Type-Based Verification of Electronic Voting Protocols

Véronique Cortier[1], Fabienne Eigner[2], Steve Kremer[1], Matteo Maffei[2], and Cyrille Wiedling[3]

[1] LORIA, CNRS & INRIA & University of Lorraine, France
{cortier,steve.kremer}@loria.fr
[2] CISPA, Saarland University, Germany
{eigner,maffei}@cs.uni-saarland.de
[3] Université Catholique de Louvain, Belgium
cyrille.wiedling@uclouvain.be

Abstract. E-voting protocols aim at achieving a wide range of sophisticated security properties and, consequently, commonly employ advanced cryptographic primitives. This makes their design as well as rigorous analysis quite challenging. As a matter of fact, existing automated analysis techniques, which are mostly based on automated theorem provers, are inadequate to deal with commonly used cryptographic primitives, such as homomorphic encryption and mix-nets, as well as some fundamental security properties, such as verifiability.
This work presents a novel approach based on refinement type systems for the automated analysis of e-voting protocols. Specifically, we design a generically applicable logical theory which, based on pre- and post-conditions for security-critical code, captures and guides the type-checker towards the verification of two fundamental properties of e-voting protocols, namely, vote privacy and verifiability. We further develop a code-based cryptographic abstraction of the cryptographic primitives commonly used in e-voting protocols, showing how to make the underlying algebraic properties accessible to automated verification through logical refinements. Finally, we demonstrate the effectiveness of our approach by developing the first automated analysis of Helios, a popular web-based e-voting protocol, using an off-the-shelf type-checker.

1 Introduction

Cryptographic protocols are notoriously difficult to design and their manual security analysis is extremely complicated and error-prone. As a matter of fact, security vulnerabilities have accompanied early academic protocols like Needham-Schroeder [48] as well as carefully designed de facto standards like SSL [55], PKCS #11 [18], and the SAML-based Single Sign-On for Google Apps [4]. E-voting protocols are particularly tricky, since they aim at achieving sophisticated security properties, such as verifiability and coercion-resistance, and, consequently, employ advanced cryptographic primitives such as homomorphic encryptions, mix-nets, and zero-knowledge proofs. Not surprisingly, this makes the attack surface even larger, as witnessed by the number of attacks on e-voting protocols proposed in the literature (see e.g., [35,29,44]).

This state of affairs has motivated a substantial research effort on the formal analysis of cryptographic protocols, which over the years has led to the development of

several automated tools based on symbolic abstractions of cryptography. *Automated theorem provers* build on a term-based abstraction of cryptography and proved successful in the enforcement of various trace properties [16,6,9,32] and even observational equivalence relations [17,25,26]. While some of these tools have also been used in the context of e-voting [42,8,52,3], they fall short of supporting the cryptographic primitives and security properties specific of this setting. For instance, none of them supports the commutativity property of homomorphic encryption that is commonly exploited to compute the final tally in a privacy-preserving manner (e.g., [27,31,2]), and the proof of complex properties like verifiability or coercion-resistance must be complemented by manual proofs [33,52] or encodings [8] respectively, which are tedious and error-prone.

Another line of research has focused on the design of *type systems* for cryptographic protocol analysis. Refinement type systems, in particular, allow for tracking pre- and post-conditions on security-sensitive code, and thus enforcing various trace properties, such as authentication [39,23,24,19,20,5,36], classical authorization policies [7,14,10], and linear authorization policies [21,22]. Type systems proved capable to enforce even observational equivalence relations, such as secrecy [37], relational properties [54], and differential privacy [34]. Type systems are to some extent less precise than theorem provers and are not suitable to automatically report attacks, in that they do not explicitly compute abstractions of execution traces, but they are modular and therefore scale better to large-scale protocols. Furthermore, by building on a code-based, as opposed to term-based, abstraction of cryptography, they enable reasoning about sophisticated cryptographic schemes [37,10]. Although they look promising, type systems have never been used in the context of e-voting protocols. This task is challenging since, for guiding the type-checking procedure, one needs to develop a dedicated logical theory, capturing the structure of e-voting systems and the associated security and privacy properties.

Our contributions. We devise a novel approach based on refinement type systems for the formal verification of e-voting protocols. Specifically,

- we design a generically applicable logical theory based on pre- and post-conditions for security-critical code, which captures and guides the type-checker towards the verification of two fundamental properties, namely, vote privacy and verifiability;
- we formalize in particular three different verifiability properties (i.e., individual, universal, and end-to-end verifiability), proving for the first time that individual verifiability plus universal verifiability imply end-to-end verifiability, provided that ballots cannot be confused (no-clash property [44]);
- we develop a code-based cryptographic abstraction of the cryptographic primitives commonly used in e-voting protocols, including homomorphic encryption, showing how to make its commutativity and associativity properties accessible to automated verification through logical refinements;
- we demonstrate the effectiveness of our approach by analyzing Helios [2], a popular, state-of-the-art, voting protocol that has been used in several real-scale elections, including elections at Louvain-la-Neuve, Princeton, and among the IACR [40]. We analyze the two main versions of Helios that respectively use homomorphic encryption and mix-net based tally. For this we use F* [54], an off-the-shelf type-checker supporting the verification of trace properties and observational equivalence relations, as required for verifiability and vote privacy, through refinement

and relational types, respectively. Analyzing Helios with homomorphic encryption was out of reach of existing tools due to the need of a theory that reflects the addition of the votes. A strength of our approach is that proof obligations involving such theories can be directly discharged to SMT solvers such as Z3 [50].

Related work. Many symbolic protocol verification techniques have been applied to analyze e-voting systems [42,8,33,52,43,29,30,3]. In all of these works, the cryptographic primitives are modeled using terms and an equational theory, as opposed to the code-based abstractions we use in this paper. While code-based abstractions of cryptography may look at a first glance more idealized than the modeling using equational theories, they are actually closer to ideal functionalities in simulation-based cryptographic frameworks. Although a formal computational soundness result is out of the scope of this work, the code-based abstractions we use are rather standard and computational soundness results for similar abstractions have been proven in [12,37].

One of the main advantages of symbolic protocol verification is the potential for automation. However, current automated protocol verification tools are not yet mature enough to analyze most voting protocols. Firstly, existing tools do not support equational theories modeling homomorphic encryption. Thus, existing analyses of systems that rely on homomorphic tallying all rely on hand proofs [29,30,43], which are complicated and error-prone due to the complexity of the equational theories. Secondly, most current automated tools offer only limited support for verifying equivalence properties, which is required for verifying vote privacy. For instance, in [8] the analysis of Civitas using ProVerif relies on manual encodings and many other works, even though the equational theory is in the scope of the tools, again rely on hand proofs of observational equivalences [42,33]. Although some recent tools, such as AKiSs [25] succeed in analyzing simple protocols such as [38], more complicated protocols are still out of reach. In [3], the privacy of the mix-net based version of Helios was shown using AKiSs, but mix-nets were idealized by simply outputting the decrypted votes in a non-deterministic order. In contrast, our model manipulates lists to express the fact that a mix-net produces a permutation. ProVerif was used to check some cases of verifiability [52], but automation is only supported for protocols without a homomorphic tally.

Other work on e-voting protocols considers definitions of privacy [45,15] and verifiability [41,46,28] in computational models. However, no computer aided verification techniques have yet been applied in this context. Furthermore, prior work [47] demonstrated that individual and universal verifiability in general do not imply end-to-end verifiability, not even by assuming the no-clash property, using as an example the Three-Ballot voting system [51]. In this paper, we show on the contrary that individual and universal verifiability do imply end-to-end verifiability. This is due to the fact that our individual verifiability notion is actually stronger and assumes that the board can be uniquely parsed as a list of ballots. This is the case for many voting systems but not for ThreeBallot where each ballot is split into three components.

2 Background

We review the fundamental concepts underlying the typed-based analysis of security protocols and we present the Helios e-voting protocol that constitutes our case study.

2.1 Refinement types for cryptographic protocols

Computational RCF. The protocols we analyze are implemented in Computational RCF et al. [37], a λ-calculus with references, assumptions, and assertions. We briefly review below the syntax and semantics of the language, referring to Appendix A for more details. Constructors, ranged over by h, include inl and inr, which are used to construct tagged unions, and fold, which is used to construct recursive data structures. Values, ranged over by M, N, comprise variables x, y, z, the unit value $()$, pairs (M, N), constructor applications $h\ M$, functions $\mathsf{fun}\ x \to A$, and functions read_a and write_a to read from and write to a memory location a, respectively. The syntax and semantics of expressions are mostly standard. $M\ N$ behaves as $A\{N/x\}$ (i.e., A where x is replaced by N) if $M = \mathsf{fun}\ x \to A$, otherwise it gets stuck; $\mathsf{let}\ x = A\ \mathsf{in}\ B$ evaluates A to M and then behaves as $B\{M/x\}$; $\mathsf{let}\ (x, y) = M\ \mathsf{in}\ A$ behaves as $A\{N/x, N'/y\}$ if $M = (N, N')$, otherwise it gets stuck; $\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$ behaves as $A\{N/x\}$ if $M = h\ N$, as B otherwise; $\mathsf{ref}\ M$ allocates a fresh label a and returns the reading and writing functions $(\mathsf{read}_a, \mathsf{write}_a)$. The code is decorated with assumptions $\mathsf{assume}\ F$ and assertions $\mathsf{assert}\ F$. The former introduce logical formulas that are assumed to hold at a given program point, while the latter specify logical formulas that are expected to be entailed by the previously introduced assumptions.

Definition 1 (Safety). *A closed expression A is safe iff the formulas asserted at runtime are logically entailed by the previously assumed formulas.*

The code is organized in *modules*, which are intuitively a sequence of function declarations. A module may export some of the functions defined therein, which can then be used by other modules: we let $B \cdot A$ denote the composition of modules B and A, where the functions exported by B may be used in A.

Types and typing judgements. Table 1 shows the syntax of types. Types bool for boolean values and bytes for bitstrings can be constructed from unit by encoding[4]. The singleton unit type is populated by the value $()$; $\mu\alpha.T$ describes values of the form $\mathsf{fold}\ M$, where M has the unfolded type $T\{\mu\alpha.T/\alpha\}$; $T + U$ describes values of the form $\mathsf{inl}\ M$ or $\mathsf{inr}\ M$, where M has type T or U, respectively; the dependent type $x : T * U$ describes pairs of values (M, N), where M has type T and N has type $U\{M/x\}$; the dependent type $x : T \to U$ describes functions taking as input a value M of type T and returning a value of type $U\{M/x\}$; the dependent refinement type $x : T\{F\}$ describes values M of type T such that the logical formula $F\{M/x\}$ is entailed by the active assumptions. Notice that a refinement on the input of a function expresses a pre-condition, while a refinement on the output expresses a post-condition.

The typing judgement $I \vdash A : T$ says that expression A can be typed with type T in a typing environment I. Intuitively, a typing environment binds the free variables and labels in A to a type. The typing judgement $I \vdash B \rightsquigarrow I'$ says that under environment I module B is well-typed and exports the typed interface I'.

Modeling the protocol and the opponent. The protocol is encoded as a module, which exports functions defining the cryptographic library as well as the protocol parties. The latter are modeled as cascaded functions, which take as input the messages received

[4] E.g., boolean values are encoded as $\mathsf{true} \triangleq \mathsf{inl}\ ()$ and $\mathsf{false} \triangleq \mathsf{inr}\ ()$

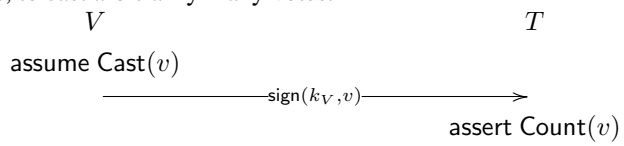
$T, U, V ::=$		type
	unit	unit type
	α	type variable
	$\mu\alpha.T$	iso-recursive type (α bound in τ)
	$T + U$	sum type
	$x : T * U$	dependent pair type (x bound in U)
	$x : T \to U$	dependent function type (x bound in U)
	$x : T\{F\}$	dependent refinement type (x bound in F)

Table 1. Syntax of types

from the network and return the pair composed of the value to be output on the network and the continuation code [5]. Concurrent communication is modeled by letting the opponent, which has access to the exported functions, act as a scheduler.

Modeling the cryptographic library. We rely on a sealing-based abstraction of cryptography [49,53]. A seal for a type T consists of a pair of functions: the sealing function of type $T \to \mathsf{bytes}$ and the unsealing function of type $\mathsf{bytes} \to T$. The sealing mechanism is implemented by storing a list of pairs in a global reference that can only be accessed using the sealing and unsealing functions. The sealing function pairs the payload with a fresh, public value (the handle) representing its sealed version, and stores the pair in the list. The unsealing function looks up the public handle in the list and returns the associated payload. For symmetric cryptography, the sealing and unsealing functions are both private and naturally model encryption and decryption keys, respectively: a payload of type T is sealed to type bytes and can be sent over the untrusted network, while a message retrieved from the network with type bytes can be unsealed to its correct type T. Different cryptographic primitives, like public key encryptions and signature schemes, can be encoded in a similar way, by exporting the function modeling the public key to the opponent. We will give further insides on how to build sealing-based abstractions for more sophisticated cryptographic primitives, such as homomorphic encryptions and proofs of knowledge in Section 4.

Type-based verification. Assumptions and assertions can be used to express a variety of trace-based security properties. For instance, consider the very simple e-voting protocol below, which allows everyone in possession of the signing key k_V, shared by all eligible voters, to cast arbitrarily many votes.



The assumption $\mathsf{Cast}(v)$ on the voter's side tracks the intention to cast vote v. The authorization policy $\forall v.\mathsf{Cast}(v) \Rightarrow \mathsf{Count}(v)$, which is further defined in the system as a global assumption expresses the fact that all votes cast by eligible voters should be counted. Since this is the only rule entailing $\mathsf{Count}(v)$, this rule actually captures a correspondence assertion: votes can be counted only if they come from eligible voters.

[5] For the sake of readability we use the standard message-passing-style syntax in our examples and some additional syntactic sugar (e,g., sequential let declarations) that are easy to encode.

The assertion assert $\mathsf{Count}(v)$ on the tallying authority's side expresses the expectation that vote v should be counted.

In order to type-check the code of authority T, it suffices to prove $\mathsf{Cast}(v)$ on the authority's side, which entails $\mathsf{Count}(v)$ through the authorization policy. Since the type-checking algorithm is modular (i.e., each party is independently analyzed) and $\mathsf{Cast}(v)$ is assumed on the voter's side, this formula needs to be conveyed to T. This is achieved by giving the vote v the *refinement type* $x : \mathsf{bytes}\{\mathsf{Cast}(x)\}$. In order to type v on the voter's side with such a type, v needs to be of type bytes and additionally, the formula $\mathsf{Cast}(v)$ needs to be entailed by the previous assumptions, which is indeed true in this case. In our sealing-based library for signatures signing corresponds to sealing a value and verification is modeled using the unsealing function and thus the types of signing and verification are $\mathsf{sigkey}(T) \triangleq T \to \mathsf{bytes}$ and $\mathsf{verkey}(T) \triangleq \mathsf{bytes} \to T$, while the types of the signing and verification functions are $\mathsf{sig} : \mathsf{sigkey}(T) \to T \to \mathsf{bytes}$ and $\mathsf{ver} : \mathsf{verkey}(T) \to \mathsf{bytes} \to T$, respectively.[6] Here T is $x : \mathsf{bytes}\{\mathsf{Cast}(x)\}$, thereby imposing a pre-condition on the signing function (before signing x, one has to assume the formula $\mathsf{Cast}(x)$) and a post-condition on the verification function (after a successful verification, the formula $\mathsf{Cast}(x)$ is guaranteed to hold for the signed x).

When reasoning about the implementations of cryptographic protocols, we are interested in the safety of the protocol against an arbitrary opponent.

Definition 2 (Opponent and Robust Safety). *A closed expression O is an opponent iff O contains no assumptions or assertions. A closed module A is robustly safe w.r.t. interface I iff for all opponents O such that $I \vdash O : T$ for some type T, $A \cdot O$ is safe.*

Following the approach advocated in [37], the typed interface I exported to the opponent is supposed to build exclusively on the type bytes, without any refinement. This means that the attacker is not restricted by any means and can do whatever it wants with the messages received from the network, except for performing invalid operations that would lead it to be stuck (e.g., treating a pair as a function). In fact, the well-typedness assumption for the opponent just makes sure that the only free variables occurring therein are the ones exported by the protocol module. Robust safety can be statically enforced by type-checking, as stated below.

Theorem 1 (Robust Safety). *If $\emptyset \vdash A \rightsquigarrow I$ then A is robustly safe w.r.t. I.*

2.2 Helios

Helios [2] is a verifiable and privacy-preserving e-voting system. It has been used in several real-life elections such that student elections at the University of Louvain-la-Neuve or at Princeton. It is now used by the IACR to elect its board since 2011 [40]. The current implementation of Helios (Helios 2.0) is based on homomorphic encryption, which makes it possible to decrypt only the aggregation of the ballots as opposed

[6] We note that the verification function only takes the signature as an input, checks whether it is indeed a valid signature and if so, retrieves the corresponding message that was signed. This is a standard abstraction and used for convenience, an alternate approach would be to have verification take both the signature and message as an input and return a boolean value. The sealing-based library functions for both versions are very similar.

to the individual ballots. Homomorphic tally, however, requires encrypted ballots to be split in several ciphertexts, depending on the number of candidates. For example, in case of 4 candidates and a vote for the second one, the encrypted ballot would be $\{0\}^{r_1}_{\mathsf{pk}}, \{1\}^{r_2}_{\mathsf{pk}}, \{0\}^{r_3}_{\mathsf{pk}}, \{0\}^{r_4}_{\mathsf{pk}}$. In case the number of candidates is high, the size of a ballot and the computation time become large. Therefore, there exists a variant of Helios that supports mix-net-based tally: ballots are shuffled and re-randomized before being decrypted. Both variants co-exist since they both offer advantages: mix-nets can cope with a large voting space while homomorphic tally eases the decryption phase (only one ballot needs to be decrypted, no need of mixers). We present here both variants of Helios, which constitute our case studies. For simplicity, in the case of homomorphic tally, we assume that voters are voting either 0 or 1 (referendum).

The voting process in Helios is divided in two main phases. The bulletin board is a public webpage that starts initially empty. Votes are encrypted using a public key pk. The corresponding decryption key dk is shared among trustees. For privacy, the trust assumption is that at least one trustee is honest (or that the trustees do not collaborate).

Voting phase. During the voting phase, each voter encrypts her vote v using the public key pk of the election.She then sends her encrypted vote $\{v\}^r_{\mathsf{pk}}$ (where r denotes the randomness used for encrypting), together with some auxiliary data aux, to the bulletin board through an authenticated channel. In the homomorphic version of Helios, aux contains a zero-knowledge proof that the vote is valid, that is 0 or 1. This avoids that a voter gives e.g. 100 votes to a candidate. In the mix-net variant of Helios, aux is empty. Provided that the voter is entitled to vote, the bulletin board adds the ballot $\{v\}^r_{\mathsf{pk}}, \mathsf{aux}$ to the public list. The voter should check that her ballot indeed appears on the public bulletin board.

The voter's behavior is described in Figure 1. It corresponds to the mix-net version but could be easily adapted to the homomorphic version. Note that this description contains assume and assert annotations that intuitively represent different states of the voter's process. These annotations are crucially used to state verifiability, cf Section 3.

The voting phase also includes an optional audit phase allowing the voter to audit her ballot instead of casting it. In that case, her ballot and the corresponding randomness are sent to a third party that checks whether the correct choice has been encrypted. We do not model here the auditing phase, since a precise characterization would probably require probabilistic reasoning, which goes beyond the scope of this paper.

Tallying phase. Once the voting phase is over, the bulletin board contains a list of ballots $\{v_1\}^{r_1}_{\mathsf{pk}}, \ldots, \{v_n\}^{r_n}_{\mathsf{pk}}$ (we omit the auxiliary data). We distinguish the two variants.

- *Homomorphic tally*. The ballots on the bulletin board are first homomorphically combined. Since $\{v\}^r_{\mathsf{pk}} * \{v'\}^{r'}_{\mathsf{pk}} = \{v+v'\}^{r+r'}_{\mathsf{pk}}$ anyone can compute the encrypted sum of the votes $\{\sum_{i=1}^n v_i\}^{r*}_{\mathsf{pk}}$. Then the trustees collaborate to decrypt this ciphertext. Their computation yields $\sum_{i=1}^n v_i$ and a proof of correct decryption.
- *Mix-net tally*. Ballots are shuffled and re-randomized, yielding $\{v_{i_1}\}^{r'_1}_{\mathsf{pk}}, \ldots, \{v_{i_n}\}^{r'_n}_{\mathsf{pk}}$ with a proof of correct permutation. This mixing is performed successively by several mixers. For privacy, the trust assumption is that as least one mix-net is honest (that is, will not leak the permutation). Then the trustees collaborate to decrypt each (re-randomize) ciphertext and provide a corresponding proof of correct decryption.

$$\begin{array}{lll} \mathsf{Voter}(id, v) = \mathsf{assume}\ \mathsf{Vote}(id, v); & & \mathsf{send}(\mathsf{net}, b); \\ \quad \mathsf{let}\ r = \mathsf{new}()\ \mathsf{in} & & \mathsf{let}\ bb = \mathsf{recv}(\mathsf{net})\ \mathsf{in} \\ \quad \mathsf{let}\ b = \mathsf{enc}(\mathsf{pk}, v, r)\ \mathsf{in} & & \mathsf{if}\ b \in bb\ \mathsf{then} \\ \quad \mathsf{assume}\ \mathsf{MyBallot}(id, v, b); & & \mathsf{assert}\ \mathsf{VHappy}(id, v, bb) \end{array}$$

Fig. 1. Modeling of a voter.

3 Verifiability

Verifiability is a key property in both electronic as well as paper-based voting systems. Intuitively, verifiability ensures that the announced result corresponds to the votes such as intended by the voters. Verifiability is typically split into several sub-properties.

- *Individual verifiability* ensures that a voter is able to check that her ballot is on the bulletin board.
- *Universal verifiability* ensures that any observer can verify that the announced result corresponds to the (valid) ballots published on the bulletin board.

Symbolic models provide a precise definition of these notions [43].

The overall goal of these two notions is to guarantee *end-to-end verifiability*: if a voter correctly follows the election process her vote is counted in the final result. In our terminology, *strong end-to-end verifiability* additionally guarantees that at most k dishonest votes have been counted, where k is the number of compromised voters. This notion of strong end-to-end verifiability includes the notion of what is called *eligibility verifiability* in [43]. For simplicity, we focus here on end-to-end verifiability.

We will now explain our modeling of individual, universal, and end-to-end verifiability. One of our contributions is a logical formalization of these properties that enables the use of off-the-shelf verification techniques, in our case a type system, at least in the case of individual and universal verifiability. End-to-end verifiability may be more difficult to type-check directly. Instead, we formally prove for the first time that individual and universal verifiability entail end-to-end verifiability provided that there are no "clash attacks" [44]. A clash attack typically arises when two voters are both convinced that the same ballot b is "their" own ballot. In that case, only one vote will be counted instead of two. The fact that individual and universal verifiability entail end-to-end verifiability has two main advantages. First, it provides a convenient proof technique: it is sufficient to prove individual and universal verifiability, which as we will show can be done with the help of a type-checker. Second, our results provide a better understanding of the relation between the different notions of verifiability.

Notations. Before presenting our formal model of verifiability we introduce a few notations. Voting protocols aim at counting the votes. Formally, a *counting function* is a function $\rho : \mathbb{V}^* \to R$, where $\mathbb{V}$ is the vote space and R the result space. A typical voting function is the number of votes received by each candidate. By a slight abuse of notation, we may consider $\rho(l)$ where l is a list of votes instead of a sequence of votes.

If l is a list, $\#l$ denotes the size of l and $l[i]$ refers to the ith element of the list. $a \in l$ holds if a is an element of l. Given $a_1, \ldots, a_n$, we denote by $\{\!|a_1, \ldots, a_n|\!\}$ the corresponding multiset. $\subseteq_m$ denotes multiset inclusion. Assume l_1, l_2 are lists; by a slight abuse of notation, we may write $l_1 \subseteq_m l_2$ where l_1, l_2 are viewed as multisets. We also write $l_1 =_m l_2$ if the two lists have the same multisets of elements.

In order to express verifiability and enforce it using a type system, we rely on the following assumptions:

- assume $\mathsf{Vote}(id, v, c)$ means that voter id intends to vote for c possibly using some credential c. This predicate should hold as soon as the voter starts to vote: he knows for whom he is going to vote.
- assume $\mathsf{MyBallot}(id, v, b)$ means that voter id thinks that ballot b contains her vote v. In case votes are sent in clear, b is simply the vote v itself. In the case of Helios, we have $b = \{v\}^r_{\mathsf{pk}}, \mathsf{aux}$. Typically, this predicate should hold as soon as the voter (or her computer) has computed the ballot.

An example of where and how to place these predicates for Helios can be found in Figure 1. The credential c is omitted since there is no use of credentials in Helios.

3.1 Individual verifiability

Intuitively, individual verifiability enforces that whenever a voter completes her process successfully, her ballot is indeed in the ballot box. Formally we define the predicate VHappy as follows:

$$\mathsf{assume}\ \mathsf{VHappy}(id, v, c, bb) \ \Leftrightarrow\ \mathsf{Vote}(id, v, c)\ \wedge\ \exists b \in bb.\ \mathsf{MyBallot}(id, v, b)$$

This predicate should hold whenever voter id has finished her voting process, and believes she has voted for v. At that point, it should be the case that the ballot box bb contains the vote v (in some ballot). We therefore annotate the voter function with the assertion $\mathsf{assert}\ \mathsf{VHappy}(id, v, c, bb)$. This annotation is generally the final instruction, see Figure 1 for the Helios example.

Definition 3 (Individual Verifiability). *A protocol with security annotations*

- $\mathsf{assume}\ \mathsf{Vote}(id, v, c)$, $\mathsf{assume}\ \mathsf{MyBallot}(id, v, b)$*;*
- *and* $\mathsf{assert}\ \mathsf{VHappy}(id, v, c, bb)$

as described above guarantees individual verifiability *if it is robustly safe.*

3.2 Universal verifiability

Intuitively, universal verifiability guarantees that anyone can check that the result corresponds to the ballots present in the ballot box. Formally, we assume a program $\mathsf{Judge}(bb, r)$ that checks whether the result r is valid w.r.t. ballot box bb. Typically, Judge does not use any secret and could therefore be executed by anyone. We simply suppose that Judge contains $\mathsf{assert}\ \mathsf{JHappy}(bb, r)$ at some point, typically when all the verification checks succeed. For Helios, the Judge program is displayed Figure 2. We first introduce a few additional predicates that we use to define the predicate JHappy.

Good sanitization. Once the voting phase is closed, the tallying phase proceeds in two main phases. First, some "cleaning" operation is performed in bb, e.g., invalid ballots (if any) are removed and duplicates are weeded, resulting in the sanitized valid bulletin board vbb. Intuitively, a good cleaning function should not remove ballots that correspond to honest votes. We therefore define the predicate $\mathsf{GoodSan}(bb, vbb)$ to hold if the honest ballots of bb are not removed from vbb.

$$\mathsf{assume}\ \mathsf{GoodSan}(bb, vbb) \Leftrightarrow \forall b.[(b \in bb\ \wedge\ \exists id, v.\mathsf{MyBallot}(id, v, b)) \Rightarrow b \in vbb]$$

$$\begin{aligned}\mathsf{Judge}(bb, r) = {} & \mathsf{let}\ vbb = \mathsf{recv}(\mathsf{net})\ \mathsf{in} \\ & \mathsf{let}\ zkp = \mathsf{recv}(\mathsf{net})\ \mathsf{in} \\ & \mathsf{if}\ vbb = \mathsf{removeDuplicates}(bb) \wedge \mathsf{check\_zkp}(zkp, vbb, r)\ \mathsf{then} \\ & \mathsf{assert}\ \mathsf{JHappy}(bb, r)\end{aligned}$$

Fig. 2. Judge function for Helios

Good counting. Once the ballot box has been sanitized, ballots are ready to be tallied. A good tallying function should count the votes "contained" in the ballots. To formally define that a vote is "contained" in a ballot, we consider a predicate $\mathsf{Wrap}(v, b)$ that is left undefined, but has to satisfy the following properties:

- any well-formed ballot b corresponding to some vote v satisfies: $\mathsf{MyBallot}(id, v, b) \Rightarrow \mathsf{Wrap}(v, b)$
- a ballot cannot wrap two distinct votes: $\mathsf{Wrap}(v_1, b) \wedge \mathsf{Wrap}(v_2, b) \Rightarrow v_1 = v_2$

If these two properties are satisfied, we say that Wrap is *voting-compliant*. For a given protocol, the definition Wrap typically follows from the protocol specification.

Example 1. In the Helios protocol, the Wrap predicate is defined as follows.

$$\mathsf{assume}\ \mathsf{Wrap}(v, b) \Leftrightarrow \exists r.\ \mathsf{Enc}(v, r, \mathsf{pk}, b)$$

where $\mathsf{Enc}(v, r, \mathsf{pk}, b)$ is a predicate that holds if b is the result of the encryption function called with parameters pk, v and r. It is easy to see that Wrap is voting-compliant and this can in fact be proved using a type-checker. It is sufficient to add the annotations

- assert $\mathsf{MyBallot}(id, v, b) \Rightarrow \mathsf{Wrap}(v, b)$ and
- assert $\forall v_1, v_2.\ \mathsf{Wrap}(v_1, b) \wedge \mathsf{Wrap}(v_2, b) \Rightarrow v_1 = v_2$

to the voter function (Figure 1) just after the MyBallot assumption. The second assertion is actually a direct consequence of our modeling of encryption which implies that a ciphertext cannot decrypt to two different plaintexts.

We are now ready to state when votes have been correctly counted: the result should correspond to the counting function ρ applied to the votes contained in each ballot. Formally, we define $\mathsf{GoodCount}(vbb, r)$ to hold if the result r corresponds to counting the votes of $rlist$, i.e., the list of votes obtained from the ballots in vbb'. The list vbb' is introduced for technical convenience and either denotes the list of valid votes vbb itself (in the homomorphic variant) or any arbitrary permutation of vbb (for mix-nets).

$$\begin{aligned}\mathsf{assume}\ \mathsf{GoodCount}(vbb, r) \Leftrightarrow \exists vbb', rlist.\ [\ & \#vbb = \#rlist \wedge vbb =_m vbb' \wedge \\ & \forall b, i.[vbb'[i] = b \\ & \Rightarrow \exists v.(\mathsf{Wrap}(v, b) \wedge (rlist[i] = v))] \wedge \\ & r = \rho(rlist)\ ]\end{aligned}$$

Note that the definition of GoodCount is parameterized by the counting function ρ of the protocol under consideration. We emphasize that for $\mathsf{GoodCount}(vbb, r)$ to hold, the sanitized bulletin board may only contain correctly wrapped ballots, i.e., we assume that the sanitization procedure is able to discard invalid ballots. In the case of mix-net-based Helios we therefore require that the sanitization discards any ballots that do not decrypt. This can for instance be achieved by requiring a zero knowledge proof that the

submitted bitstring is a correct ciphertext. We may however allow that a ballot decrypts to an invalid vote, as such votes can be discarded by the tallying function.

Universal verifiability. Finally, universal verifiability enforces that whenever the verification checks succeed (that is, the Judge's program reaches the JHappy assertion), then GoodSan and GoodCount should be guaranteed. Formally, we define the predicate

$$\mathsf{assume}\ \mathsf{JHappy}(bb, r) \Leftrightarrow \exists vbb.\ (\mathsf{GoodSan}(bb, vbb) \wedge \mathsf{GoodCount}(vbb, r))$$

and add the annotation $\mathsf{assert}\ \mathsf{JHappy}(bb, r)$ at the end of the judge function.

Definition 4 (Universal Verifiability). *A protocol with security annotations*
- $\mathsf{assume}\ \mathsf{MyBallot}(id, v, b)$*, and*
- $\mathsf{assert}\ \mathsf{JHappy}(bb, r)$

as described above guarantees universal verifiability *if it is robustly safe and the predicate* $\mathsf{Wrap}(v, b)$ *is voting-compliant.*

3.3 End-to-end verifiability

End-to-end verifiability is somehow simpler to express. End-to-end verifiability ensures that whenever the result is valid (that is, the judge has reached his final state), the result contains at least all the votes of the voters that have reached their final states. In other words, voters that followed the procedure are guaranteed that their vote is counted in the final result. To formalize this idea we define the predicate EndToEnd as follows:

$$\begin{aligned}\mathsf{assume\ EndToEnd} \Leftrightarrow\ & \forall bb, r, id_1, \ldots, id_n, v_1, \ldots, v_n, c_1, \ldots, c_n. \\ & (\mathsf{JHappy}(bb, r) \wedge \mathsf{VHappy}(id_1, v_1, c_1, bb) \wedge \ldots \wedge \mathsf{VHappy}(id_n, v_n, c_n, bb)) \\ & \Rightarrow \exists rlist.\ r = \rho(rlist) \wedge \{\!|v_1, \ldots, v_n|\!\} \subseteq_m rlist\end{aligned}$$

To ensure that this predicate holds we can again add a final assertion assert EndToEnd.

Definition 5 (End-to-End Verifiability). *A protocol with security annotations*
- $\mathsf{assume}\ \mathsf{Vote}(id, v, c)$*,* $\mathsf{assume}\ \mathsf{MyBallot}(id, v, b)$*;*
- *and* $\mathsf{assert}\ \mathsf{VHappy}(id, v, c, bb)$*,* $\mathsf{assert}\ \mathsf{JHappy}(bb, r)$*,* $\mathsf{assert}\ \mathsf{EndToEnd}$

as described above guarantees end-to-end verifiability *if it is robustly safe.*

For simplicity, we have stated end-to-end verifiability referring explicitly to a bulletin board. It is however easy to state our definition more generally by letting bb be any form of state of the protocol. This more general definition does not assume a particular structure of the protocol, as it is also the case in a previous definitions of end-to-end verifiability in the literature [46].

It can be difficult to directly prove end-to-end verifiability using a type-checker. An alternative solution is to show that it is a consequence of individual and universal verifiability. However, it turns out that individual and universal verifiability are actually not sufficient to ensure end-to-end verifiability. Indeed, assume that two voters id_1 and id_2 are voting for the same candidate v. Assume moreover that they have built the same ballot b. In case of Helios, this could be the case if voters are using a bad randomness generator. Then a malicious bulletin board could notice that the two ballots are identical

and could display only one of the two. The two voters would still be "happy" (they can see their ballot on the bulletin board) as well as the judge since the tally would correspond to the bulletin board. However, only one vote for v would be counted instead of two. Such a scenario has been called a clash attack [44].

We capture this property by the predicate NoClash defined as follows.

$$\mathsf{NoClash} \Leftrightarrow \forall id_1, id_2, v_1, v_2, b.\ \mathsf{MyBallot}(id_1, v_1, b)\ \wedge\ \mathsf{MyBallot}(id_2, v_2, b) \\ \Rightarrow\ id_1 = id_2\ \wedge\ v_1 = v_2$$

The assertion assert NoClash is then added after the assumption MyBallot.

Definition 6 (No Clash). *A protocol with security annotations*
- assume $\mathsf{MyBallot}(id, v, b)$ *and*
- assert NoClash

as described above guarantees no clash *if it is robustly safe.*

We can now state our result (proved in Appendix B) that no clash, individual, and universal verifiability entail end-to-end verifiability.

Theorem 2. *If a protocol guarantees individual and universal verifiability as well as no clash, then it satisfies end-to-end verifiability.*

3.4 Verifiability analysis of Helios

Using the F* type-checker (version 0.7.1-alpha) we have analyzed both the mix-net and homomorphic versions of Helios. The corresponding files can be found in [1]. The (simplified) model of the voter and judge functions is displayed in Figures 1 and 2.

Helios with mix-nets. Using F*, we automatically proved both individual and universal verifiability. As usual, we had to manually define the types of the functions, which crucially rely on refinement types to express the expected pre- and post-conditions. For example, for universal verifiability, one has to show that GoodSan and GoodCount hold whenever the judge agrees with the tally. For sanitization, the judge verifies that vbb is a sublist of bb, where duplicate ballots have been removed. Thus, the type-checker can check that the function $\mathsf{removeDuplicates}(bb)$ returns a list vbb whose type is a refinement stating that $x \in bb \Rightarrow x \in vbb$, which allows us to prove GoodSan. Regarding GoodCount, the judge verifies a zero-knowledge proof that ensures that any vote in the final result corresponds to an encryption on the sanitized bulletin board. Looking at the type of the check_zkp function we see that this information is again conveyed through a refinement of the boolean type returned by the function:

$$\mathsf{check\_zkp} : zkp : \mathsf{bytes} \rightarrow vbb : \mathsf{list\ ballot} \rightarrow res : \mathsf{result} \rightarrow b : \mathsf{bool}\{b = \mathsf{true} \Rightarrow \varphi\}$$

$$\text{where } \varphi \triangleq \exists vbb'.\ [\ \#vbb = \#res\ \wedge\ vbb =_m vbb' \wedge \\ \forall b, i.[vbb'[i] = b \Rightarrow\ \exists v, r.(\mathsf{Enc}(v, r, \mathsf{pk}, b)\ \wedge (res[i] = v))]\ ]$$

In the case where check_zkp returns true we have that the formula φ holds. The formula φ is similar to the GoodCount predicate (with ρ being the identity function for mix-net based Helios) except that it ensures that a ballot is an encryption, rather than a wrap.

This indeed reflects that the zero-knowledge proof used in the protocol provides exactly the necessary information to the judge to conclude that the counting was done correctly.

The *no clash* property straightforwardly follows from observing that the logical predicate $\mathsf{MyBallot}(id, v, b)$ is assumed only once in the voter's code, that each voter has a distinct id, and that, as argued for $\mathsf{Wrap}(v, b)$, the same ciphertext cannot decrypt to two different plaintexts. By Theorem 2, we can conclude that the mix-net version of Helios indeed satisfies end-to-end verifiability.

Type-checkers typically support lists with the respective functions (length, membership test, etc.). As a consequence, we prove individual and universal verifiability *for an arbitrary number of dishonest voters*, while only a fixed number of dishonest voters can typically be considered with other existing protocol verification tools.

Helios with homomorphic tally. The main difference with the mix-net version is that each ballot additionally contains a zero-knowledge proof, that ensures that the ballot is an encryption of either 0 or 1. The judge function also differs in the tests it performs. In particular, to check that the counting was performed correctly, the judge verifies a zero-knowledge proof that ensures that the result is the sum of the encrypted votes that are on the sanitized bulletin board. This ensures in turn that the result corresponds to the sum of the votes. Considering the "sum of the votes" is out of reach of classical automated protocol verification tools. Here, F* simply discharges the proof obligations involving the integer addition to the Z3 solver [50] which is used as a back-end.

Finally, as for the mix-net based version, we proved individual and universal verifiability using F*, while the *no clash* property relies on (the same) manual reasoning. Again, we conclude that end-to-end verifiability is satisfied using Theorem 2.

4 Privacy

The secrecy of a ballot is of vital importance to ensure that the political views of a voter are not known to anyone. Vote privacy is thus considered a fundamental and universal right in modern democracies.

In this section we review the definition of vote privacy based on observational equivalence [33] and present a type-based analysis technique to verify this property using RF*, an off-the-shelf type-checker. We demonstrate the usefulness of our approach by analyzing vote privacy in the homomorphic variant of Helios, which was considered so far out of the scope of automated verification techniques.

4.1 Definition of privacy

Observational equivalence. We first introduce the concept of observational equivalence, a central tool to capture indistinguishability properties. The idea is that two runs of the same program with different secrets should be indistinguishable for any opponent. The definition is similar to the natural adaption of the one presented in [37] to a deterministic, as opposed to probabilistic, setting.

Definition 7 (Observational Equivalence). *For all modules A, B we say that A and B are observationally equivalent, written $A \approx B$, iff they both export the same interface*

I and and for all opponents O that are well-typed w.r.t the interface I it holds that $A \cdot O \rightarrow^* M$ iff $B \cdot O \rightarrow^* M$ for all closed values M.

Here, $A \rightarrow^* N$ denotes that expression A eventually evaluates to value N, according to the semantic reduction relation.

Privacy. We adopt the definition of vote privacy presented in [42]. This property ensures that the link between a voter and her vote is kept secret. Intuitively, in the case of a referendum this can only be achieved if at least two honest voters exist, since otherwise all dishonest voters could determine the single honest voter's vote from the final tally by colluding. Furthermore, both voters must vote for different parties, thus counterbalancing each other's vote and ensuring that it is not known who voted for whom. Our definition of privacy thus assumes the existence of two honest voters Alice and Bob and two candidates v_1 and v_2. We say that a voting system guarantees privacy if a protocol run in which Alice votes v_1 and Bob votes v_2 is indistinguishable (i.e., observationally equivalent) from the protocol run in which Alice votes v_2 and Bob votes v_1.

In the following, we assume the voting protocol to be defined as $\mathsf{fun}\ (v_A, v_B) \rightarrow \mathsf{S}[\mathsf{Alice}(v_A), \mathsf{Bob}(v_B)]$. The two honest voters Alice and Bob are parameterized over their votes v_A and v_B. Here, $\mathsf{S}[\bullet, \bullet]$ describes a two-hole context (i.e., an expression with two holes), which models the behavior of the cryptographic library, the public bulletin board, and the election authorities (i.e., the surrounding system).

Definition 8 (Vote Privacy). *$P = \mathsf{fun}\ (v_A, v_B) \rightarrow \mathsf{S}[\mathsf{Alice}(v_A), \mathsf{Bob}(v_B)]$ guarantees vote privacy iff for any two votes v_1, v_2 it holds that $P(v_1, v_2) \approx P(v_2, v_1)$.*

4.2 RF*: A type system for observational equivalence properties

To prove privacy for voting protocols we rely on RF*, an off-the-shelf existing type-checker that can be used to enforce indistinguishability properties. RF* was introduced by Barthe et al. [13] and constitutes the relational extension of the F* type-checker [54]. The core idea is to let refinements reason about two runs (as opposed to a single one) of a protocol. Such refinements are called *relational refinements*. A relational refinement type has the form $x : T\{|F|\}$, where the formula F may capture the instantiation of x in the left run of the expression that is to be type-checked, denoted $\mathsf{L}\ x$, as well as the instantiation of x in the right run, denoted $\mathsf{R}\ x$. Formally, $A : x : T\{|F|\}$ means that whenever A evaluates to M_L and M_R in two contexts that provide well-typed substitutions for the free variables in A, then the formula $F\{^{M_L}/_{\mathsf{L}\ x}\}\{^{M_R}/_{\mathsf{R}\ x}\}$ is valid. We note that relational refinements are strictly more expressive than standard refinements. For instance, $x : \mathsf{bytes}\{H(x)\}$ can be encoded as $x : \mathsf{bytes}\{|H(\mathsf{L}\ x) \wedge H(\mathsf{R}\ x)|\}$. A special instance of relational refinement types is the so-called *eq-type*. Eq-types specify that a variable is instantiated to the same value in both the left and the right protocol run. Formally, $\mathsf{eq}\ T \triangleq x : T\{|\mathsf{L}\ x = \mathsf{R}\ x|\}$. The authors show how such types can be effectively used to verify both non-interference and indistinguishability properties.

4.3 Type-based verification of vote privacy

In the following, we show how to leverage the aforementioned technique to statically enforce observational equivalence and, in particular, vote privacy. The key observation

$\mathsf{Alice}\ v_A =$
$\mathsf{let}\ b_A = \mathsf{create\_ballot}_A(v_A)\ \mathsf{in}$
$\mathsf{send}(c_A, b_A)$

Fig. 3. Model of Alice

$\mathsf{Bob}\ v_B =$
$\mathsf{let}\ b_B = \mathsf{create\_ballot}_B(v_B)\ \mathsf{in}$
$\mathsf{send}(c_B, b_B)$

Fig. 4. Model of Bob

is that whenever a value M is of type eq bytes it can safely be published, i.e., given to the opponent. Intuitively, this is the case since in both protocol runs, this value will be the same, i.e., the opponent will not be able to observe any difference. Given that both runs consider the same opponent O, every value produced by the opponent must thus also be the same in both runs, which means it can be typed with eq bytes.

We denote typed interfaces that solely build on eq bytes by I_{eq} and following the above intuition state that if a voting protocol can be typed with such an interface, the two runs where (i) Alice votes v_1, Bob votes v_2 and (ii) Alice votes v_2, Bob votes v_1 are observationally equivalent, since no opponent will be able to distinguish them.

Theorem 3 (Privacy by Typing). *For all* $P = \mathsf{fun}\ (v_A, v_B) \to \mathsf{S}[\mathsf{Alice}(v_A), \mathsf{Bob}(v_B)]$ *and all* M, M', v_1, v_2 *such that* $M : x : \mathsf{bytes}\{|\mathsf{L}\ x = v_1 \wedge \mathsf{R}\ x = v_2|\}$ *and* $M' : x : \mathsf{bytes}\{|\mathsf{L}\ x = v_2 \wedge \mathsf{R}\ x = v_1|\}$ *it holds that if* $\emptyset \vdash P(M, M') \rightsquigarrow I_{\mathsf{eq}}$*, then* P *provides vote privacy.*

Modeling a protocol for privacy verification. We demonstrate our approach on the example of Helios with homomorphic encryption. For simplicity, we consider one ballot box that owns the decryption key dk and does the complete tabulation. An informal description of Alice and Bob's behavior is displayed in Figures 3 and 4, respectively. The voters produce the relationally refined ballots using the ballot creation functions $\mathsf{create\_ballot}_A, \mathsf{create\_ballot}_B$ respectively. The ballots b_A, b_B consist of the randomized homomorphic encryption of the votes and a zero-knowledge proof of correctness and knowledge of the encrypted vote. The ballots are then sent to the ballot box over secure https-connections c_A and c_B respectively.

The behavior of the ballot box is described in Figure 5. For the sake of simplicity, we consider the case of three voters. The ballot box receives the ballots of Alice and Bob and publishes them on the bulletin board. It then receives the ballot of the opponent and checks that the proofs of validity of all received ballots succeed. Furthermore, it checks that all ballots are distinct before performing homomorphic addition on the ciphertexts. The sum of the ciphertexts is then decrypted and published on the bulletin board.

Intuitively, all outputs on the network are of type eq bytes, since (i) all ballots are the result of an encryption that keeps the payload secret and thus gives the opponent no distinguishing capabilities, and (ii) the homomorphic sum of all ciphertexts $b_{ABO} = \{v_A + v_B + v_O\}_{\mathsf{pk}}$ is the same in both runs of the protocol up to commutativity. Indeed, $\mathsf{L}\ b_{ABO} = \{v_1 + v_2 + v_O\}_{\mathsf{pk}}$ and $\mathsf{R}\ b_{ABO} = \{v_2 + v_1 + v_O\}_{\mathsf{pk}} = \mathsf{L}\ b_{ABO}$.

However, since the application of the commutativity rule happens on the level of plaintexts, while the homomorphic addition is done one level higher-up on ciphertexts, we need to guide the type-checker in the verification process.

Sealing-based library for voting. While privacy is per se not defined by logical predicates, we rely on some assumptions to describe properties of the cryptographic library, such as homomorphism and validity of payloads, in order to guide the type-checker in

$$
\begin{array}{ll}
\mathsf{BB} = \mathsf{let}\ b_A = \mathsf{recv}(c_A)\ \mathsf{in} & \mathsf{match}\ \mathsf{check\_zkp}(b_B)\ \mathsf{with}\ \mathsf{true}\ \mathsf{then} \\
\quad \mathsf{let}\ b_B = \mathsf{recv}(c_B)\ \mathsf{in} & \mathsf{match}\ \mathsf{check\_zkp}(b_O)\ \mathsf{with}\ \mathsf{true}\ \mathsf{then} \\
\quad \mathsf{send}(\mathsf{net}, (b_A, b_B)); & \mathsf{match}\ (b_A \neq b_O \wedge b_A \neq b_B \wedge b_B \neq b_O)\ \mathsf{with}\ \mathsf{true}\ \mathsf{then} \\
\quad \mathsf{let}\ b_O = \mathsf{recv}(\mathsf{net})\ \mathsf{in} & \quad \mathsf{let}\ b_{AB} = \mathsf{add\_ballot}(b_A, b_B)\ \mathsf{in} \\
\quad \mathsf{if}\ \mathsf{check\_zkp}(b_A)\ \mathsf{true}\ \mathsf{then} & \quad \mathsf{let}\ b_{ABO} = \mathsf{add\_ballot}(b_{AB}, b_O)\ \mathsf{in} \\
& \quad \mathsf{let}\ \mathit{result} = \mathsf{dec\_ballot}(b_{ABO})\ \mathsf{in} \\
& \quad \mathsf{send}(\mathsf{net}, \mathit{result})
\end{array}
$$

Fig. 5. Model of the ballot box.

the derivation of eq-types. The (simplified) type of the sealing reference for homomorphic encryption with proofs of validity is given below:[7]

$$
\begin{array}{l}
m : \mathsf{bytes} * c : \mathsf{eq\ bytes}\{|\mathsf{Enc}(m, c) \wedge \mathsf{Valid}(c) \wedge \\
(\mathsf{FromA}(c) \vee \mathsf{FromB}(c) \vee (\mathsf{FromO}(c) \wedge \mathsf{L}\ m = \mathsf{R}\ m))|\}
\end{array}
$$

Here, predicates FromA, FromB, FromO are used to specify whether an encryption was done by Alice, Bob or the opponent, while $\mathsf{Enc}(m, c)$ states that c is the ciphertext resulting from encrypting m and $\mathsf{Valid}(c)$ reflects the fact that the message corresponds to a valid vote, i.e., a validity proof for c can be constructed. Note that if a ballot was constructed by the opponent, the message stored therein must be the same in both runs ($\mathsf{L}\ m = \mathsf{R}\ m$), i.e., the message must have been of type eq bytes. These logical predicates are assumed in the sealing functions used by Alice, Bob, and the opponent, respectively. These functions, used to encode the public key, share the same code, and in particular they access the same reference, and only differ in the internal assumptions.

Similarly, there exist three ballot creation functions $\mathsf{create\_ballot}_A$, $\mathsf{create\_ballot}_B$, and $\mathsf{create\_ballot}_O$, used by Alice, Bob and the opponent, respectively, only differing in their refinements and internal assumptions. Their interfaces are listed below:

$$
\begin{array}{ll}
\mathsf{create\_ballot}_A : & m : x : \mathsf{bytes}\{|\mathsf{L}\ x = v_1 \wedge \mathsf{R}\ x = v_2|\} \rightarrow \\
& \quad c : \mathsf{eq\ bytes}\{|\mathsf{Enc}(m, c) \wedge \mathsf{FromA}(c)|\} \\
\mathsf{create\_ballot}_B : & m : x : \mathsf{bytes}\{|\mathsf{L}\ x = v_2 \wedge \mathsf{R}\ x = v_1|\} \rightarrow \\
& \quad c : \mathsf{eq\ bytes}\{|\mathsf{Enc}(m, c) \wedge \mathsf{FromB}(c)|\} \\
\mathsf{create\_ballot}_O : & = \mathsf{eq\ bytes} \rightarrow \mathsf{eq\ bytes}
\end{array}
$$

Notice that, as originally proposed in [13], the result of probabilistic encryption (i.e., the ballot creation function) is given an eq bytes type, reflecting the intuition that there always exist two randomnesses, which are picked with equal probability, that make the ciphertexts obtained by encrypting two different plaintexts identical, i.e., probabilistic encryption does not leak any information about the plaintext.

The interfaces for the functions dec_ballot, check_zkp, add_ballot for decryption, validity checking of the proofs, and homomorphic addition are listed below. The public interfaces for the latter two functions, built only on eq-types, are exported to the

[7] The actual library includes *marshaling* operations, which we omit for simplicity.

opponent. The interface for decryption is however only exported to the ballot box.

$$\begin{aligned}
&\mathsf{dec_ballot} : c : \mathsf{eq\ bytes} \to \mathsf{privkey} \to m : \mathsf{bytes}\{|\forall z.\mathsf{Enc}(z,c) \Rightarrow z = m|\}\\
&\mathsf{check_zkp} : c : \mathsf{eq\ bytes} \to b : \mathsf{bool}\{|b = \mathsf{true} \Rightarrow (\mathsf{Valid}(c) \wedge (\exists m.\mathsf{Enc}(m,c)) \wedge\\
&\qquad\qquad (\mathsf{FromA}(c) \vee \mathsf{FromB}(c) \vee (\mathsf{FromO}(c) \wedge \mathsf{L}\ m = \mathsf{R}\ m)))|\}\\
&\mathsf{add_ballot} : c : \mathsf{eq\ bytes} \to c' : \mathsf{eq\ bytes} \to\\
&\qquad c'' : \mathsf{eq\ bytes}\{|\forall m, m'.(\mathsf{Enc}(m,c) \wedge \mathsf{Enc}(m',c')) \Rightarrow \mathsf{Enc}(m+m', c'')|\}
\end{aligned}$$

Intuitively, the type returned by decryption assures that the decryption of the ciphertext corresponds to the encrypted message. The successful application of the validity check on ballot c proves that the ballot is a valid encryption of either v_1 or v_2 and that it must come from either Alice, Bob, or the opponent. In the latter case it must be the same in both runs. When homomorphically adding two ciphertexts, the refinement of function add_ballot guarantees that the returned ciphertext contains the sum of the two. The implementation of dec_ballot is standard and consists of the application of the unsealing function. The implementation of check_zkp follows the approach proposed in [10,11]: in particular, the zero-knowledge proof check function internally decrypts the ciphertexts and then checks the validity of the vote, returning a boolean value. Finally, the add_ballot homomorphic addition function is implemented in a similar manner, internally decrypting the two ciphertexts and returning a fresh encryption of the sum of the two plaintexts.

Global assumptions. In order to type-check the complete protocol we furthermore rely on three natural assumptions:

- A single ciphertext only corresponds to one plaintext, i.e., decryption is a function:
  $\mathsf{assume}\ \forall m, m', c.(\mathsf{Enc}(m,c) \wedge \mathsf{Enc}(m',c)) \Rightarrow m = m'$
- Alice and Bob only vote once:
  $\mathsf{assume}\ \forall c, c'.(\mathsf{FromA}(c) \wedge \mathsf{FromA}(c')) \Rightarrow c = c'$
  $\mathsf{assume}\ \forall c, c'.(\mathsf{FromB}(c) \wedge \mathsf{FromB}(c')) \Rightarrow c = c'$

Modeling revoting would require a bit more work. Revoting requires some policy that explains which ballot is counted, typically the last received one. In that case, we would introduce two types depending on whether the ballot is really the final one (there is a unique final one) or not.

4.4 Privacy analysis of Helios

Using the RF* type-checker (version 0.7.1-alpha) we have proved privacy for the homomorphic version of Helios. The corresponding files can be found in [1]. Our implementation builds on the above defined cryptographic library and global assumptions as well as Alice, Bob, and the ballot box BB as defined in the previous section.

We briefly give the intuition why the final tally $result = \mathsf{dec\_ballot}(b_{ABO})$ can be typed with type eq bytes, i.e., why both runs return the same value by explaining the typing of the ballot box BB.

- The ballots b_A, b_B, b_O that are received by the ballot box must have the following types (by definition of the corresponding ballot creation functions):

$$\begin{aligned}
&b_A : c : \mathsf{eq\ bytes}\{|\mathsf{Enc}(v_A, b_A) \wedge \mathsf{FromA}(b_A)|\}\\
&b_B : c : \mathsf{eq\ bytes}\{|\mathsf{Enc}(v_B, b_B) \wedge \mathsf{FromB}(b_B)|\}\\
&b_O : \mathsf{eq\ bytes}
\end{aligned}$$

– Adding b_A and b_B together using add_ballot thus yields that the content of the combined ciphertext corresponds to v_A+v_B and in particular, due to commutativity, this sum is the same in both protocol runs.
– The most significant effort is required to show that the payload v_O contained in b_O is indeed of type eq bytes, i.e., $\mathsf{L}\ v_O = \mathsf{R}\ v_O$, meaning the sum of $v_A + v_B + v_O$ is the same in both runs. Intuitively, the proof works as follows: From checking the proof of b_O it follows that there exists v_O such that $\mathsf{Enc}(v_O, b_O) \wedge (\mathsf{FromA}(b_O) \vee \mathsf{FromB}(b_O) \vee (\mathsf{FromO}(b_O) \wedge \mathsf{L}\ v_O = \mathsf{R}\ v_O))$. From checking the distinctness of the ciphertexts we furthermore know that $b_A \neq b_O \neq b_C$. Given $\mathsf{FromA}(b_A)$ and $\mathsf{FromB}(b_B)$, the second and third global assumptions imply that neither $\mathsf{FromA}(b_O)$ nor $\mathsf{FromB}(b_O)$ hold true. Thus, it must be the case that $\mathsf{FromO}(b_O) \wedge \mathsf{L}\ v_O = \mathsf{R}\ v_O$.

5 Conclusion

In this paper we proposed a novel approach, based on type-checking, for analyzing e-voting systems. It is based on a novel logical theory which allows to verify both verifiability and vote privacy, two fundamental properties of election systems. We were able to put this theory into practice and use an off-the-shelf type-checker to analyze the mix-net-, as well as homomorphic tallying-based versions of Helios, resulting in the first automated verification of Helios with homomorphic encryption. Indeed, the fact that the type-checker can discharge proof obligations on the algebraic properties of homomorphic encryption to an external solver is one of the strengths of this approach. Providing the right typing annotations constitutes the only manual effort required by our approach: in our analysis this was, however, quite modest, in our analysis, thanks to the support for type inference offered by RF*.

As a next step we are planning to extend our theory to handle *strong* end-to-end verifiability, which additionally takes the notion of eligibility verifiability into account. This stronger notion is not satisfied by Helios, but the Helios-C protocol [28] was designed to achieve this property, providing an interesting case study for our approach.

We also plan to apply our approach to the e-voting protocol recently deployed in Norway for a political election. The privacy of this protocol was analyzed in [30], but due to the algebraic properties of the encryption, the proof was completely done by hand. Our approach looks promising to enable automation of proofs for this protocol.

Acknowledgements. This work was supported by the German research foundation (DFG) through the Emmy Noether program, the German Federal Ministry of Education and Research (BMBF) through the Center for IT-Security, Privacy and Accountability (CISPA), the European Research Council under the EU 7th Framework Programme (FP7/2007-2013) / ERC grant agreement no 258865 and the ANR project Sequoia ANR-14-CE28-0030-01.

References

1. http://sps.cs.uni-saarland.de/voting
2. Adida, B.: Helios: Web-based Open-Audit Voting. In: USENIX'08. pp. 335–348 (2008)
3. Arapinis, M., Cortier, V., Kremer, S., Ryan, M.D.: Practical Everlasting Privacy. In: POST'13. pp. 21–40 (2013)

4. Armando, A., Carbone, R., Compagna, L., Cuellar, J., Tobarra, L.: Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google Apps. In: FMSE'08. pp. 1–10 (2008)
5. Backes, M., Cortesi, A., Focardi, R., Maffei, M.: A calculus of challenges and responses. In: FMSE'07. pp. 51–60 (2007)
6. Backes, M., Cortesi, A., Maffei, M.: Causality-based abstraction of multiplicity in security protocols. In: CSF'07. pp. 355–369 (2007)
7. Backes, M., Grochulla, M.P., Hritcu, C., Maffei, M.: Achieving security despite compromise using zero-knowledge. In: CSF'09. pp. 308–323 (2009)
8. Backes, M., Hriţcu, C., Maffei, M.: Automated Verification of Remote Electronic Voting Protocols in the Applied Pi-calculus. In: CSF'08. pp. 195–209 (2008)
9. Backes, M., Lorenz, S., Maffei, M., Pecina, K.: The CASPA Tool: Causality-Based Abstraction for Security Protocol Analysis. In: CAV'08. pp. 419–422 (2008)
10. Backes, M., Maffei, M., Hriţcu, C.: Union and Intersection Types for Secure Protocol Implementations. In: TOSCA'11. pp. 1–28 (2011)
11. Backes, M., Maffei, M., Hriţcu, C.: Union and Intersection Types for Secure Protocol Implementations. JCS pp. 301–353 (2014)
12. Backes, M., Maffei, M., Unruh, D.: Computationally Sound Verification of Source Code. In: CCS'10. pp. 387–398 (2010)
13. Barthe, G., Fournet, C., Grégoire, B., Strub, P., Swamy, N., Béguelin, S.Z.: Probabilistic Relational Verification for Cryptographic Implementations. In: POPL'14. pp. 193–206 (2014)
14. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement Types for Secure Implementations. TOPLAS 33(2), 8 (2011)
15. Bernhard, D., Cortier, V., Pereira, O., Smyth, B., Warinschi, B.: Adapting Helios for provable ballot secrecy. In: ESORICS'11. pp. 335–354 (2011)
16. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: CSFW'01. pp. 82–96 (2001)
17. Blanchet, B., Abadi, M., Fournet, C.: Automated Verification of Selected Equivalences for Security Protocols. JLAP 75(1), 3–51 (Feb–Mar 2008)
18. Bortolozzo, M., Centenaro, M., Focardi, R., Steel, G.: Attacking and Fixing PKCS#11 Security Tokens. In: CCS'10. pp. 260–269 (2010)
19. Bugliesi, M., Focardi, R., Maffei, M.: Analysis of typed-based analyses of authentication protocols. In: CSFW'05. pp. 112–125. IEEE (2005)
20. Bugliesi, M., Focardi, R., Maffei, M.: Dynamic types for authentication. JCS 15(6), 563–617 (2007)
21. Bugliesi, M., Calzavara, S., Eigner, F., Maffei, M.: Resource-aware Authorization Policies in Statically Typed Cryptographic Protocols. In: CSF'11. pp. 83–98 (2011)
22. Bugliesi, M., Calzavara, S., Eigner, F., Maffei, M.: Logical Foundations of Secure Resource Management in Protocol Implementations. In: POST'13. pp. 105–125 (2013)
23. Bugliesi, M., Focardi, R., Maffei, M.: Principles for entity authentication. In: PSI'03. pp. 294–306 (2003)
24. Bugliesi, M., Focardi, R., Maffei, M.: Authenticity by tagging and typing. In: FMSE'04. pp. 1–12 (2004)
25. Chadha, R., Ciobâcă, Ş., Kremer, S.: Automated verification of equivalence properties of cryptographic protocols. In: ESOP'12. pp. 108–127 (2012)
26. Cheval, V.: APTE: an Algorithm for Proving Trace Equivalence. In: TACAS'14. pp. 587–592 (2014)
27. Cohen, J.D., Fischer, M.J.: A Robust and Verifiable Cryptographically Secure Election Scheme. In: FOCS'85. pp. 372–382 (1985)
28. Cortier, V., Galindo, D., Glondu, S., Izabachène, M.: Election Verifiability for Helios under Weaker Trust Assumptions. In: ESORICS'14. pp. 327–344 (2014)

29. Cortier, V., Smyth, B.: Attacking and fixing Helios: An analysis of ballot secrecy. In: CSF'11. pp. 297–311 (2011)
30. Cortier, V., Wiedling, C.: A formal analysis of the Norwegian E-voting protocol. In: POST'12. pp. 109–128 (2012)
31. Cramer, R., Gennaro, R., Schoenmakers, B.: A Secure and Optimally Efficient Multi-Authority Election Scheme. In: EUROCRYPT'97. pp. 103–118 (1997)
32. Cremers, C.: The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In: CAV'08. pp. 414–418 (2008)
33. Delaune, S., Kremer, S., Ryan, M.D.: Verifying privacy-type properties of electronic voting protocols. JCS 17(4), 435–487 (2009)
34. Eigner, F., Maffei, M.: Differential Privacy by Typing in Security Protocols. In: CSF'13. pp. 272–286. IEEE (2013)
35. Estehghari, S., Desmedt, Y.: Exploiting the Client Vulnerabilities in Internet E-voting Systems: Hacking Helios 2.0 as an Example. In: EVT/WOTE'10 (2010)
36. Focardi, R., Maffei, M.: Types for security protocols. In: Formal Models and Techniques for Analyzing Security Protocols. IOS (2011)
37. Fournet, C., Kohlweiss, M., Strub, P.: Modular Code-Based Cryptographic Verification. In: CCS'11. pp. 341–350 (2011)
38. Fujioka, A., Okamoto, T., Ohta, K.: A Practical Secret Voting Scheme for Large Scale Elections. In: AUSCRYPT'92. pp. 244–251 (1992)
39. Gordon, A.D., Jeffrey, A.: Types and effects for asymmetric cryptographic protocols. JCS 12(3), 435–484 (2004)
40. IACR. Elections page at http://www. iacr.org/elections/
41. Juels, A., Catalano, D., Jakobsson, M.: Coercion-Resistant Electronic Elections. In: Towards Trustworthy Elections: New Directions in Electronic Voting, pp. 37–63. Springer (2010)
42. Kremer, S., Ryan, M.D.: Analysis of an Electronic Voting Protocol in the Applied Pi Calculus. In: ESOP'05. pp. 186–200 (2005)
43. Kremer, S., Ryan, M.D., Smyth, B.: Election verifiability in electronic voting protocols. In: ESORICS'10. pp. 389–404 (2010)
44. Küsters, R., Truderung, T., Vogt, A.: Clash Attacks on the Verifiability of E-Voting Systems. In: S&P'12. pp. 395–409 (2012)
45. Küsters, R., Truderung, T., Vogt, A.: A Game-Based Definition of Coercion-Resistance and its Applications. In: CSF'10. pp. 122–136 (2010)
46. Küsters, R., Truderung, T., Vogt, A.: Accountabiliy: Definition and Relationship to Verifiability. In: CCS'10. pp. 526–535 (2010)
47. Küsters, R., Truderung, T., Vogt, A.: Verifiability, Privacy, and Coercion-Resistance: New Insights from a Case Study. In: S&P'11. pp. 538–553 (2011)
48. Lowe, G.: Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In: TACAS'96. pp. 147–166. Springer (1996)
49. Morris, J.: Protection in Programming Languages. CACM 16(1), 15–21 (1973)
50. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS'08. pp. 337–340 (2008)
51. Rivest, R.L.: The threeballot voting system (2006), unpublished draft
52. Smyth, B., Ryan, M.D., Kremer, S., Kourjieh, M.: Towards automatic analysis of election verifiability properties. In: ARSPA-WITS'10. pp. 165–182 (2010)
53. Sumii, E., Pierce, B.: A Bisimulation for Dynamic Sealing. TCS 375(1-3), 169–192 (2007)
54. Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bhargavan, K., Yang, J.: Secure Distributed Programming with Value-Dependent Types. In: ICFP'11. pp. 266–278 (2011)
55. Wagner, D., Schneier, B.: Analysis of the SSL 3.0 protocol. In: USENIX Workshop on Electronic Commerce. pp. 29–40 (1996)

a, b, c	label
x, y, z	variable
$h ::= \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{fold}$	constructor
F, G	first-order formula
$M, N ::=$	value
$\quad x, y, z$	variable
$\quad ()$	unit
$\quad (M, N)$	pair
$\quad h\ M$	construction
$\quad \mathsf{fun}\ x \to A$	function (x bound in A)
$\quad \mathsf{read}_a$	reference read
$\quad \mathsf{write}_a$	reference write
$A, B ::=$	expression
$\quad M$	value
$\quad M\ N$	application
$\quad \mathsf{let}\ x = A\ \mathsf{in}\ B$	let (x bound in B)
$\quad \mathsf{let}\ (x, y) = M\ \mathsf{in}\ A$	split (x, y bound in A)
$\quad \mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$	constructor match (x bound in A)
$\quad \mathsf{ref}\ M$	reference creation
$\quad \mathsf{assume}\ F$	assumption
$\quad \mathsf{assert}\ F$	assertion

$\mathsf{true} \triangleq \mathsf{inl}\ ()$ and $\mathsf{false} \triangleq \mathsf{inr}\ ()$

Table 2. Syntax of values and expressions

A Computational RCF

The syntax of Computational RCF is presented in Table 2. The semantics is standard: we refer to [37] for the complete formalization.

B Proof of Theorem 2

We prove here that no clash, individual, and universal verifiability entails end-to-end verifiability. Assume no clash, individual, and universal verifiability. We wish to show end-to-end verifiability, that is

$$\begin{aligned}&\forall bb, r, id_1, \ldots, id_n, v_1, \ldots, v_n, c_1, \ldots, c_n.\\ &\quad (\mathsf{JHappy}(bb, r) \wedge \mathsf{VHappy}(id_1, v_1, c_1, bb) \wedge \ldots \wedge \mathsf{VHappy}(id_n, v_n, c_n, bb))\\ &\qquad \Rightarrow \exists rlist.\ r = \rho(rlist) \wedge \{\!|v_1, \ldots, v_n|\!\} \subseteq_m rlist\end{aligned}$$

Therefore, assume $\mathsf{JHappy}(bb, r), \mathsf{VHappy}(id_1, v_1, c_1, bb), \ldots, \mathsf{VHappy}(id_n, v_n, c_n, bb)$ for some $bb, r, id_1, \ldots, id_n, v_1, \ldots, v_n, c_1, \ldots, c_n$.

– Thanks to universal verifiability, we deduce that there exists vbb such that

$$\mathsf{GoodSan}(bb, vbb) \tag{1}$$

$$\mathsf{GoodCount}(vbb, r) \tag{2}$$

– By definition of $\mathsf{GoodCount}$, $\mathsf{GoodCount}(vbb, r)$ implies that there exist vbb' and $rlist$ such that

$$vbb =_m vbb' \tag{3}$$

$$\forall b, i.[vbb'[i] = b \Rightarrow \exists v.(\mathsf{Wrap}(v, b) \wedge (rlist[i] = v))] \tag{4}$$

$$r = \rho(rlist) \tag{5}$$

– Now, $\mathsf{VHappy}(id_k, v_k, c_k, bb)$ and individual verifiability imply $\mathsf{Vote}(id_k, v_k, c_k)$ and there is $b_k \in bb$ such that $\mathsf{MyBallot}(id_k, v_k, b_k)$. Since Wrap is voting compliant, we deduce

$$\mathsf{Wrap}(v_k, b_k) \tag{6}$$

We then deduce $b_k \in vbb$ from the definition of $\mathsf{GoodSan}(bb, vbb)$ and the fact that $b_k \in bb$ and $\mathsf{MyBallot}(id_k, v_k, b_k)$. (3) also ensures $b_k \in vbb'$.

– $\mathsf{NoClash}$ ensures that the b_k are pairwise-disjoint (since the id_k are pairwise disjoint).
– Since the b_k are pairwise-disjoint and $b_k \in vbb'$, there exist pairwise-disjoint i_k such that $vbb'[i_k] = b_k$. We then deduce from (4) that there exists v'_k such that $\mathsf{Wrap}(v'_k, b_k)$ and $rlist[i_k] = v'_k$.
– $\mathsf{Wrap}(v'_k, b_k)$ and $\mathsf{Wrap}(v_k, b_k)$ (Equation 6) imply $v_k = v'_k$ since Wrap is voting compliant.

Therefore, we have shown that $\{\!|v_1, \ldots, v_n|\!\} \subseteq_m rlist$ and $r = \rho(rlist)$ (Equation 5), which concludes the proof.