such that:

Associative: $\forall x, y, z \in G, x \cdot (y \cdot z) = (x \cdot y) \cdot z$
Identity: $\forall x \in G, e \cdot x = x$
Inverse: $\forall x \in G, e = -x \cdot x$
Commutative: $\forall a, b \in G, a \cdot b = b \cdot a$.

```
Class AbeGroup (G: Set) (dot: G -> G -> G) (one: G) (inv: G -> G)
:= {
  dot_assoc : forall x y z: G, dot x (dot y z) = dot (dot x y) z;
  one       : forall x: G, dot one x = x;
  inv       : forall x: G, one =  dot (inv x) x;
  comm      : forall a b: G, dot a b = dot b a; }.
```

Here, our definition is encoded as an abstract type class, *AbeGroup*, which takes four parameters: G, *dot*, *one* and *inv*. The type of G is *Set* as required by our definition. The type $G \rightarrow G \rightarrow G$ of *dot* encodes it as a prefix function (rather than an infix operator) which takes two elements from G and returns an element in G. The type of *one* is G itself, encoding that $one \in G$. The type $G \rightarrow G$ of *inv* encodes it is a function which takes an element from G and returns an element in G. The conditions that make G and Abelian group are encoded inside the body of the class. We can now instantiate the class *AbeGroup* by providing the four parameters, together with proofs that the four conditions are met by the chosen parameters. Coq will only allow the chosen instance if all proofs are correct.

A vector space is the primary structure used in our work on Helios and mixnets. In particular, we will normally be interested in the vector space of a cyclic group of prime order over the field of integers modulo the same order. Our Coq formalisation of the vector space follows.

Definition 5.2 (Vector Space). A vector space is a set G with a binary operator $\cdot$, identity element e in G, a unary operator $-_G$, and a set F with two binary operators $+$ and $*$, two identity elements 0 and 1 in F, two unary operators $-_F$, $1/$, and binary (exponentiation) operator $-^{-}$ such that:

Abelian group: $\langle G, \cdot, e, -_G \rangle$ form an Abelian Group
Field: $\langle F, +, *, 0, 1, -_F, 1/ \rangle$ form a field
Distributivity with respect to vector addition:
$\forall r \in F, \forall x, y \in G, (x \cdot y)^r = x^r \cdot y^r$
Distributivity with respect to field addition:
$\forall r, s \in F, \forall x \in G, x^{r+s} = x^r \cdot x^s$
Compatibility: $\forall r, s \in F, \forall x \in G, x^{r*s} = x^{r^s}$
Identity: $\forall x \in G, x^1 = x$
Annihilator: $\forall x \in G, x^0 = 1$

```
Class VectorSpace(G: Set) (Gdot: G -> G -> G) (Gone: G)
 (Ginv: G -> G) (op: G -> F -> G)  (F: Set)
 (Fadd: F -> F -> F) (Fzero: F) (Fsub: F -> F -> F) (Finv: F -> F)
 (Fmul: F -> F -> F) (Fone: F) (FmulInv: F -> F) (Fdiv: F-> F-> F)
:= {
  vs_field :> field_theory Fzero Fone Fadd Fmul Fsub Finv Fdiv
  FmulInv (@eq F);
  vectorspace_abegrp :> AlbGroup G Gdot Gone Ginv;

  mod_dist_Gdot : forall (r : F) (x y : G),
                  op (Gdot x y) r = Gdot (op x r) (op y r);
  mod_dist_Fadd : forall (r s : F) (x : G),
                  op x (Fadd r s) = Gdot (op x r) (op x s);
  mod_dist_FMul : forall (r s: F) (x : G),
                  op x (Fmul r s) = op (op x s) r;
  mod_id        : forall (x : G), op x Fone = x;
  mod_ann       : forall (x : G), op x Fzero = Gone;
}.
```

Here, @eq denotes the Leibniz equality of elements of type F.

Having defined the basic algebraic structures we proceed to define the type of a sigma protocol. In essence it is a collection of sets and algorithms defined over those sets.

Definition 5.3 (Sigma protocol form). The form of a sigma protocol is a collection of the following sets and functions:

- A set S of statements
- A set W of witness
- A function *Rel* defining a relationship between S and W
- A set C of commitments
- A set R of random coins for the prover

- A set E of challenges
- A binary operator + on E
- An element 0 of E
- A unary operator − on E
- A function *disjoint* from two elements of E to a boolean
- A set T of responses
- A function P_0 mapping a statement, random coin and witness into a tuple of a statement and a commitment.
- A function V_0 mapping a tuple of a statement, a commitment and a challenge into a tuple of a statement, commitment and challenge.
- A function P_1 mapping a tuple containing a statement, commitment and challenge, a random coin, and a witness into a tuple containing a statement, commitment, challenge and response.
- A function V_1 which maps a tuple containing a statement, commitment, challenge and response into a boolean.
- A function *simulator* which maps a statement, repose and challenge into a tuple containing a statement, commitment, challenge and response.
- A function *simMap* which maps a statement, random coin, challenge and witness into a response.
- A function *extractor* which maps two response and challenges into a witness.

```
Variable E: Set. (* The set of challenges *)
Record form := mkForm {
   S: Set; W : Set; (* sets of statements and witnesses *)
       (* The relation function and the set of commitments *)
   Rel: S -> W -> bool; C: Set;
       (* The set of random coins for the prover *)
   R: Set; add: E -> E -> E; zero: E; inv : E -> E;
   disjoint: E -> E -> bool; (* required for product groups *)
   T: Set; (* The set of responses *)
      (* The initial step of the prover, outputs a commitment *)
   P0: S -> R -> W -> (S * C);
       (* The initial step of the verifier, outputs a challenge *)
   V0: (S * C) -> E -> (S * C * E);
       (* The final step of the prover, outputs a response *)
   P1: (S * C * E) -> R -> W -> (S * C * E * T);
       (* The final step of the verifier *)
   V1: (S * C * E * T) -> bool;
      (* The simulator *)
   simulator: S -> T -> E -> (S * C * E * T);
      (* An explicit mapping between honest and simulated *)
   simMap: S -> R -> E -> W -> T;
      (* The extractor *)
   extractor: T -> T -> E -> E -> W
}.
```

A Coq *Record* is akin to *Class*, and they can be used interchangeably, but a class supports more automated type inference and can be easily extended by another class. The keyword *Variable* which introduces can abstract variable which is concretely defined later, the pairing operation $S * C$ which is the type consisting of pairs (s, c) where s is of type S and c is of type C, and the type *bool* which encodes that the function *disjoint* returns a Boolean value of true or else false; this is why "= true" turns up later. The set E of challenges is external to the record for technical reasons. We later use combiners which only work if the set of challenges for two sigma protocol are equal; for Coq to type check these combiners it needs to know externally of the record that the challenge set is the same. For this reason you will see $Sigma.form$ E to denote an instantiation of sigma protocol form with the challenge set E.

An object of the type *Sigma protocol form* is a *Sigma protocol* if it satisfies the conditions shown below in Definition 5.4. The core of the requirements are, of course, correctness, special soundness, and honest verifier knowledge. We define honest verifier zero knowledge in a concrete way without referring to probabilities; we show that there exists a bijection between the transcripts generated by taking the random coin from the commit in $P0$ and by taking the response at random in the simulation. In addition we require the challenge space to be an abelian group, the algorithms to output the transcript they receive without change, that algorithm $V0$ outputs the challenge from its randomness tape without modification, and that the simulator produces accepting transcripts on all inputs. The principal advantage of this formalisation is that we define a final verification step which is what we will extract.

Definition 5.4 (Sigma protocol). An object with the form of a sigma protocol is a sigma protocol if it satisfies the following, where we use ⇒ for logical implication since → is used for another concept in the cryptographic community:

- Correctness: $\forall s \in S, w \in W, r \in R, c \in E, Rel(s, w) = true \Rightarrow V_1(P_1(V_0(P_0(s, r, w), c)r, w) = true$
- Special Soundness: $\forall s \in S, c \in C, e_1 e_2 \in E, t_1 t_2 \in T,$ $disjoint(e_1, e_2) = true \Rightarrow V_1(s, c, e_1, t_1) = true \Rightarrow$ $V_1(s, c, e_2, t_2) = true \Rightarrow Rel(s, extractor(t_1, t_2, e_1, e_2)) = true.$
- Honest Verifier Zero Knowledge: $\forall s \in S, w \in W, r \in R, e \in E, Rel(s, w) = true \Rightarrow P_1(V_0(P_0(s, r, w), e), r, w) =$ $simulator(s, simMap(s, r, e, w), e) \wedge \forall t \in T, \exists r \in R$ s.t. $t = simMap(s, r, e, w)$.
- Simulator Correctness: $\forall s \in S, t \in T, e \in E,$ $V_1(Simulator(s, t, e)) = true.$
- The first part of the output of the functions $P_0, V_0, P_1, V_1,$ *simulator* is the first element of their input. That is, if you consider the first part of the input as the protocol transcript, they appended to that transcript without modifying the prefix.
- The function V_0 appends to the protocol transcript the challenge it receives without modification.
- The function *simulator* outputs the challenge it receives without modification.
- The response included in the output of function *simulator* does not depend upon the statement.

```
Class SigmaProtocol (Sig: Sigma.form E) := {
  e_abgrp :> AbeGroup E Sig.add Sig.zero Sig.inv;

  (** The functions do not modify the previous transcript *)
  pres_p0: forall (s: Sig.S) (r: Sig.R) (w: Sig.W),
    (Sig.P0 s r w) = (s,(Sig.P0 s r w).2);
  pres_v0: forall (sc : Sig.S*Sig.C)(e : E),
    (Sig.V0 sc e) = (sc,(Sig.V0 sc e).2);
  pres_p1: forall (sce: Sig.S*Sig.C*E) (r: Sig.R) (w: Sig.W),
    (Sig.P1 sce r w) = (sce,(Sig.P1 sce r w).2);
  pres_sim: forall (s: Sig.S)(t : Sig.T)(e : E),
      (s, (Sig.simulator s t e).1.1.2) = (Sig.simulator s t e).1.1;

  (** For composability V0 maps E to E independently of S*C *)
   comp_v0: forall (sc: Sig.S*Sig.C) (e: E), e = (Sig.V0 (sc) e).2;
  (** The simulator's challenge and response are independent
    of the statement *)
  chal_sim: forall (s: Sig.S)(t : Sig.T)(e : E),
    e = (Sig.simulator s t e).1.2;
  comp_sim1: forall (s1 s2 : Sig.S)(t :Sig.T)(e: E),
          (Sig.simulator s1 t e).2 = (Sig.simulator s2 t e).2;

  (** Properties *)
  correctness: forall (s :Sig.S) (w: Sig.W) (r: Sig.R) (c: E),
    Sig.Rel s w ->
    Sig.V1 (Sig.P1 (Sig.V0 (Sig.P0 s r w) c) r w) = true;

  special_soundness: forall (s: Sig.S) (c: Sig.C) (e1 e2: E)
    (t1 t2: Sig.T),
    Sig.disjoint e1 e2 ->
    Sig.V1 (s, c, e1, t1) = true ->
    Sig.V1 (s, c, e2, t2) = true ->
    Sig.Rel s (Sig.extractor t1 t2 e1 e2) = true;

  honest_verifier_ZK:
    forall (s: Sig.S) (w: Sig.W) (r: Sig.R) (e: E),
      Sig.Rel s w = true ->
      (Sig.P1(Sig.V0 (Sig.P0 s r w) e) r w) =
      Sig.simulator s (Sig.simMap s r e w) e  /\
      forall (t: Sig.T),
       exists r: Sig.R, t = (Sig.simMap s r e w);

  simulator_correct: forall (s: Sig.S) (t: Sig.T) (e: E),
    Sig.V1(Sig.simulator s t e) = true;
```

Given a tuple $a = (A, B)$, the notation $a.1$ and $a.2$ respectively return A as the first and B as the second element of a.

To allow some of the combiners we need, we define a stronger variant called a composable sigma protocol. In essence a composable sigma protocol is a sigma protocol where the statement does not effect the response returned, this is true of most sigma protocols but notably, not for those over disjunctive statements.

Definition 5.5 (Composable Sigma Protocol). An object with the form of a sigma protocol is a composable sigma protocol if the following conditions are satisfied:

- The object is a sigma protocol
- Compose P_1: $\forall s_1 s_2 \in S, c_1 c_2 \in C, e \in E, r \in R, w \in W, Rel(s_1, w) \wedge Rel(s_2, w) \Rightarrow$ $P1(((s_1, c_1), e), r, w).2 = P1(((s_2, c_2), e), r, w).2$.
- Compose *simMap*: $\forall s_1 s_2 \in S, r \in R, e \in E, w \in W$, $simMap(s_1, r, e, w) = simMap(s_2, r, e, w)$.

On top of this formalisation of sigma-protocols, we define, encode and formally verify various combiners, as explained next. To save space, their Coq encodings are in Appendix C. Given a sigma protocol for relation R, we can define sigma protocols for the following relationships. The *equality* relation R' such that $((s_1, s_2), w)$ are in R' iff $(s_1, w) \in R \wedge (s_2, w) \in R$, where the prover shows they have a witness that simultaneously satisfies two statements.

Theorem 5.6. *If a sigma protocol S is composable then so is the sigma protocol produced by running eqSigmaProtocol on S.*

```
Theorem eqCorr :
   CompSigmaProtocol E sigma ->
     CompSigmaProtocol E (eqSigmaProtocol E sigma).
```

The *and* relation R' such that $((s_1, s_2), (w_1, w_2))$ are in R' iff $(s_1, w_1) \in R \wedge (s_2, w_2) \in R$, where the prover shows they know two witnesses that satisfy two statements.

Theorem 5.7. *If S is a sigma protocol then so is the result of running andSigmaProtocol on S.*

The *disjunctive* relation R' such that (s_1, s_2, w) are in R' iff $(s_1, w) \in R \vee R(s_2, w) \in R$, where the prover shows they have know a witness that satisfies one of two statements.

Theorem 5.8. *If S is a sigma protocol then running disSigmaProtocol on S produces a sigma protocol, provided that the disjoint function of S is equivalent to the negation of the equal function.*

Given two sigma protocols for relationship R and R', respectively, can we define a new sigma protocol for the following relationship? The *parallel* relation R'' such that $((s_1, s_2), (w_1, w_2))$ are in R'' iff $(s_1, w_1) \in R \wedge (s_2, w_2) \in R$'. That is, where the prover shows they know two witnesses that satisfy the two statements.

Theorem 5.9. *If S and S′ are sigma protocols then running parSigmaProtocol on S and S′ produces a sigma protocol.*

Given two sigma protocols, one for the relationship R and one for the relationship R', where both sigma protocols have the same challenge set, can we define a new sigma protocol for the following relationship. The *and* relation R' such that $((s_1, s_2), (w_1, w_2))$ are in R' iff $(s_1, w_1) \in R \wedge (s_2, w_2) \in R$, where the prover shows they have know two witness that satisfy two statements.

Theorem 5.10. *If S and S′ are sigma protocols with the same challenge space then running genAndSigmaProtocol on S and S′ produces a sigma protocol.*

5.2 Concrete Sigma Protocols

Many of the most common sigma protocols used in e-voting are some form of proof knowledge of discrete log.

Definition 5.11. The discrete log (dLog) is a relationship between two group elements and one field element where the second group element is equal to the first to the power of the field element.

```
Definition dLog (s: G*G)(w: F) :=
  let g     := s.1 in
  let gtox  := s.2 in
  gtox = (g^w).
```

We then define the component algorithms for the sigma protocol for knowledge of discrete log.

```
Definition valid_P0 (ggtox: G*G) (r: F) (w: F): (G*G*G) :=
  let g := ggtox.1 in
  (ggtox, g^r).

Definition valid_V0 (ggtoxgtor: G*G*G) (c: F): (G*G*G*F)
  := (ggtoxgtor, c).

Definition valid_P1 (ggtoxgtorc: G*G*G*F) (r x: F) : G*G*G*F*F :=
  let c    :=  snd (ggtoxgtorc) in
  let s    := (r + c*x) in (ggtoxgtorc, s).

Definition valid_V1 (ggtoxgtorcs : G*G*G*F*F) :=
  let g    :=  fst (fst (fst (fst ggtoxgtorcs))) in
  let gtox :=  snd (fst (fst (fst ggtoxgtorcs))) in
  let gtor :=  snd (fst (fst ggtoxgtorcs)) in
  let c    :=  snd (fst ggtoxgtorcs) in
  let s    :=  snd ggtoxgtorcs in
  (g^s) = ((gtox^c) o gtor).

Definition simulator_mapper (s: G*G) (r c x: F):=
  (r+x*c).

Definition simulator (ggtox: G*G) (z e: F) :=
let g    := fst ggtox in
let gtox := snd ggtox in
  (ggtox, g^(z) o \ gtox^e, e, z).

Definition extractor (s1 s2 c1 c2: F) :=
  (s1 - s2) / (c1 - c2).

Definition disjoint (c1 c2: F) :=
  neg (c1 = c2).
```

We use ∘ for the group operation and \ for the group inverse.

We then use these algorithms to define the sigma protocol form.

```
Definition dLogForm : Sigma.form F := Sigma.mkForm F (prod G G) F
 dLog G F Fadd Fzero Finv Fbool_eq disjoint F valid_P0 valid_V0
 valid_P1 valid_V1 simulator simulator_mapper extractor.
```

We then prove that the sigma protocol form is a sigma protocol.

```
Theorem dLogSigma: CompSigmaProtocol F dLogForm.
```

We can now take advantage of the combiners to generate more complicated protocols. First, we apply the the equality combiner to construct a sigma protocol for equality of discrete logs. This sigma protocol is used to prove that a ciphertext decrypts to a particular value, as we will show below. We call this the DHTForm because it is used to prove that four groups elements are a Diffie-Hellman Tuple that is of the form (g, g^a, g^b, g^{ab}).

```
Definition DHTForm: Sigma.form F := eqSigmaProtocol F dLogForm.
```

To prove that one of two tuples is a Diffie-Hellman Tuple, we form the disjunction of DHTForm, and use it to prove that an ElGamal ciphertext is the encryption of one of two messages.

```
Definition DHTDisForm: Sigma.form F := disSigmaProtocol F DHTForm.
```

6 A PROVABLY CORRECT VERIFIER FOR HELIOS

Having encoded the basic algebraic structures and the definitions and combiners for sigma protocols, we proceed to instantiate them to Helios, as used in the 2018 IACR directors election. To do so, we must instantiate our basic algebraic definitions for the specific Schnorr group used in the election. Recall that a Schnorr group is a multiplicative Abelian subgroup of prime order q of the field of integers modulo a prime p, with $p = kq+1$ for some k. To instantiate this group we must first prove that p and q are prime.

Interactive theorem provers are designed for proving mathematical statements, but not for running computations inside their environment. Consequently, computationally intensive proofs of mathematical statements, such as number theoretic proofs, are not ideal for interactive theorem provers. However, recent advances in interactive theorem provers (specifically Coq) allowed us to prove primality of two large prime numbers inside Coq. To begin with we utilise the CoqPrime library[3] to prove in Coq that the numbers p and q used by ICAR to define the Schnorr group (as described above) are in fact prime.

```
Definition P : Z := 16328632084933010002384055033805457329601614771
18595538973916730908621480040646579903858363495375294167564556218249
81207502649804923813755793676756487712938003103709647457670142436385
18442553823973482995267304044326777047662957480269391322789378384
619428596446446984694306187644767462460965622580087564339212631775817
8959584090166763989756712661796378985576873170761772188432331506951
5788106125705301913307854592898356222139631316962247550981844266104
70184362648069010239662367183672047107559358990137503061077380023
64137917426595737403871114187750804346564731250609196846638183903982
3878845782661365036974934746820 71.
Lemma P_prime : prime P.
```

```
Definition Q : Z := 61329566248342901292543872769978950870633559608
669337131139375508370458778917.
Lemma Q_prime : prime Q.
```

We can now make use of the modular arithmetic components of the CoqPrime library to define the field of integers modulo Q.

```
Definition F       : Set := (znz Q).
Definition Fadd    : R -> R -> R  := (add _).
Definition Fzero   : R := (zero _).
Definition Fbool_eq (a b :R) : bool := Z.eqb (val Q a) (val Q b).
Definition Fsub    : R -> R -> R := (sub _).
Definition Finv    : R -> R := (opp _).
Definition Fmul    : R -> R -> R := (mul _).
Definition Fone    : R := (one _).
Definition FmulInv : R -> R := (inv _).
Definition Fdiv    : R-> R-> R := (div _).
Lemma Ffield : field_theory Fzero Fone Fadd Fmul Fsub Finv Fdiv
    FmulInv (@eq F).
```

We now define the function to check membership in the subgroup of order Q of the integer modulo P.

```
Definition inQSubGroup (n: Fp): Prop
:= binary_power n (Z.to_nat Q) = one P.
```

With this function we can define the group that we will work in.

```
Definition G: Set := { Fp | inQSubGroup Fp }.
```

A subset type, represented as {x : A | P x}, denotes the subset of elements of the type A which satisfy the predicate P. Subset types are ideal for the situation where we need some kind of restriction on data, e.g. in an integer division function, we want the divisor to be non-zero. This restriction can easily be represent using the subset type {`div : Z | div ≠ 0`}.

Finally, we can define the vector space.

```
Instance HeliosIACR2018: VectorSpace F Fadd Fzero Fbool_eq Fsub
  Finv Fmul Fone FmulInv Fdiv G Gdot Gone Gbool_eq Ginv op := {}.
```

`Instance` introduces an instance of a class, in this case VectorSpace. The values immediately following the keyword `VectorSpace` are parameters and the contents of ":= { }" are the methods, which in this case is empty. We now define ElGamal encryption [25] in Coq. Given a generator g, public key h, randomness r and message m return $(g^r, h^r m)$.

```
Definition enc (g h: G) (r: F) (m: G): (G*G) := (g^r,h^r o m).
```

An ElGamal ciphertext c decrypts to the message m if there exists r such that $c = (g^r, h^r \ o \ m)$.

```
Definition decryptsTo (g h: G) (c: (G*G)) (m: G)
:= exists (r: F), enc g h r m = c.
```

Our ElGamal ciphertext c must decrypt to one or zero where zero is the identity element in the group and one is the generator.

```
Definition decryptsToOneOrZero (g h: G) (c: (G*G)): Prop  :=
  let zero := Gone in
  let one  := g in
  decryptsTo g h c zero \/ decryptsTo g h c one.
```

A list cs of ElGamal ciphertexts is correctly formed for a Helios system using approval voting if all ciphertexts decrypt to one or zero.

```
Definition HeliosCorrectEncrApproval  (g h: G) (cs: list (G*G)) :=
    let zero := Gone in
    let one  := g in
    Forall (decryptsToOneOrZero g h) cs.
```

To generalise our approach we define a function ApprovalSigma which given a list of ElGamal ciphertexts produces a sigma protocol to check that the ciphertexts are correctly encrypted. Similarly, we define a function ApprovalSigmaStatement which given a list of ElGamal ciphertext converts them into the statement of the sigma protocol produced by ApprovalSigma.

We will now proceed to define two theorems "HeliosCorrectEncrApprovalList" and "HeliosCorrectDecrList", which together imply that all the ballots are correctly formed and that the announced result is the decryption of the summation of the ballots.

[3] https://github.com/thery/coqprime

Theorem 6.1 (HeliosCorrectEncrApprovalList). *For any generator g and public key h and for any list of voters voting on any number of options, if you generate the statement and sigma protocol using the functions ApprovalSigmaStatement and ApprovalSigma and the verifier accepts two transcripts with different challenges then we extract a witness which shows that the ciphertexts are correctly encrypted, as defined in HeliosCorrectEncrApproval.*

As we have already noted, this style of theorem combined with the known relationship between special soundness and soundness implies that the verifier only accepts with negligible probability unless the statement is true.

```
Definition HeliosCorrectEncrApprovalList (g h: G)
        (cs: list (list (G*G))): Prop :=
forall x: list(G*G),
  In x cs ->

  let Sig := ApprovalSigma x in
  let   s := (ApprovalSigmaStatement g h x) in

  forall (c: Sigma.C (recChalType x) Sig) (e1 e2: (recChalType x))
         (t1 t2: Sigma.T (recChalType x) Sig),
  Sigma.V1 (recChalType x) Sig (s,c,e1,t1) ->
  Sigma.V1 (recChalType x) Sig (s,c,e2,t2) ->
  Sigma.disjoint (recChalType x) Sig e1 e2 ->
  HeliosCorrectEncrApproval g h x.
```

We define `decryptsTo2` as the decryption of an ElGamal ciphertext. The implication is the same, however the evidence provided differs. In the earlier definition the proof consisted of the randomness used in encryption whereas here it consists of the secret key.

```
Definition decryptsTo2 (g h: G) (c: (G*G)) (m: G) :=
  exists (x: F), g^x = h /\ (c.2 o - m) = c.1^x.
```

We now define decryptionConsistent which given a generator g, public key h and pair consisting of a ciphertext and message, returns true if the message is encrypted in the ciphertext.

```
Definition decryptionConsistent (g h: G) (pair: (G*G)*G) :=
  decryptsTo2 g h pair.1 pair.2.
```

We then define DecryptionSigma, which generates the sigma protocol for correct decryption, as a parallel composition of the Diffie Hellman tuple sigma protocol.

```
Definition DecryptionSigma :=
  parSigmaProtocol (F*F*F) F (parSigmaProtocol (F*F) F
    (parSigmaProtocol F F DHTForm DHTForm) DHTForm) DHTForm.
```

decFactorStatement takes a generator g, public key h, ciphertext c, and decryption factor d and returns a statement for the Diffie Hellman tuple sigma protocol.

```
Definition decFactorStatement (g h: G)(c: G*G)(d: G): DHTForm.S
:= ((g,h),(c.1,d)).
```

We then define DecryptionSigmaStatement which produces statements for the sigma protocol produced by DecryptionSigma.

```
Definition DecryptionSigmaStatement (g: G) (c: G*G)
  (y: (G*G*G*G)) (d: (G*G*G*G)): DecryptionSigma.S :=
  let '(y1,y2,y3,y4) := y in
  let '(d1,d2,d3,d4) := d in
  ((decFactorStatment g y1 c d1),(decFactorStatment g y2 c d2),
    (decFactorStatment g y3 c d3),(decFactorStatment g y4 c d4)).
```

We now define a theorem, which we prove later, which says the verifier of the sigma protocol accepting on the output of the statement generator implies with overwhelming probability that the result was correctly announced. For simplicity, we present this definition for the concrete case of 4 authorities though this could be easily generalised. The theorem says that for any generator g, public key h, for authority subkeys, for all decryption factors and ciphertexts and claimed result, if the verifier accepts for two different challenges we can extract a witness for the truth of the statement.

```
Definition HeliosCorrectDecrList  (g h: G)(y: (G*G*G*G))
  (df: list(G*G*G*G))
  (dt: list ((G*G)*G)) : Prop :=

  let d := combine dt df in

  forall x: ((G*G)*G*(G*G*G*G)),
  In x d -> (*For all values to check the decryption of *)

   let '(c1, c2) := x.1.1 in
   let         m := x.1.2 in
   let '(df1, df2, df3, df4) := x.2 in

   (*Basic consistence, the sum of decryption factors is equal to
 second element of the ciphertext divided by the claimed message *)
  df1 o df2 o df3 o df4 = c2 o - m ->

  let Sig := DecryptionSigma in
  let s   := DecryptionSigmaStatement g x.1.1 y x.2 in

  forall (c: Sigma.C (F*F*F*F) Sig) (e1 e2: (F*F*F*F))
       (t1 t2: Sigma.T (F*F*F*F) Sig),
  Sigma.V1 (F*F*F*F) Sig (s,c,e1,t1) ->
  Sigma.V1 (F*F*F*F) Sig (s,c,e2,t2) ->
  Sigma.disjoint (F*F*F*F) Sig e1 e2 ->
  decryptionConsistent g h x.1.
```

We prove the two theorems defined above.

```
Theorem HeliosCorrectResultApproval :
  (*The outer list contains each voter and the inner list
       contains their ciphertext on each option *)
  forall numOpts numVoter: nat,
  forall (g h: G)(cs: list (list (G*G))),
  forall (y: (G*G*G*G)),(*Authoritity keys *)
  (* Decryption Factors, each authority each option *)
  forall (df: list (G*G*G*G)),
  forall (r: list F), (* Results *)

  length cs = numVoter -> (*Basic data length consistency *)
  Forall (lenEqualToN numOpts) cs ->
  length df = numOpts ->
  length r = numOpts ->

  (*Data consistence*)
  y.1.1.1 o y.1.1.2 o y.1.2 o y.2 = h ->
   (* The authority decryption keys combine to give the h *)

  let summation     := map Prod cs in
  let resultInGroup := map (mapToGroup g) r in
  let dt            := combine summation resultInGroup in

  (*All ballots are correctly formed AND
    the announced result is correct *)
  HeliosCorrectEncrApprovalList g h cs /\
    HeliosCorrectDecrList g h y df dt.
Proof.
```

6.1 Verifying a real election

We used the well developed program extraction [37] facility of Coq to extract the definitions above into OCaml code. It can, also, be used to extract proofs into Haskell or Scheme programs. Specifically we extracted the `DecryptionSigma`, `DecryptionSigmaStatement`, `ApprovalSigma`, and `ApprovalSigmaStatement`.

We then retrived the votes, results and proof from the Helios IACR 2018 directors election from the Helios website, the election address can be found here [4]—you can find the data combined into .json files in our code repository, see appendix A. The main verification code lies in main.ml, it is very simple, it parses the JSON files and feeds the data into the sigma protocols extracted from Coq. The

[4] https://vote.heliosvoting.org/helios/elections/60a714ea-ce6d-11e8-8248-76b4ab96574c/view

election had four authorities, seven candidates (which the voter could select any subset of), and 465 cast ballots. The election takes two minutes to verify on a single core of a MacBook Pro (2.3Ghz), the peak memory usage is 53MB. We note that the verification work can be parallelized trivially.

7 APPLICATIONS TO MIXNETS

Verifiable e-voting schemes which use homomorphic tallying, such as Helios, rely only on sigma protocols for verifiability, but many verifiable e-voting schemes use mixnets instead. Mixnets were first introduced by Chaum [15] as a solution to the traffic analysis problem. In the context of e-voting they are commonly used to provide anonymity. The process is normally, that encrypted votes appear next to the voter's identity on the first bulletin board. These ballots are then re-encrypted and shuffled several times before being decrypted. Clearly, if the mixnet doing the shuffle is not verifiable this would allow ballot substitution. To prevent this, e-voting has long made use of verifiable mixnets first introduced by Neff [39] and Furukawa et al [24]. Verifiable mixnets use more complicated zero knowledge protocols called proofs of shuffle to prove that the output of the mixnet is a permuted re-encryption of the input.

Wikström's proof of shuffle [47] and follow up work with Terelius [46] is one of the most common proofs of shuffle. Wikström's result applies to many cryptosystems so we take the optimised variant for ElGamal [45] which is widely deployed in practice [13, 44, 48]; it has been used in binding government elections in Norway, Spain and Estonia. We demonstrate that it is feasible to formally prove the correctness of verifiable mixnets inside an interactive theorem prover and then extract the verifier.

We first define the statements which the mixnet proves. The mixnet shows that the ballots were permuted using a permutation previously committed to by the prover. Since the commitment scheme used is perfectly hiding, but only computational binding, it is always possible that the commitments were broken by the adversary rather than the mix being correct. We define how the Pedersen commitment works and what it means for the binding property of the commitment to be broken in the next four definitions.

Definition 7.1 (Petersen Commitment). A Pedersen commitment for two group elements $h, h_1 \in G$ and message $m \in F$ and randomness $r \in F$ is $c \in G = h^r h_1^m$.

```
Definition PC (h: G) (h1: G) (m : F) (r: F): G
:= h^r o h1^m.
```

We enforce that $c \in G$ by setting the return type of *PC* to be G.

Definition 7.2 (Extended Petersen Commitment). An extended Pedersen commitment for a group element $h \in G$, and a vector of group elements $(h_1, ..., h_n)$ and vector of messages $(m_1, ..., m_n)$ and randomness $r \in F$ is $c \in G = h^r \sum_{i=1}^{n} h_i^{m_i}$.

```
Definition EPC (h: G) (hs: V2G) (m: V2F) (r: F): G
:=  h^r o V2G_prod (V2M_Pexp hs m).
```

Definition 7.3 (Binding Pedersen Commitment Relation). Given two group elements $h, h_1 \in G$ we say that the binding property is broken if an adversary can find a commit $c \in G$ and two openings $(m_1, r_1), (m_2, r_2) \in F$ such that $m_1 \neq m_2$ and $c = h^{r_1} h_1^{m_1} = h^{r_2} h_1^{m_2}$.

We recall that the binding property reduces to the discrete log problem in the underlying group. For this problem to be hard, g and h should be random generators chosen in a way such that no non-trivial information is leaked about the relationship between them. However, as we have already mentioned, correct parameter generation is currently out of scope for our work.

```
Definition relComPC (h: G) (h1: G) (c: G) (* Statement *)
              (m1 m2: F) (r1 r2: F) :=    (* Witness   *)
  m1 <> m2 /\
  c = (PC h h1 m1 r1) /\ c = (PC h h1 m2 r2).
```

Definition 7.4 (Binding Extended Pedersen Commitment). Given a group elements $h \in G$ and a vector of group elements $(h_1, ..., h_n) \in G$ we say that the binding property is broken if an adversary can find a commit $c \in G$ and two openings $((m_1, ..., m_n), r) \in F$ and $((m'_1, ..., m'_n), r') \in F$ such that $m_i \neq m'_i$ for some i and and $c = h^r \sum_{i=1}^{n} h_i^{m_i} = h^{r'} \sum_{i=1}^{n} h_i^{m'_i}$.

```
Definition relComEPC (h: G) (hs: V2G) (c: G) (*Statement*)
              (m1 m2: V2F) (r1 r2: F)         (*Witness  *)
:=  m1 <> m2 /\
    c = (EPC h hs m1 r1) /\ c = (EPC h hs m2 r2).
```

If the commitments were not broken then the shuffle must be correct. The correctness of the shuffle is captured by the three following definitions.

First we describe what it means to commit to a permutation.

Definition 7.5 (Matrix Commitment). Given a group element $h \in G$, a vector of group elements $(h_1, ..., h_n) \in G$, a square matrix $m \in F^{n*n}$, and a vector of field elements $(r_1, ..., r_n) \in F$, the matrix commitment is vector of group elements $(c_1, ..., c_n)$ where $c_i = h^{r_i} \sum_{j=1}^{n} h_j^{m_{i,j}}$.

Definition 7.6 (Permutation commitment relation). Given a group element h, a vector of group elements $(h_1, ..., h_n) \in G$, a commitment c, a matrix m and vector of field elements $(r_1, ..., r_n)$, we say they satisfy the permutation commitment relation if the matrix m is a permutation matrix and c is a *matrix commitment* to m using randomness r.

Definition 7.7 (Shuffle and re-encryption relation). Given a group element g, public key pk, vectors of ElGamal ciphertexts $(e_1, ..., e_n)$ and $(e'_1, ..., e'_n)$, matrix m, and vector of field elements $(r_1, ..., r_n)$, we say they satisfy the shuffle and re-encryption relation if the output ciphertext vector is the re-encryption of the vector of input ciphertexts multiplied by m using the randomness.

We begin by defining the underlying sigma protocol which underpins Wikström's mixnet in the case of ElGamal encryption. We have moved the definition of u'Form to the appendix since the fitness of the sigma protocol for its purpose is checked by Coq.

```
Definition WikstromSigma :=
  parSigmaProtocol (parSigmaProtocol dLogForm dLogForm)
    (u'Form).
```

We now have all material to define our concluding theorem. The theorem has a fair bit of detail but in essence it says that if the verifier from sigma protocol *WikstromSigma* accepts transcripts containing statements of certain forms then either the commitments are broken or the shuffle was performed correctly. This, coupled with the known relation between soundness and special soundness, implies that checking the verifier suffices for ensuring correct mixing, if the commitments parameters are generated correctly.

Theorem 7.8 (Special Soundness of Mixnet). *For all group elements g, public keys pk, two vectors of ElGamal ciphertexts $(e_1, ..., e_n)$ and $(e'_1, ..., e'_n)$, group elements h, a vector of group elements $(h_1, ..., h_n) \in G$, matrix commitments c and primary challenge matrices U if the determinant of U is non-zero it implies that:*

For all vector of group elements $\hat{c}_1, \hat{c}_2$ if you take the Wikström-sigma protocol and statements s_1 and s_2 generated by calling WikstromStatement on the above, if the adversary can create $2n$ accepting transcripts than we efficiently compute either a witness that breaks the binding property of the commitments or a witness that the shuffle was correctly done.

We will first explain why the assumptions in the theorem are acceptable and then briefly describe the proof.

The first assumption is that the primary challenge matrix U has a non-zero determinant. This is fine because the challenge matrix is chosen by the verifer at random and a random matrix has a non-zero determinant with overwhelming probability. The second set of assumptions is that we can get $2n$ accepting transcripts of the correct form. This assumption is what makes the theorem a type of special soundness. We remind the reader that special soundness implies soundness and hence proving this theorem implies producing a single accepting transcript is computationally unfeasible unless the statement is true. The Schwartz-Zippel lemma assumptions assume we are in the overwhelming probable case where polynomial tested using the Schwartz-Zippel is zero. In particular, the proof relies on a theorem which states that a square matrix is a permutation matrix if and only if it satisfies two equations. One of the equations is defined over the ring of polynomials rather than the field. The Schwartz-Zippel lemma is used to efficiently prove that the polynomial equalities hold with overwhelming probability.

By assuming that the events which occur with only negligible probability to do not happen, our proof is simplified to linear algebra. The flow of the proof is the same as in the original papers [46, 47], and in particular to the verbose proof of the optimised variant in [32]. The structure of the proof is in two stages. First, it proves that the accepting transcripts allow it to extract witness satisfying some sub-statements; this follows nearly immediately from the special soundness of the underlying sigma protocol. Then it proves given witnesses to these sub-statements it can produce a witness for validity of theorem 7.8.

8 CONCLUSION AND FURTHER WORK

We have created the first provably correct verifier for a real e-voting scheme, along with the technical machinery required to easily produce this kind of verifier for a wide class of e-voting schemes. Our verifier is able to efficiently verify a real election.

We have also proved a mixnet secure in an interactive theorem prover, which had not previously been done. As with the sigma protocols, we can extract the verifier and use this in real elections.

8.1 Future Work

We intend to model the Fiat-Shamir transform and commitment parameter generation.

It would be ideal to unify our work with the existing research on e-voting security definitions; but, this is highly non-trivial. It remains an open problem to encode the existing definitions into Coq, prove that specific voting schemes meet these definitions and then extract the code that actually implements these schemes (as well as their verifiers). This would naturally extend our work to cover the area of privacy as well as integrity.

We have demonstrated techniques that enable proving and extraction of mixnets in Coq. To make this viable for real elections, we need to generalise the proofs, since at present, the mixnet only allows the shuffling of a fixed number of ballots. We stress that this work is not technically complicated but time consuming.

Another gap is that the extraction mechanism of Coq does not come with formal correctness guarantees that reach down to the machine code level, such as, for example, in CakeML [35].

As some readers may have noticed, our Coq formalisation—while fit for purpose—could benefit from further refinement. We intend to do this and provide better tooling to allow other interested parties to more easily adapt our work to their elections.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments. Thomas Haines acknowledges the support of the Luxembourg National Research Fund (FNR) and the Research Council of Norway for the joint project SURCVS.

REFERENCES

[1] Ben Adida. 2008. Helios: Web-based Open-Audit Voting. In *USENIX Security Symposium*, Paul C. van Oorschot (Ed.). USENIX Association, 335–348.

[2] José Bacelar Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. 2010. A Certifying Compiler for Zero-Knowledge Proofs of Knowledge Based on Sigma-Protocols. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*. 151–167. https://doi.org/10.1007/978-3-642-15497-3_10

[3] José Bacelar Almeida, Manuel Barbosa, Endre Bangerter, Gilles Barthe, Stephan Krenn, and Santiago Zanella Béguelin. 2012. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. 488–500. https://doi.org/10.1145/2382196.2382249

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Fast Software Encryption*, Thomas Peyrin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 163–184.

[5] Gilles Barthe, Daniel Hedin, Santiago Zanella Béguelin, Benjamin Grégoire, and Sylvain Heraud. 2010. A Machine-Checked Formalization of Sigma-Protocols. In *CSF*. IEEE Computer Society, 246–260.

[6] Stephanie Bayer and Jens Groth. 2012. Efficient Zero-Knowledge Argument for Correctness of a Shuffle. In *EUROCRYPT (Lecture Notes in Computer Science)*, Vol. 7237. Springer, 263–280.

[7] Mihir Bellare and Oded Goldreich. 1992. On Defining Proofs of Knowledge. In *CRYPTO (Lecture Notes in Computer Science)*, Vol. 740. Springer, 390–420.

[8] Josh Benaloh. 2007. Ballot Casting Assurance via Voter-Initiated Poll Station Auditing. In *EVT*. USENIX Association.

[9] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 207–221. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer

[10] David Bernhard, Véronique Cortier, Olivier Pereira, Ben Smyth, and Bogdan Warinschi. 2011. Adapting Helios for Provable Ballot Privacy. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings (Lecture Notes in Computer Science)*, Vijay Atluri and Claudia Díaz (Eds.), Vol. 6879. Springer, 335–354. https://doi.org/10.1007\/978-3-642-23822-2_19

[11] David Bernhard, Olivier Pereira, and Bogdan Warinschi. 2012. How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios. In

ASIACRYPT (Lecture Notes in Computer Science), Vol. 7658. Springer, 626–643.

[12] Yves Bertot, Pierre Castéran, Gérard Huet, and Christine Paulin-Mohring. 2004. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions.* Springer.

[13] BFH-EVG. 2018. UniCrypt. https://github.com/bfh-evg/unicrypt. (2018).

[14] Alessandro Bruni, Eva Drewsen, and Carsten Schürmann. 2017. Towards a Mechanized Proof of Selene Receipt-Freeness and Vote-Privacy. In *E-VOTE-ID (Lecture Notes in Computer Science)*, Vol. 10615. Springer, 110–126.

[15] David Chaum. 1981. Untraceable mail, return addresses and digital pseudonyms. *Commun. ACM* 24(2) (1981), 84–88.

[16] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. 2014. Verifying Curve25519 Software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 299–309. https://doi.org/10.1145/2660267.2660370

[17] Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, and Bogdan Warinschi. 2018. Machine-Checked Proofs for Electronic Voting: Privacy and Verifiability for Belenios. In *CSF*. IEEE Computer Society, 298–312.

[18] Véronique Cortier and Ben Smyth. 2013. Attacking and fixing Helios: An analysis of ballot secrecy. *Journal of Computer Security* 21, 1 (2013), 89–148.

[19] Ronald Cramer. 1997. Modular Design of Secure yet Practical Cryptographic Protocols.

[20] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. 1994. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. In *CRYPTO (Lecture Notes in Computer Science)*, Vol. 839. Springer, 174–187.

[21] Stéphanie Delaune, Mark Ryan, and Ben Smyth. 2008. Automatic Verification of Privacy Properties in the Applied pi Calculus. In *IFIPTM (IFIP Advances in Information and Communication Technology)*, Vol. 263. Springer, 263–278.

[22] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/SP.2019.00005

[23] Amos Fiat and Adi Shamir. 1986. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO (Lecture Notes in Computer Science)*, Vol. 263. Springer, 186–194.

[24] Jun Furukawa and Kazue Sako. 2001. An Efficient Scheme for Proving a Shuffle. In *CRYPTO (Lecture Notes in Computer Science)*, Vol. 2139. Springer, 368–387.

[25] Taher El Gamal. 1984. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *CRYPTO (Lecture Notes in Computer Science)*, Vol. 196. Springer, 10–18.

[26] Taher El Gamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory* 31, 4 (1985), 469–472.

[27] Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. 2002. A Constructive Proof of the Fundamental Theorem of Algebra Without Using the Rationals. In *Selected Papers from the International Workshop on Types for Proofs and Programs (TYPES '00)*. Springer-Verlag, Berlin, Heidelberg, 96–111. http://dl.acm.org/citation.cfm?id=646540.696038

[28] Milad K. Ghale, Rajeev Goré, and Dirk Pattinson. 2017. A Formally Verified Single Transferable Voting Scheme with Fractional Values. In *E-VOTE-ID (Lecture Notes in Computer Science)*, Vol. 10615. Springer, 163–182.

[29] Milad K. Ghale, Rajeev Goré, Dirk Pattinson, and Mukesh Tiwari. 2018. Modular Formalisation and Verification of STV Algorithms. In *E-Vote-ID (Lecture Notes in Computer Science)*, Vol. 11143. Springer, 51–66.

[30] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract). In *STOC*. ACM, 291–304.

[31] Georges Gonthier. 2008. The Four Colour Theorem: Engineering of a Formal Proof. In *Computer Mathematics*, Deepak Kapur (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 333–333.

[32] Thomas Haines. 2019. A Description and Proof of a Generalised and Optimised Variant of Wikström's Mixnet. *CoRR* abs/1901.08371 (2019).

[33] J Alex Halderman and Vanessa Teague. 2015. The new south wales ivote system: Security failures and verification flaws in a live online election. In *International Conference on E-Voting and Identity*. Springer, 35–53.

[34] Wojciech Jamroga, Michal Knapik, and Damian Kurpiewski. 2018. Model Checking the SELENE E-Voting Protocol in Multi-agent Logics. In *E-Vote-ID (Lecture Notes in Computer Science)*, Vol. 11143. Springer, 100–116.

[35] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Proc. POPL 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192.

[36] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. 2012. Clash Attacks on the Verifiability of E-Voting Systems. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 395–409.

[37] Pierre Letouzey. 2003. A New Extraction for Coq. In *Proc. TYPES 2002 (Lecture Notes in Computer Science)*, Herman Geuvers and Freek Wiedijk (Eds.), Vol. 2646. Springer, 200–219.

[38] Lyria Bennett Moses, Rajeev Goré, Ron Levy, Dirk Pattinson, and Mukesh Tiwari. 2017. No More Excuses: Automated Synthesis of Practical and Verifiable Vote-Counting Programs for Complex Voting Schemes. In *E-VOTE-ID (Lecture Notes in Computer Science)*, Vol. 10615. Springer, 66–83.

[39] C. Andrew Neff. 2001. A verifiable secret shuffle and its application to e-voting. In *ACM Conference on Computer and Communications Security*. ACM, 116–125.

[40] Ronald L Rivest. 2008. On the notion of 'software independence' in voting systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366, 1881 (2008), 3759–3767.

[41] Ben Smyth, Steven Frink, and Michael R. Clarkson. 2015. Election Verifiability: Cryptographic Definitions and an Analysis of Helios, Helios-C, and JCJ. Cryptology ePrint Archive, Report 2015/233. (2015). https://eprint.iacr.org/2015/233.

[42] Ben Smyth, Mark Ryan, Steve Kremer, and Mounira Kourjieh. 2010. Towards Automatic Analysis of Election Verifiability Properties. In *ARSPA-WITS (Lecture Notes in Computer Science)*, Vol. 6186. Springer, 146–163.

[43] Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J Alex Halderman. 2014. Security analysis of the Estonian internet voting system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 703–715.

[44] The state of Geneva. 2018. CHVote. https://github.com/republique-et-canton-de-geneve/chvote-protocol-poc. (2018).

[45] Björn Terelius. 2015. *Some aspects of cryptographic protocols: with applications in electronic voting and digital watermarking.* Ph.D. Dissertation. KTH Royal Institute of Technology.

[46] Björn Terelius and Douglas Wikström. 2010. Proofs of Restricted Shuffles. In *AFRICACRYPT*. Springer, 100–113.

[47] Douglas Wikström. 2009. A commitment-consistent proof of a shuffle. In *Information Security and Privacy*. Springer, 407–421.

[48] Douglas Wikström. 2018. Verificatum. https://github.com/verificatum/verificatum-vcr. (2018).

[49] 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, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017.* 1789–1806. http://doi.acm.org/10.1145/3133956.3134043

A CODE

We have made our code available at https://github.com/gerlion/secure-e-voting-with-coq

B COQ CODE OMITED FROM SECTION 5

```
Class CompSigmaProtocol (Sig: Sigma.form E)   := {

  sigma_comp :> SigmaProtocol Sig;

  comp_p1: forall (sc1 sc2: Sig.S*Sig.C) (e: E) (r: Sig.R)
           (w: Sig.W),
  (Sig.Rel sc1.1 w && Sig.Rel sc2.1 w) ->
           (Sig.P1 (sc1,e) r w).2 = (Sig.P1 (sc2,e) r w).2;

  comp_simMap: forall (s1 s2: Sig.S) (r: Sig.R) (e: E) (w: Sig.W),
    Sig.simMap s1 r e w = Sig.simMap s2 r e w;
}.
```

Note that the following function does not depend on the content of the list, only its length. The function could equally be defined on a natural number; however, for technical reasons this complicates a later proof.

```
Fixpoint ApprovalSigma (c: list (G*G)) : Sigma.form :=
  match c with
    | nil => emptyForm
    | a :: b => parSigmaProtocol (ApprovalSigma b) DHTDisForm
  end.
```

```
Theorem andCorr :
    SigmaProtocol E sigma ->
     SigmaProtocol E (andSigmaProtocol E sigma).
```

```
Theorem disCorr :
    SigmaProtocol E sigma ->
    (forall (a b: E),
     sigma.disjoint a b <-> sigma.bool_eq a b = false) ->
        SigmaProtocol E (disSigmaProtocol E sigma).
```

```
Thorem parCorr :
    SigmaProtocol E sigOne ->
    SigmaProtocol E' sigTwo ->
     SigmaProtocol (E*E') (parSigmaProtocol E E' sigOne sigTwo).
```

```
Theorem andGenCorr :
    SigmaProtocol E sigOne ->
    SigmaProtocol E sigTwo ->
    (Sigma.disjoint E sigOne = Sigma.disjoint E sigTwo) ->
     SigmaProtocol E (genAndSigmaProtocol E sigOne sigTwo).
```

```
Fixpoint ApprovalSigmaStatement (g h: G) (c: list (G*G)) :
        (ApprovalSigma c).S :=
 match c with
  | nil  => (g)
  | a :: b => (ApprovalSigma g h b, (oneOrZeroCipher g h a))
 end.
```

C COQ ENCODINGS OF SIGMA PROTOCOL COMBINERS

The combiner for the equality relationship

```
Definition eqSigmaProtocol (Sig: Sigma.form E) :
         Sigma.form E  :=
  let eqS: Set := prod Sig.S1 Sig.S1 in
  let eqC: Set := prod Sig.C1 Sig.C1 in

  let eq_Rel (s: eqS) (w: Sig.W1): bool := Sig.Rel1 s.1 w
         && Sig.Rel1 s.2 w in

  let eq_P0 (s: eqS) (r: Sig.R1) (w: Sig.W1) :
         (eqS*eqC) :=
    let c1 := (Sig.P01 s.1 r w).2 in
    let c2 := (Sig.P01 s.2 r w).2 in
     (s,(c1,c2)) in

  let eq_V0 (p0: eqS*eqC) (e: E): (eqS*eqC*E) :=
    let s1 := p0.1.1 in
    let s2 := p0.1.2 in
    let c1 := p0.2.1 in
    let c2 := p0.2.2 in
   (p0, (Sig.V01 (s1,c1),e).2) in

  let eq_P1 (v0: eqS*eqC*E) (r: Sig.R1) (w: Sig.W1) :
         (eqS*eqC*E*Sig.T1) :=
  let s1 := v0.1.1.1 in
  let s2 := v0.1.1.2 in
  let c1 := v0.1.2.1 in
  let c2 := v0.1.2.2 in
  let e := v0.2 in
   (v0,(Sig.P11 (s1,c1,e) r w).2) in

 let eq_V1 (p1 : eqS*eqC*E*Sig.T1) : bool :=
   let s1 := p1.1.1.1.1 in
   let s2 := p1.1.1.1.2 in
   let c1 := p1.1.1.2.1 in
   let c2 := p1.1.1.2.2 in
   let e := p1.1.2 in
   let r := p1.2 in
   Sig.V11 (s1,c1,e,r) && Sig.V11 (s2,c2,e,r) in

  let eq_simulator(s: eqS) (r: Sig.T1) (e: E) :
         (eqS*eqC*E*Sig.T1) :=
    let s1 := s.1 in
    let s2 := s.2 in
    let sim1 := Sig.simulator1 s1 r e in
    let sim2 := Sig.simulator1 s2 r e in
    let c1 := sim1.1.1.2 in
    let c2 := sim2.1.1.2 in
    let e1 := sim1.1.2 in
    let r1 := sim1.2 in
    (s,(c1,c2),e1,r1) in

  let eq_simMap (s : eqS) (r: Sig.R1) (e :E) (w: Sig.W1) :
        (Sig.T1) :=
    Sig.simMap1 s.1 r e w in

  let eq_extractor(r1 r2: Sig.T1) (e1 e2: E): (Sig.W1) :=
    Sig.extractor1 r1 r2 e1 e2 in

  Sigma.mkForm E eqS Sig.W1 eq_Rel eqC Sig.R1
  Sig.add1 Sig.zero1 Sig.inv1 Sig.bool_eq1 Sig.disjoint1
        Sig.T1 eq_P0 eq_V0 eq_P1 eq_V1 eq_simulator
  eq_simMap eq_extractor.
```

The combiner for the and relationship

```
Definition andSigmaProtocol (Sig: Sigma.form E) :
        Sigma.form E  :=

  let andS : Set := prod Sig.S1 Sig.S1 in
  let andW : Set := prod Sig.W1 Sig.W1 in
  let andC : Set := prod Sig.C1 Sig.C1 in
  let andR : Set := prod Sig.R1 Sig.R1 in
  let andT : Set := prod Sig.T1 Sig.T1 in


  let and_Rel (s: andS) (w: andW): bool
    := Sig.Rel1 s.1 w.1 && Sig.Rel1 s.2 w.2 in

   let and_P0 (s: andS) (r: andR) (w: andW) :
         (andS*andC) :=
    let c1 := (Sig.P01 s.1 r.1 w.1).2 in
    let c2 := (Sig.P01 s.2 r.2 w.2).2 in
     (s,(c1,c2)) in

  let and_V0 (p0: andS*andC) (e: E): (andS*andC*E) :=
    let s1 := p0.1.1 in
    let s2 := p0.1.2 in
    let c1 := p0.2.1 in
    let c2 := p0.2.2 in
   (p0, (Sig.V01 (s1,c1) e).2) in

 let and_P1 (v0: andS*andC*E) (r: andR) (w: andW) :
     (andS*andC*E*andT) :=
  let s1 := v0.1.1.1 in
  let s2 := v0.1.1.2 in
  let c1 := v0.1.2.1 in
  let c2 := v0.1.2.2 in
  let e  := v0.2 in
   (v0,((Sig.P11 (s1,c1,e) r.1 w.1).2,
        (Sig.P11 (s2,c2,e) r.2 w.2).2)) in

 let and_V1 (p1: andS*andC*E*andT): bool :=
   let s1 := p1.1.1.1.1 in
   let s2 := p1.1.1.1.2 in
   let c1 := p1.1.1.2.1 in
   let c2 := p1.1.1.2.2 in
   let e  := p1.1.2 in
   let r  := p1.2 in
   Sig.V11 (s1,c1,e,r.1) && Sig.V11 (s2,c2,e,r.2) in

  let and_simulator (s: andS) (r: andT) (e: E) :
        (andS*andC*E*andT) :=
    let s1 := s.1 in
    let s2 := s.2 in
    let sim1 := Sig.simulator1 s1 r.1 e in
    let sim2 := Sig.simulator1 s2 r.2 e in
    let c1 := sim1.1.1.2 in
    let c2 := sim2.1.1.2 in
    let e1 := sim1.1.2 in
    let r1 := sim1.2 in
    let r2 := sim2.2 in
    (s,(c1,c2),e1,(r1,r2)) in

  let and_simMap (s: andS) (r: andR) (e :E) (w: andW) :
        (andT) :=
    ((Sig.simMap1 s.1 r.1 e w.1),
        (Sig.simMap1 s.2 r.2 e w.2)) in

  let and_extractor(r1 r2 : andT)(e1 e2 : E): (andW) :=
    (Sig.extractor1 r1.1 r2.1 e1 e2,
        Sig.extractor1 r1.2 r2.2 e1 e2) in
  Sigma.mkForm E andS andW and_Rel andC andR
  Sig.add1 Sig.zero1 Sig.inv1 Sig.bool_eq1 Sig.disjoint1
         andT and_P0 and_V0 and_P1 and_V1 and_simulator
         and_simMap and_extractor.
```

The combiner for the disjunctive relationship

```
Definition disSigmaProtocol (Sig: Sigma.form E) :
        Sigma.form E  :=

  (*new statement space*)
  let disS: Set := prod Sig.S1 Sig.S1 in
  (*new commit space *)
  let disC: Set := prod Sig.C1 Sig.C1 in
  (*new random space *)
  let disR: Set := prod (prod Sig.R1 Sig.T1) E in
  (*new response space*)
  let disT: Set := prod (prod Sig.T1 E) Sig.T1 in

  let dis_Rel (s: disS) (w: Sig.W1): bool :=
        Sig.Rel1 s.1 w || Sig.Rel1 s.2 w in

  let dis_P0 (s: disS) (rzeb: disR) (w: Sig.W1) :
        (disS*disC) :=
    let e   := rzeb.2 in
    let z   := rzeb.1.2 in
    let r   := rzeb.1.1 in
    let hc1 := (Sig.P01 s.1 r w).2 in
    let hc2 := (Sig.P01 s.2 r w).2 in
    let sc1 := (Sig.simulator1 s.1 z e).1.1.2 in
    let sc2 := (Sig.simulator1 s.2 z e).1.1.2 in
    if (Sig.Rel1 s.1 w) then (s,(hc1,sc2))
        else (s,(sc1,hc2)) in

  let dis_V0 (p0: disS*disC) (e: E): (disS*disC*E) :=
   (p0, e) in

 let dis_P1 (v0: disS*disC*E) (rzeb: disR) (w: Sig.W1) :
        (disS*disC*E*disT) :=
  let s1 := v0.1.1.1 in
  let s2 := v0.1.1.2 in
  let c1 := v0.1.2.1 in
  let c2 := v0.1.2.2 in
  let e  := v0.2 in
  let se := rzeb.2 in
  let z  := rzeb.1.2 in
  let r  := rzeb.1.1 in
  let e1 := (Sig.V01 (s1, c1)
        (Sig.add1 e (Sig.inv1 se))).2 in
  let ht1 := (Sig.P11 (s1,c1,e1) r w).2 in
  let ht2 := (Sig.P11 (s2,c2,e1) r w).2 in
  let st1 := (Sig.simulator1 s1 z se).2 in
  let st2 := (Sig.simulator1 s2 z se).2 in
   if (Sig.Rel1 s1 w) then (v0, ((ht1,e1,st2)))
      else   (v0, ((st1,se,ht2))) in

 let dis_V1 (p1: disS*disC*E*disT) : bool :=
   let s1 := p1.1.1.1.1 in
   let s2 := p1.1.1.1.2 in
   let c1 := p1.1.1.2.1 in
   let c2 := p1.1.1.2.2 in
   let e  := p1.1.2 in
   let e1 := p1.2.1.2 in
   let e2 := (Sig.add1 e (Sig.inv1 e1)) in
   let r1 := p1.2.1.1 in
   let r2 := p1.2.2 in
   (Sig.V11 (s1,c1,e1,r1) && Sig.V11 (s2,c2,e2,r2)) in

  let dis_simulator(s: disS) (t: disT) (e: E) :
        (disS*disC*E*disT) :=
    let s1 := s.1 in
    let s2 := s.2 in
    let e1 := t.1.2 in
    let e2 := (Sig.add1 e (Sig.inv1 e1)) in
    let r1 := t.1.1 in
    let r2 := t.2 in
    let sim1 := Sig.simulator1 s1 r1 e1 in
    let sim2 := Sig.simulator1 s2 r2 e2 in
    let c1 := sim1.1.1.2 in
    let c2 := sim2.1.1.2 in
    let sr1 := sim1.2 in
    let sr2 := sim2.2 in
    let se1 := sim1.1.2 in
    let se2 := sim2.1.2 in
      (s,(c1,c2),e,((sr1,se1), (sr2))) in

  let dis_simMap (s: disS) (rtcb: disR) (e :E) (w: Sig.W1)
        : (disT) :=
    let r := rtcb.1.1 in
    let t := rtcb.1.2 in
    let c := rtcb.2 in
    let h1 := Sig.simMap1 s.1
        r (Sig.add1 e (Sig.inv1 c)) w in
    let h2 := Sig.simMap1 s.2 r
         (Sig.add1 e (Sig.inv1 c)) w in
    if (Sig.Rel1 s.1 w) then (h1, Sig.add1 e (
        Sig.inv1 c),t) else (t,c,h2) in

  let dis_extractor (r1 r2: disT) (c1 c2: E): (Sig.W1) :=
    let e1 := r1.1.2 in
    let e2 := (Sig.add1 c1 (Sig.inv1 e1)) in
    let e3 := r2.1.2 in
    let e4 := (Sig.add1 c2 (Sig.inv1 e3)) in
    let t1 := r1.1.1 in
    let t2 := r1.2 in
    let t3 := r2.1.1 in
    let t4 := r2.2 in
   if ~~(Sig.bool_eq1 e1 e3) then Sig.extractor1 t1 t3 e1
      e3 else
    Sig.extractor1 t2 t4 e2 e4 in

  Sigma.mkForm E disS Sig.W1 dis_Rel disC disR
  Sig.add1 Sig.zero1 Sig.inv1 Sig.bool_eq1 Sig.disjoint1
disT dis_P0 dis_V0
  dis_P1 dis_V1 dis_simulator
  dis_simMap dis_extractor.
```

The combiner for the parallel relationship

```
Definition parSigmaProtocol (Sig1: Sigma.form E)
        (Sig2: Sigma.form E'): Sigma.form (E*E') :=

  let parS: Set := prod Sig1.S1 Sig2.S2 in
  let parW: Set := prod Sig1.W1 Sig2.W2 in
  let parC: Set := prod Sig1.C1 Sig2.C2 in
  let parR: Set := prod Sig1.R1 Sig2.R2 in
  let parE: Set := prod E E' in
  let parT: Set := prod Sig1.T1 Sig2.T2 in

  let par_Rel (s: parS) (w: parW): bool :=
        Sig1.Rel1 s.1 w.1 && Sig2.Rel2 s.2 w.2 in

  let par_add (e1 e2: parE): parE :=
    (Sig1.add1 e1.1 e2.1, Sig2.add2 e1.2 e2.2) in

  let par_zero: parE  :=
    (Sig1.zero1, Sig2.zero2) in

  let par_bool_eq (e1 e2: parE): bool :=
     Sig1.bool_eq1 e1.1 e2.1 &&
         Sig2.bool_eq2 e1.2 e2.2 in

  let par_inv (e: parE): parE :=
    (Sig1.inv1 e.1, Sig2.inv2 e.2) in

  let par_disjoint (e1 e2: parE): bool :=
     Sig1.disjoint1 e1.1 e2.1 &&
         Sig2.disjoint2 e1.2 e2.2 in

  let par_P0 (s: parS) (r: parR) (w: parW):
        (parS*parC) :=
    let c1 := (Sig1.P01 s.1 r.1 w.1).2 in
    let c2 := (Sig2.P02 s.2 r.2 w.2).2 in
     (s,(c1,c2)) in
  let par_V0 (p0: parS*parC) (e: parE) :
         (parS*parC*parE) :=
    let s1 := p0.1.1 in
    let s2 := p0.1.2 in
    let c1 := p0.2.1 in
    let c2 := p0.2.2 in
   (p0, ((Sig1.V01 (s1,c1) e.1).2,
        (Sig2.V02 (s2,c2) e.2).2)) in

 let par_P1 (v0: parS*parC*parE) (r: parR) (w: parW) :
     (parS*parC*parE*parT) :=
  let s1 := v0.1.1.1 in
  let s2 := v0.1.1.2 in
  let c1 := v0.1.2.1 in
  let c2 := v0.1.2.2 in
  let e  := v0.2 in
   (v0,((Sig1.P11 (s1,c1,e.1) r.1 w.1).2,
        (Sig2.P12 (s2,c2,e.2) r.2 w.2).2)) in

 let par_V1 (p1: parS*parC*parE*parT) : bool :=
   let s1 := p1.1.1.1.1 in
   let s2 := p1.1.1.1.2 in
   let c1 := p1.1.1.2.1 in
   let c2 := p1.1.1.2.2 in
   let e  := p1.1.2 in
   let r  := p1.2 in
   Sig1.V11 (s1,c1,e.1,r.1) && Sig2.V12 (s2,c2,e.2,r.2) in

  let par_simulator (s: parS) (r: parT) (e: parE) :
        (parS*parC*parE*parT) :=
    let s1 := s.1 in
    let s2 := s.2 in
    let sim1 := Sig1.simulator1 s1 r.1 e.1 in
    let sim2 := Sig2.simulator2 s2 r.2 e.2 in
    let c1 := sim1.1.1.2 in
    let c2 := sim2.1.1.2 in
    let e1 := sim1.1.2 in
    let e2 := sim2.1.2 in
    let r1 := sim1.2 in
    let r2 := sim2.2 in
    (s,(c1,c2),(e1,e2),(r1,r2)) in

  let par_simMap (s: parS) (r: parR) (e :parE) (w: parW) :
        (parT) :=
    ((Sig1.simMap1 s.1 r.1 e.1 w.1),
        (Sig2.simMap2 s.2 r.2 e.2 w.2)) in

  let par_extractor(r1 r2: parT) (e1 e2: parE) :
         (parW) :=
    (Sig1.extractor1 r1.1 r2.1 e1.1 e2.1,
         Sig2.extractor2 r1.2 r2.2 e1.2 e2.2) in

  Sigma.mkForm (E*E') parS parW par_Rel parC parR
  par_add par_zero par_inv par_bool_eq par_disjoint parT
  par_P0 par_V0 par_P1 par_V1 par_simulator par_simMap
 par_extractor.
```

The combiner for the generalised and relationship

```
Definition genAndSigmaProtocol (Sig1 Sig2: Sigma.form E)
        : Sigma.form E :=
  let genAndS : Set := prod Sig1.S1 Sig2.S1 in
  let genAndW : Set := prod Sig1.W1 Sig2.W1 in
  let genAndC : Set := prod Sig1.C1 Sig2.C1 in
  let genAndR : Set := prod Sig1.R1 Sig2.R1 in
  let genAndT : Set := prod Sig1.T1 Sig2.T1 in

  let genAnd_Rel (s : genAndS) (w :genAndW): bool :=
         Sig1.Rel1 s.1 w.1 && Sig2.Rel1 s.2 w.2 in

  let genAnd_P0 (s: genAndS) (r: genAndR) (w: genAndW)
        : (genAndS*genAndC) :=
    let c1 := (Sig1.P01 s.1 r.1 w.1).2 in
    let c2 := (Sig2.P01 s.2 r.2 w.2).2 in
     (s,(c1,c2)) in

  let genAnd_V0 (p0: genAndS*genAndC) (e: E) :
         (genAndS*genAndC*E) :=
    let s1 := p0.1.1 in
    let s2 := p0.1.2 in
    let c1 := p0.2.1 in
    let c2 := p0.2.2 in
    (p0, e) in

 let genAnd_P1 (v0: genAndS*genAndC*E) (r: genAndR)
      (w: genAndW): (genAndS*genAndC*E*genAndT) :=
  let s1 := v0.1.1.1 in
  let s2 := v0.1.1.2 in
  let c1 := v0.1.2.1 in
  let c2 := v0.1.2.2 in
  let e  := v0.2 in
   (v0,((Sig1.P11 (s1,c1,e) r.1 w.1).2,
        (Sig2.P11 (s2,c2,e) r.2 w.2).2)) in

 let genAnd_V1 (p1: genAndS*genAndC*E*genAndT): bool :=
   let s1 := p1.1.1.1.1 in
   let s2 := p1.1.1.1.2 in
   let c1 := p1.1.1.2.1 in
   let c2 := p1.1.1.2.2 in
   let e  := p1.1.2 in
   let r  := p1.2 in
   Sig1.V11 (s1,c1,e,r.1) && Sig2.V11 (s2,c2,e,r.2) in

  let genAnd_simulator (s: genAndS) (r: genAndT) (e: E) :
         (genAndS*genAndC*E*genAndT) :=
    let s1 := s.1 in
    let s2 := s.2 in
    let sim1 := Sig1.simulator1 s1 r.1 e in
    let sim2 := Sig2.simulator1 s2 r.2 e in
    let c1 := sim1.1.1.2 in
    let c2 := sim2.1.1.2 in
    let r1 := sim1.2 in
    let r2 := sim2.2 in
    (s,(c1,c2),e,(r1,r2)) in

  let genAnd_simMap (s: genAndS) (r: genAndR) (e: E)
        (w: genAndW) : (genAndT) :=
    ((Sig1.simMap1 s.1 r.1 e w.1),
        (Sig2.simMap1 s.2 r.2 e w.2)) in

  let genAnd_extractor(r1 r2: genAndT) (e1 e2: E) :
         (genAndW) :=
    (Sig1.extractor1 r1.1 r2.1 e1 e2, Sig2.extractor1
         r1.2 r2.2 e1 e2) in

  Sigma.mkForm E genAndS genAndW genAnd_Rel genAndC
  genAndR Sig1.add1 Sig1.zero1 Sig1.inv1 Sig1.bool_eq1
  Sig1.disjoint1 genAndT genAnd_P0 genAnd_V0
  genAnd_P1 genAnd_V1 genAnd_simulator
  genAnd_simMap genAnd_extractor.
```

D COQ CODE OMITTED FOR SECTION 7

```
Definition com (h: G) (hs: V2G) (m: M2F) (r: V2F): V2G :=
  Build_V2G(EPC h hs (M2F_col1 m) (r1 r))
        (EPC h hs (M2F_col2 m) (r2 r))
```

```
(* The commitment is to a permutation *)
Definition relPi (h: G) (hs: V2G) (c: V2G) (*Statement *)
  (m : M2F)(r : V2F) :=     (*Witness*)
    M2F_isPermutation m
     /\ c = (com h hs m r).
```

```
(* Definition of shuffling *) (* e2_i = e1_p_i * r_p_i *)
Definition relReEnc(g pk : G)(e e' : (V2G*V2G))(m : M2F)
  (r : V2F) :=
  let e'' := ciphMatrix e' m in
  let r'' := M2F_CVmult m r in
    IsReEnc g pk e.1 e''.1 (r1 r'')
    /\ IsReEnc g pk e.2 e''.2 (r2 r'').
```

```
Definition u'_Rel (s: (G*G*G*V2G*(V2G*V2G))*(G*V2G*V2G))
        (w: V2F*F*F*V2F) :=
  let parm := s.1 in
  let g    := parm.1.1.1.1 in
  let pk   := parm.1.1.1.2 in
  let h    := parm.1.1.2 in
  let hs   := parm.1.2 in
  let e'   := parm.2 in

  let stat := s.2 in
  let a    := stat.1.1 in
  let b    := stat.1.2 in
  let cHat := stat.2 in

  let u'    := w.1.1.1 in
  let rTil  := w.1.1.2 in
  let rStar := w.1.2 in
  let rHat  := w.2 in

  Gbool_eq a (EPC h hs u' rTil) &&
  V2G_eq b (V2G_mult (V2G_Tprod (ciphExp e' u'))
        (Enc g pk rStar Gone)) &&
  Gbool_eq (m1 cHat) (PC h (m1 hs) (r1 u') (r1 rHat)) &&
  Gbool_eq (m2 cHat) (PC h (m1 cHat) (r2 u') (r2 rHat)).
```

```
(** Begin Sigma Proper *)
(* We pass why to allow branching in disjunction *)
Definition u'_P0 (s : (G*G*G*V2G*(V2G*V2G))*(G*V2G*V2G))
    (r w : V2F*F*F*V2F) : (G*G*G*V2G*(V2G*V2G))*
        (G*V2G*V2G)*(G*V2G*V2G) :=
  let parm := s.1 in
  let g    := parm.1.1.1.1 in
  let pk   := parm.1.1.1.2 in
  let h    := parm.1.1.2 in
  let hs   := parm.1.2 in
  let e'   := parm.2 in

  let stat := s.2 in
  let a    := stat.1.1 in
  let b    := stat.1.2 in
  let cHat := stat.2 in

  let u'   := w.1.1.1 in
  let rTil := w.1.1.2 in
  let rStar := w.1.2 in
  let rHat := w.2 in

  let w'   := r.1.1.1 in
  let w3   := r.1.1.2 in
  let w4   := r.1.2 in
  let wHat := r.2 in

  let t3 := EPC h hs w' w3 in
  let t4 := V2G_mult (V2G_Tprod (ciphExp e' w'))
        (Enc g pk w4 Gone) in
  let tHat1 := PC h (m1 hs) (r1 w') (r1 wHat) in
  let tHat2 := PC h (m1 cHat) (r2 w') (r2 wHat) in

  (s, (t3, t4, Build_V2G tHat1 tHat2)).
```

```
Definition u'_V0 (ggtoxgtor: (G*G*G*V2G*(V2G*V2G))*
        (G*V2G*V2G)*(G*V2G*V2G)) (c: F):
         ((G*G*G*V2G*(V2G*V2G))*(G*V2G*V2G)*(G*V2G*V2G)*F)
  := (ggtoxgtor, c).

Definition u'_P1 (ggtoxgtorc: (G*G*G*V2G*(V2G*V2G))*
        (G*V2G*V2G)*(G*V2G*V2G)*F) (r w: V2F*F*F*V2F):
          (G*G*G*V2G*(V2G*V2G))*(G*V2G*V2G)*
          (G*V2G*V2G)*F*(V2F*F*F*V2F) :=
  let c      :=  snd (ggtoxgtorc) in

  let u'    := w.1.1.1 in
  let rTil  := w.1.1.2 in
  let rStar := w.1.2 in
  let rHat  := w.2 in

  let w'   := r.1.1.1 in
  let w3   := r.1.1.2 in
  let w4   := r.1.2 in
  let wHat := r.2 in

  let s3   := w3+rTil*c in
  let s4   := w4+rStar*c in
  let sHat := V2F_add wHat (V2F_scale rHat c) in
  let s'   := V2F_add w' (V2F_scale u' c) in

  (ggtoxgtorc, (s', s3, s4, sHat)).
```

```
Definition u'_V1 (transcript :
  (G*G*G*V2G*(V2G*V2G))*(G*V2G*V2G)*(G*V2G*V2G)*F*
        (V2F*F*F*V2F)) :=

  let s := transcript.1.1.1 in
  let c := transcript.1.1.2 in
  let e := transcript.1.2 in
  let t := transcript.2 in

  let parm := s.1 in
  let g    := parm.1.1.1.1 in
  let pk   := parm.1.1.1.2 in
  let h    := parm.1.1.2 in
  let hs   := parm.1.2 in
  let e'   := parm.2 in

  let stat := s.2 in
  let a    := stat.1.1 in
  let b    := stat.1.2 in
  let cHat := stat.2 in

  let t3   := c.1.1 in
  let t4   := c.1.2 in
  let tHat :=  c.2 in

  let s3   := t.1.1.2 in
  let s4   := t.1.2 in
  let sHat := t.2 in
  let s'   := t.1.1.1 in

  Gbool_eq (t3 o a^e) (EPC h hs s' s3)
  && V2G_eq (V2G_mult t4 (V2G_Sexp b e))
           (V2G_mult (V2G_Tprod (ciphExp e' s'))
           (Enc g pk s4 Gone))
  && Gbool_eq (m1 (V2G_mult tHat (V2G_Sexp cHat e)))
             (PC h (m1 hs) (r1 s') (r1 sHat))
  && Gbool_eq (m2 (V2G_mult tHat (V2G_Sexp cHat e)))
             (PC h (m1 cHat) (r2 s') (r2 sHat)).
```

```
Definition u'_simulator_mapper (s : (G*G*G*V2G*(V2G*V2G))
        *(G*V2G*V2G))
  (r:  V2F*F*F*V2F) (c: F) (w: V2F*F*F*V2F):=

  let u'    := w.1.1.1 in
  let rTil  := w.1.1.2 in
  let rStar := w.1.2 in
  let rHat  := w.2 in

  let w'   := r.1.1.1 in
  let w3   := r.1.1.2 in
  let w4   := r.1.2 in
  let wHat := r.2 in

  let s3   := w3+rTil*c in
  let s4   := w4+rStar*c in
  let sHat := V2F_add wHat (V2F_scale rHat c) in
  let s'   := V2F_add w' (V2F_scale u' c) in

  (s', s3, s4, sHat).
```

For reasons of space we have omitted the remaining coq formalisations and refer the reader to the code repository mentioned in appendix A for the relevant material.