INRIA Nancy Grand est

# Modeling e-voting Protocols in Fstar

*Author:*
Akash Garg

*Supervisor:*
Dr. Steve Kremer

July 10, 2015

# Acknowledgement

Simply put, I could not have done this work without the lots of help I received from the entire CASSIS Research Team at INRIA. The work culture at INRIA is really motivating. Everybody is such friendly and cheerful companion here that work stress never comes in way.

I would like to express my gratitude towards Dr. Steve Kremer for providing the nice ideas to work upon. Not only did he guided about my project but he always encouraged me to discuss ideas with him, corrected my mistakes and also helped me so as to make my adaptation smooth. Without those discussion it would have been impossible to complete the work in such a manner.

I would also like thank all the members of the CASSIS team for making my stay so pleasant in France.

Author

# Preface

This report documents the work done during the summer internship at CASSIS Research Team, Inria Nancy Grand Est , France under the supervision of Dr. STEVE KREMER. The report first gives all the back ground information about security protocols and their properties and verified functional programming in Fstar. Then it presents the two protocols Helios and Civitas which are the main focus of the project. It shall give a brief introduction to the two protocols and their theoretical implementation and then we will go technical details of the security properties in Fstar.

The various aspects of the protocols have been explained in an abstract sense so as to make it easier to understand. Also a good introduction has been given about Fstar which is used later for verifying the properties.

I have tried my best to present technical details in as simple manner as possible. I hope to succeed in my attempt. Any questions or suggestion about this work are most welcome.

Akash Garg
akash.garg@inria.fr

# 1 Introduction

Elections are unique. They change the fate of nations, influence participation and activism in politics, deeply affect the lives and attitude of citizens. Due to huge importance of elections in our society, not only they should work but people should believe that they work. This leads us to the biggest question in elections, ethics.

In the recent past there have been many changes to the process of elections and changes continue to take place. One of the most influential change which is underway is e-voting over the internet. E-voting is believed to significantly increase voter turnout, efficiency of casting, counting and declaration of results. Also it provide new ways of verification like voter verification (each voter can verify whether his vote has been counted or not) and universal verification (anyone can verify the entire election process) which were not possible in traditional methods.

All the features of e-voting have been achieved by using different types of protocols. These protocols provide all the advantages that e-voting enjoys. Security is one of the key concerns in e-voting. Various cryptographic protocols are used to make the election process secure. Analyzing these protocols is a very difficult task. Since manual analysis is error prone so is not considered for analyzing them. Most of the protocols that have been designed over the years have some form of vulnerabilities associated with them. Even the carefully designed standard like SSL/TLS, PKCS #11 have vulnerabilities in them. This lays the foundations and needs of formal analysis of security [7].

One of the main lines of research is the development of type systems for cryptographic protocol analysis [7]. Refinement type systems works on the principle of formally verifying each part of the program like Hoare triple. It tracks pre and post conditions for every fragment and thus enforces various trace properties such as authentication, classical authorization policies as well as linear authorization policies. Thus various tools are being developed based on type systems for analyzing protocols.

In this report we focus on modeling two very important protocols, Helios and Civitas, in Fstar, a type based strict functional programming language which is used for verification of the security properties of the protocols. We first give a background about Fstar in Section 2 . We introduce some programming features of Fstar and some of the standard library functions in Sections 2.1 to 2.4 . Then we give some example programs for proving some properties in Section 2.5 . Next we start with out first protocol Helios in Sec-

tion 3 . We first describe the working of the protocol along with the details of all phases and methods used for voting and tabulation. We also describe how Helios is verified and then we model the same in Fstar in section 3.2 & 3.3 . In section 4 we focus on Civitas describing the functioning and implementation. We also describe how Civitas is coercion resistant and privacy preserving in Section 4.1 . We describe the implementation of Civitas and working of the tabulation phase in Section 4.2 & 4.3 . Finally we model the verifiability of Civitas in Fstar in Section 4.6 . We end the report with conclusions and suggestions for future work in Section 5 .

# 2 Fstar

Fstar is a verification oriented programming language. It has most of the properties similar to that of ML family of languages. Inspite of many similarities its type system is richer than ML's. It is strict and higher-order programming language. It verifies and enforces the constraints of types on the code - type checking. When a code is type checked then every assertion can by syntactically entailed from previously made assumptions. Its quite similar to that of propositional logic.

The usage of Fstar is wide spread comprising construction of verified programs for security protocols, intermediate verification language (programs from other languages can be translated to Fstar) etc. Even parts of Fstar compiler are written in Fstar itself. Some of the main languages from which translations have been done are JavaScript, F7 and DKAL. [3]

The entire syntax of Fstar can be found in the tutorial available at http://rise4fun.com/FStar/tutorialcontent/guide [2]

## 2.1 Fstar Program Structure

An Fstar program is composed of several modules. Each module begins with a module declaration. A module can be thought of as a class in java. A module includes a number of definitions of functions and data types and optionally a main expression.

Program expressions are differentiated based on their properties. Fstar uses types to describe properties of program expressions. Further to define properties of types Fstar uses the kinds. So each expression has a type and each type has a kind. The default kind is S [2]. In Fstar we also have

dependent function types. In general these types have the form x:t → t' where x names the t-typed formal parameter and the variable x is in scope to the right of the arrow i.e., in the type t'. In other words, the return type t' of the function depends on the value of the formal parameter. An example of a function with dependent return type is is

$$val \quad twice : x : int \rightarrow y : int\{y = x + x\}$$

the signature of the function shows that y depends on x for its value which is twice that of x.

## 2.2 Standard Libraries

Fstar has several standard libraries which define many important constructs. The most commonly used is the Prims library. It defines many of the standard constructs of Fstar like unit, int, nat, string, etc. Fstar also provides many other libraries like List, Array, Map, Heap, etc. They all define important data structures used in Fstar.

A comprehensive list of all libraries can be found at
https://github.com/FStarLang/FStar/tree/master/lib [4]

## 2.3 Refinement Types

Refinement types are one of the most important and most useful feature of Fstar. Suppose we define a list data type and kept the length as int. But we know that length has to be non-negative. In order to define another property of length, it is a bad idea to define the entire list again. So we resort to refining the type list. We use the entire list type and add a property that length has to be non-negative. Thus list with non-negative length is a refinement of the list with length int. Similarly we can define natural numbers from integers and so on.

The standard syntax for type is

$$\text{Y = x:t\{t'\}}$$

where x is type's name, t is the data type and t' is the refinement property.
e.g type nat = i:int{0 <=i}

## 2.4 Subtyping Relations in Fstar

Refinement types in Fstar are equipped with a built-in subtyping relation. Let us take an expression x:t{t'}. Clearly its type is refined from type t with the property t'. But since it is just a refinement it has all the properties of type t and it is an expression of type t as well. On the other hand, in a typing context $\tau$, a value v of type t can be viewed as a value of type x:t{t'} if the proposition t'[v/x] is derivable from the assumptions on $\tau$. We write $\tau$ | - t <: t' when t is a subtype of t' in a context $\tau$ [2].

## 2.5 Programs in Fstar

We illustrate some of the features of Fstar through some simple programs about lists and integers.

**Summing an integer list:**

```
val sum: l:list int → Tot int
   let rec sum l=
   match l with
       |[]→0
       |hd::tl→hd + sum tl
```

The above function is returns the sum of all the elements of a list of integers. The program is recursive as the length of the list is not known. The "rec" keyword is used to make a program recursive. Also we used the "Tot" effect for the return type of the program. If we don't use the Tot keyword then the return type is inferred to have type "int" but on using Tot keyword it guarantees that the program will not halt and will definitely evaluate to a type "int".

**Checking for membership:**

```
val mem: l:list 'a → x: 'a → Tot bool
   let rec mem: l x =
      match l with
          |[]→false
          |hd::tl→if hd = x then true
             else mem tl x
```

This program is used to check whether a particular element is present in a list or not. The return type is bool. The " 'a" is used to denote a variable type list. It is similar to template functions in object oriented programming language like C++. The " 'a" indicates that a list made of any data type can be used.

**Extracting nth element from the list:**

```
val nth: l:list 'a  → n:int{0 ≤ n ∧ n < length l}
   → Tot (x:'a)
      let rec nth l n =
         match l with
            |hd::tl → if n = 0 then hd
               else nth tl (n−1)
```

This function extracts the nth element from a given list of variable type. The integer n is given as a parameter to the function. But since n is an integer so it can take any value. We want the function to be well defined. For it to be well defined n has to take values between 0 and length of the list (0 included). So we use a refinement type for this. The type of n has been refined from a general integer to an integer that can only take value from a finite set and it makes the function well defined. It is also a dependent type on the length of list.

**Checking for permutation:**

```
val permut: l1:list 'a → l2:list 'a → l3:list 'a
 → Tot bool
    let rec permut l1 l2 l3=
        match l2 with
            |[]→if length l1=length l3 then true else false
            |hd::tl→if count hd l1 = count hd l3
                  then permut l1 tl l3 else false

val ispermutation: l1:list 'a → l2:list 'a → Tot bool
        let ispermutation l1 l2=
            permut l1 l1 l2
```

We formally say that a list $l_1$ is a permutation of a list $l_3$ if the number of occurrences of each element of $l_1$ in $l_1$ equals that of $l_3$ and their

lengths are equal. So to check if two lists are permutations of each other we have two function. First function takes three lists as input and checks that count of each element from the second list is the same in first and third and their lengths are equal. The second function calls the first function with two inputs as same list. Combined together both ensures that the lists are permutation of each other.

# 3 Helios

## 3.1 Protocol Definition

Helios is a verifiable and privacy preserving voting system. It is used at many places for small scale elections where the problem of coercion is not a major concern. Some of the main places where it is used are student elections at the University of Louvain-la-Neuve or at Princeton. It has also been used by IACR to elect its board since 2011 [7]. We have two types of implementation of Helios - homomorphic and mixnet based. Both have their own advantages and disadvantages. In the homomorphic version the individual votes are never revealed while the mixnet version reveals the individual votes. The homomorphic version is not particularly good for elections with a large number of candidates while the mixnet version is more useful in such cases. But the implementation of the mixnet version is also more involved than the homomorphic one.

The election process in Helios is mainly divided into two main phases:

**Voting Phase:** This phase involves voters casting their votes and the authorities recording them in ballots. To vote, a voter has to encrypt his vote using the election public key and send the encrypted vote $\{v\}^r_{pk}$ (where r denotes randomness used for encrypting), together with some auxiliary data to the bulletin board through the authenticated channel.

During voting the voter also has an option to audit. If the voter chooses to audit, then the ballot and the corresponding randomness are sent to a third party which checks whether the correct choices have been encrypted or not. If the voter chooses to audit the ballot then he has to make a separate ballot to be cast [5].

There are some requirements of Zero Knowledge Proofs (zkp) in both versions. The homomorphic version requires an extra zkp to ensure

that the vote is valid. Since in homomorphic version the votes are added to obtain the final tally so if a voter sends 100 as the vote then his vote will get counted as 100 votes. Hence a ZKP is required so as to ensure that the vote has value 0 or 1. In both cases we provide ZKP that the final results corresponds to the votes cast. Mixnet and homomorphic use different conditions in this proof as the final result is presented in different formats. The Zero Knowledge Proofs are checked to ensure fairness and correctness in the election process.

**Tallying Phase:** Once the voting phase is over, the bulletin board contains a list of ballots. We distinguish the two variants for tallying purpose:

-Homomorphic tally: In the homorphic tally the authorities take the ballots from the bulletin board and combine them homomorphically. The Zero Knowledge Proof of well formedness is checked for every ballot before combining. Since $\{v\}_{pk}^{r} * \{v'\}_{pk}^{r'} = \{v + v'\}_{pk}^{r+r'}$ anyone can compute the encrypted sum of votes. Then all the authorities collaborate to decrypt this cipher-text. They also give a proof of correct decryption. [7]

-Mixnet tally: Ballots are shuffled and re-randomized and a proof of it being correct is also given. The mixing is performed by several servers to ensure secrecy of the shuffle. So if at least one of the mixnet server is honest, privacy is preserved. Then the trustees collaborate to decrypt each ciphertext and also provide a corresponding proof of correctness. The list of votes obtained is thus the final result.

## 3.2 Verifiability in Fstar

Verifiability is one of the most important properties in electronic voting. No matter which protocol we use for e-voting, we have to ensure that the result announced corresponds to the votes cast by the voters. We wish to ensure End-to-End verifiability in the election process. It was shown in [7] that according to their definitions, Individual Verifiability and Universal Verifiability entail End-to-End Verifiability provided there are no clash attacks (when two or more voters are convinced that the same ballot is their own ballot). Through Individual Verifiability we ensure that each voter is able to verify that his vote has successfully reached in ballot box. Universal Verifiability complements Individual Verifiability by providing the advantage that

anyone (including any external party) can verify that the result declared by the election authorities is according to the votes in the ballot boxes.

To verify Helios and Civitas in Fstar we use some assertions. Each of the assertion entails a specific property about voting or judging. The assertions and conditions that we have used are described below.

**VoterHappy:** We define a voter to be happy when he is ensured that the vote he sent has reached the ballot box. If the voter has followed the rules while casting his vote then he can check whether his vote is indeed in the ballot box. This way we ensure Individual Verifiability. We have an assertion VoterHappy which takes credential and vote as input and checks whether there is a ballot in the ballot box which contains that credential and vote.

```
type VoterHappy : bytes → prin → vote
   → list ballot → Type

assume VH:
   forall (by:bytes)(p:prin) (v:vote) (bb:list ballot).
     VoterHappy by p v bb
     ⟺ Votes p v
      ∧ ( exists (b:ballot). mem b bb ∧ MyBallot by p v b )
```

**GoodSanitization:** GoodSanitization takes two lists bb and vbb of ballots and checks whether vbb list contains only authentic and non-duplicate ballots from the list bb.

```
type GoodSanitization : list ballot → list ballot → Type
assume GS: forall (bb:list ballot) (vbb: list ballot).
     GoodSanitization bb vbb
     ⟺
     (forall (b:ballot).
      (mem b bb /\ fakeballot b = false
      ∧ (exists (by:bytes)(p:prin) (v:vote).
         (MyBallot by p v b)) ⟹ mem b vbb )
      ∧ mem b vbb ⟹ fakeballot b = false
      )
```

**GoodCounting:** GoodCounting ensures correct counting of votes from list of ballots. It checks that in the given result (list of votes), there is a unique vote corresponding to each and every ballot in vbb.

```
type GoodCounting : list ballot → result → Type
assume GC: forall (vbb: list ballot) (final: result).
     GoodCounting vbb final
      ⟺
      length vbb = length final
       ∧ ( exists (vbb': list ballot).
            ispermutation vbb' vbb = true
            ∧ ( forall (b: ballot)
            (i:nat{i < length vbb' ∧ i < length final}).
              b = nth vbb' i ⟹
              (exists (by:bytes). Wrap by (nth final i) b  )
            )
          )
```

**JudgeHappy:** The JudgeHappy assertion checks the initial list bb of ballots with the final result. It combines the effect of GoodSanitization and GoodCounting. It checks that the result has each and every authentic vote from initial list of ballot and contains only one vote corresponding to one credential.

```
type JudgeHappy : list ballot → result → Type
assume JH: forall (bb: list ballot) (final: result).
    JudgeHappy bb final
    ⟺
    (exists (vbb: list ballot).
     ( GoodSanitization bb vbb ∧ GoodCounting vbb final
     ))
```

All the above conditions and assertions gives us Individual Verifiability and Universal Verifiability. Both these along with no clash attack entails End-to-End Verifiability.

## 3.3 Verifying Helios in Fstar

We model the verifiability of Helios in the new version of Fstar. We model the two implementation (homomorphic and mixnet) separately. We have modeled the verifiability in the newest version available at the time writing this report.

There are two main modules in each implementation - Utils and Helios. The cryptography has also been implemented in the Helios module itself.

**Utils:** This module contains the basic function which perform operation on list. Since Fstar also has a standard list library, many of the functions are already defined in it. But we implemented some of the functions like ith, mem, ispermutation etc. mem and ith have already been defined in standard library but the implementation uses ML effect which is not compatible for calling function while verifying certain properties. We require Tot effect in those functions so we have to separately define the ith and mem function. The main functions that we define in Utils are remove_duplicates, ith, mem, sum and ispermutation. All these functions except remove_duplicates have been explained in Section 2.5 .

**remove_duplicates:** This function takes a list as input and returns a list. The list returned contains each element of the input list but it occurs only once. Moreover when an element is present more than once then it keeps the last occurrence of that element.

**Helios(Homomorphic):** This is the second module. It has the main functionality. First it defines the main data to be used. The final result is an integer, ballot is sequence of byte, vote is an integer, vvote (valid vote) is also an integer which can only take value 0 or 1 (refinement of the integer type) and voter id is string, etc. We have pkey (public key) and skey (secret key) and then an encryption function which is used to encrypt the ballot before sending it on the untrusted channel. We also assume that the encryption function we have used is injective.

```
type prin = string
type ballot = (bytes * bytes)
type vote = int
type vvote = i:int{i=0 ∨ i=1}
type result = int


type pkey 'a 'b = ('a * 'b) → bytes
type skey 'a 'b = (('a * 'b) → bytes) * (bytes → 'a)

assume val mkSkey: unit → skey 'a 'b
assume val mkPkey: skey 'a 'b → pkey 'a 'b

type Encryption:  vote -> bytes
```

```
      → pkey vote bytes → bytes → Type
```

While casting vote the voter forms a ballot containing his vote and submits it. The authorities verify that the ballot is well formed before counting the vote. So we use Zero knowledge proof of well formedness. The function check_zk_wf serves this purpose. The main use of this function is to make sure every vote which is counted has same weightage. Since we know that to calculate the final result, all the votes will be added, and votes are cast as integer 0 or 1. So a voter can involve in malpractice by sending 100 instead of 1. So we have to ensure that each voter sends either 0 or 1.

```
assume val check_zk_wf: z:bytes → c:bytes
  → b:bool{b=true ⟺ ZKValid (c,z) ∧
            (exists (v:vote) (r:bytes).
              Encryption v r pkT c ∧ (v=0 ∨ v=1) ) }
```

Next we have Zero Knowledge Proof (zkp) for proving that the result has been calculated correctly from the list of ballots. This function returns true only when there exists a list of votes which contains all the votes corresponding to the list of ballots and the sum of the list is the final result. Thus the result is indeed correct.

```
assume val check_zkp : zkp:bytes → vbb:list ballot
→ final:result → b:bool{b=true ⟹
      ( exists (res : list vote).
           length res = length vbb
           ∧ final = sum res
           ∧ ( forall (b:ballot)
           (n:nat{n < length vbb ∧ n < length res }).
             b = ith vbb n ⟹
              (exists (v:vote) (c: bytes) (z: bytes) (r:bytes).
                        b = (c,z)
                        ∧ Encryption v r pkT c
                        ∧ v = ith res n )
           )
      )}
```

We also define the attacker interface where we define all the functionality that the attacker can do. We model all the information that is available to the attacker and all the operations that the attacker can perform on it. This is modeled in the attacker interface described by the following functions. These functions show the information that is available to a well typed adversary

and the information that the adversary is capable of extracting from it.

```
val att_enc: k: bytes → v:bytes → r:bytes → bytes
let att_enc key msg nce =
    let a=unmarshallpkt key in
    let b=unmarshallv msg in
    enc a b nce

val att_dec: bytes → bytes → bytes
let att_dec key cipher =
    let a=unmarshalls key in
    let b = dec a cipher in
    marshall b

val att_pkT: bytes
let att_pkT = marshall pkT

val att_check_zkp : zkp:bytes → vbb:bytes
  → final:bytes → bytes
      let att_check_zkp zkp vbb final =
      let a=unmarshallb zkp in
      let b=unmarshalllb vbb in
      let c=unmarshallr final in
      let x = check_zkp a b c  in
        marshall x
```

After that we define the voter and the judging authority. The voter takes an id and a vvote (valid vote whose value can be 0 or 1 only) and forms his ballot. We also assert the voter happy condition which is true when we the there is a ballot box which has the vote sent by the voter. Thus voter is ensured that his vote will be counted.

```
val voter: prin → vvote → unit
let voter p v  =
    assume (Votes p v);
    let r = mkBytes() in
    let c = enc pkT v r in
    let z = zk_wf v r c in
    let b = ( (c,z) <: ballot)in
    assume (MyBallot p v b);
    assert (Wrap v b);
    assert (forall (v1:vote)(v2:vote).
       ((Wrap v1 b ∧ Wrap v2 b) ⟹ v1 = v2));
    send (marshall b);
```

```
    let bb = (unmarshall recv) in
    if mem b bb then
      assert ( VoterHappy p v bb )
      ()
    else
      ()
```

The judging function is a little more elaborate and it requires many other dependent functions. We have two assertions GoodSanitization and Good Counting which ensures proper mixnet function has been performed and from the final list the counting has been done properly. Then we have judge function which takes the input of list of ballots and sees whether the duplicate removal has been performed correctly or not. And then it checks Zero Knowledge Proof of the counting being done properly. Then it asserts the condition of the judge being happy which uses the previous assertions of GoodCounting and GoodSanitization.

```
val judge: list ballot → result → unit
let judge bb final =
    let vbb = (unmarshall recv) in
    let zkp = (unmarshall recv) in
    if (vbb = sanitize2 (remove_duplicates bb))
       && (check_zkp zkp vbb final) then
          assert (JudgeHappy vbb final)
          ()
          else ()
```

**Helios(Mixnet):** The implementation of mixnet based Helios is quite similar to homomorphic implementation. The major difference is in the ballot format. We do not need a ballot well formed proof in mixnet as voter can send any vote and it need not be 0 or 1. This changes the counting function and the type of result. Earlier we returned a single number as result while now we return an entire list of all the votes. Thus there is no quantity like vvote. The check_zkp and good counting also reflect the same. Now they just check for the existence of each vote in the final list.

```
type prin = string
type ballot = bytes
type vote
```

```
type result = list vote

assume val check_zkp : zkp:bytes → vbb:list ballot
   → final:result → b:bool{b=true ⟹
       length vbb = length final
       ∧ ( exists (vbb': list ballot).
            isPermutation vbb' vbb = true
            ∧ ( forall (b:ballot).
            mem b vbb ⟺ mem b vbb')
            ∧ ( forall (b: ballot)
            (i:nat{i < length vbb' ∧ i < length final}).
              b = ith vbb' i ⟹ (exists (v:vote) (r:bytes).
                 Encryption v r pkT b ∧ v = ith final i )
            )
       )}
```

Thus we have modeled the verifiability of Helios in Fstar. We have seen how the entire protocol works and Fstar has type checked it. Thus we are assured that Helios works correctly and there is no leak in the working of the protocol. This also assures us that the final tally is verifiable and since the judge function has verified so the final result declared is indeed in compliance with the votes and the attacker hasn't been able to break the privacy. We thus have an automated verification of Verifiability of Helios in Fstar using type-checker.

A web-based implemenetation of Helios for organizing elections is available at http://heliosvoting.org [1]

# 4 Civitas

Civitas is the first voting protocol that provides coercion resistance and end-to-end verifiability by ensuring voter and universal verifiability. It preserves voter privacy and ensures that a voter cannot prove to the adversary whom he voted for. Thus Civitas is one one of the most suitable voting system for remote voting.

Civitas is the first voting system to implement a scheme proved to satisfy coercion resistance and thus taking the secure electronic voting to reality.

## 4.1 Security Model

**Remote Voting:** It allows the voter to vote from any location thus making it very convenient for him to vote. Remote voting is thus much more general and harder than any form of supervised voting. The supervised voting can also be considered as special case of remote voting where the problem of coersion is not a major concern. [6].

We wish to have a voting system where the voter never gets to contact the authorities and just casts his vote during the voting phase but such a system is nearly impossible to achieve as the authorities first need to establish the identity of the voter. In Civitas as well, the voter is required to contact the registration tellers in order to obtain his credentials.We assume that voter is able to do so without any effect from adversary [6].

**Security Properties:** Civitas is required to satisfy verifiability should protect voter privacy and the voting system should be coercion resistance. The final tally should be verifiably correct with respect to both vote verifiability and universal verifiability.

One of the ways to fulfill the confidentiality requirement is by guaranteeing anonymity, meaning that the system server never clearly reveals how a voter voted. But this is not sufficient for a remote voting system like Civitas. Voters can gain additional information during the voting phase which could enable them to sell their votes. Such information can also be used to coerce voters and in this case the coercer could be someone present at the time of voting like family member or employer or friend.

We use the concept of non-provability to provide coercion resistance. We make sure that a voter is not able to prove to the anyone whom he voted for. Thus even if adversary tries to coerce the voter, it knows that the voter can easily fool it and vote according to his wish.

The adversary model we use can perform many types of attacks. It is able to corrupt a threshold of the election authorities, demanding their secret keys, controlling all public channels on the networks, corrupting a threshold of the voters, asking the voter to reveal his secret credential etc. Civitas is secure against all attacks from such adversaries.

## 4.2 Implementation of Civitas

**Agents:** To implement Civitas we need various types of agents. The voters are of course the very basic requirement of a voting system. The other agents required are supervisor, registrar, registration tellers and tabulation tellers [6]. The functions performed by each of them are:

- The Supervisor administers an election. This includes specifying the ballot design and the tellers and starting and stopping the election.
- The registrar decides who all eligible to vote.
- Registration tellers generate the credentials that voters use to cast their votes.
- Tabulation tellers tally votes and declares the result.

**Election Setup:** It starts with the posting of the electoral roll on an empty bulletin board by the registrar. In this way all eligible voters have been identified. After that the tabulation tellers generate public key ensuring that the decryption of the message under this key requires presence of all the tellers. Finally all the credentials are generated to authenticate the voters. The credentials are pair of public-private key values.

**Voting Phase:** This is where the real power of Civitas is clearly visible. Adversary may try as much as he wants but the voter can easily prevent from being coerced. This phase starts with voters registering and acquiring their private credentials. Each registration teller authenticates a voter using the voter's registration key and voter gets its credentials. Voter receives small part of credential from each teller and combines them to form the credential. For voting the voter has to submit a choice of the candidate and private credential in encrypted form. This submission does not require either of the voters' key. The vote is collected in all ballot boxes. This ensures that even if any ballot box gets corrupt then also the vote will get counted in final tally.

**Resisting Coercion:** The key idea that enables the Civitas to resists coercion and defeats the selling of votes is that a vote can be cast using a fake credential. So if an adversary tries to coerce a voter, the voter may do

as he wishes but by using fake credentials. The vote with fake credentials will be removed during tallying phase. Thus the voter can easily make the adversary a fool and cast his vote later. Also the voter is free to submit more than one vote in which case only the last vote will be counted.

To construct a fake credential, a voter chooses at least one registration teller and substitutes a random group element for the share that registration teller sent to the voter. The voter can construct a DVRP (Designated Verifier Re-encryption Proof is used to prove to the adversary that the fake credential is re-encryption of the public credential share) that causes fake share to appear real to the adversary.

Civitas is compatible with the use of any ballot design for which the proof of well formedness is possible.(citation)

## 4.3 TABULATION PHASE

The tabulation tellers tally the elections collectively. Firstly each teller retrieve the votes from each ballot box and also from the public bulletin board. Then they only keep the votes whose proof of well-formedness is available. In other words they discard all the ballots which are not well formed. Now since every voter has equal rights so all the duplicate ballots are removed. While removing the duplicate ballots, care is taken that only the last ballot remains and previous ones are removed. Then before removing the fake ballots they are anonymized so that it cannot be known which ones from the bulletin board were fake. The fake ballots are removed and then the result is declared along with a proof of correct counting.

Removing the duplicates and authorized votes would be easier if we had the power to decrypt but this pose a problem to coercion resistance and even anonymity. Because the voters private credentials becomes vulnerable. Instead a zero knowledge proof PET (plaintext equivalence test) is used to compare ciphertexts. This helps preserve voter privacy and thus supports coercion resistance.

The main components of the tabulation phase are the ballot boxes and the mix networks. The ballot boxes are log only services on which data can be written only once and no data can be removed. They task is to report all their content after the end of the elections. At the end of the elections, each ballot box posts a commitment to all its content on the bulletin board. The supervisor posts his signature on all commitments and thus clearly defining all the votes that will be used in tallying phase. The second

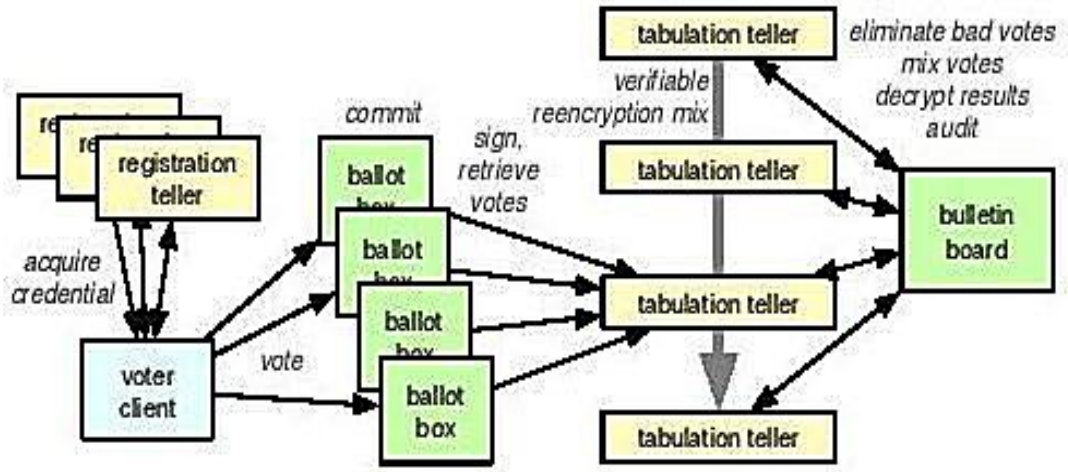


Figure 1: Civitas Architecture[1]

component is the mixnet. It plays a very important role in preserving the voter privacy. Before the fake ballots are removed, the ballots are anonymized so that adversary/voter cannot infer which ones from the bulletin board are fake. This anonymization is done by the mixnets. To reduce dependence on a single mixnet, multiple mixnets are used.

## 4.4 Verifying Elections

The tabulation tellers are required to post proof that they have done their assigned task correctly. And all tabulation tellers verify those proofs as the tabulation proceeds. Since we have already assumed that there is at least one honest teller, he refuses to continue as soon as he discovers anything wrong. These proofs are publicly verifiable and an external judge can also do the same. Thus we have ensured universal verifiability. Together with the ability of a voter being able to verify that his vote is present in the ballot box it ensures end-to-end verifiability in the absence of clash attacks.

## 4.5 Scalability

[1]This figure has been taken from "Civitas: Toward a Secure Voting System" [6]

With the current implementation of Civitas, the duplicate removal method is brute force. Same is applicable for the fake removal method. In both cases we just compare each credential with each one in the list leading to the time complexity being quadratic. This poses some problems in the usage of Civitas in national level elections, limiting its usage to just small scale election like university elections. Certain methods have been proposed in the past to improve the time complexity like grouping the voters, but they haven't been useful. A lot of work is being done in the subject and improvements are expected in near future.

## 4.6 Verifying Civitas in Fstar

To prove that Civitas is verifiably correct we model it in Fstar. To the best of our knowledge, there is no implementation of Civitas available currently in Fstar available right now. We have coded the verifiability module in Fstar and type-checked it. Some parts of Civitas implementation are quite similar to that of Mixnet version of Helios and thus have been used after minimal changes. All the functionality of Civitas has been modeled in a single module name Civitas. Going along the same lines of Helios we first implements the basic functions required on list which are not available in standard library. These include the nth function for extracting $n^{th}$ element from a list, mem function for checking an element is present in a list or not and ispermutation function for checking whether one list is a permutation of the other or not. The implementation of these functions is given in Section 2.5 .

After the implementation of these basic functions, we have the main implementation of the protocol. We first define the types to be used for main things like votes, credentials etc. It is quite similar to that of that of the mixnet version of Helios. We have ballot as bytes, votes as integer, result as a list of votes etc. We also define a function "concat" for concatenation of ciphertexts of credential and vote to form a ballot. The new thing is the credentials which have been defined. We have two constructors for the secret credential. Also we define a function faker for differentiating between actual and fake credential. This is also used in the constructors for the actual and fake credentials.

```
1 type bytes = list byte
2 type prin = bytes
3 type ballot = bytes
```

```
type vote
type result = list vote
type pkey 'a 'b = ('a * 'b) → bytes
type skey 'a 'b = (('a * 'b) → bytes) * (bytes → 'a)

assume val concat: bytes → bytes → Tot bytes


type rkey 'a 'b = ('a * 'b) → bytes
type dkey 'a 'b = (('a * 'b) → bytes) * (bytes → 'a)
type pcred 'a 'b = (('a * 'b) → bytes) * (bytes → 'a)
type scred 'a 'b = (('a * 'b) → bytes) * (bytes → 'a)

assume val mkPcred: rkey 'a 'b → dkey 'a 'b → pcred 'a 'b
assume val mkScred: rkey 'a 'b → dkey 'a 'b
    → Tot (s:scred 'a 'b{faker s = false})
assume val mkSfcred: rkey 'a 'b → dkey 'a 'b
    → Tot (s:scred 'a 'b{faker s = true})
```

Then we have the operations to be done in the encryption scheme. The encryption is quite similar to that of Helios except that now we also encrypt the credentials along with the votes. We define the encryption to be injective.

We have the same condition used in the Zero Knowledge Proof of correct formation of result from the list of ballots but the implementation also takes into account the presence of credential in the ballot. This proof ensures that correct mixing has been done and the result calculated is correct by ensuring that there is a permutation of the list of ballots such that it corresponds to list of final votes. We also have the attacker interface defining the operations that the attacker can do. These includes encryption, decryption and checking the zero knowledge proof provided the keys are available to attacker. The attacker functions are the same as that of Helios except that now we also have a vote casting function for the attacker as described later.

```
assume val check_zkp : zkp:bytes → vbb:list ballot
  → final:result → b:bool{b=true ⟹
       length vbb = length final
       ∧ ( exists (vbb': list ballot).
            ispermutation vbb' vbb = true
            ∧ ( forall (b: ballot)
              (i:nat{i < length vbb' ∧ i < length final}).
                b = nth vbb' i ⟹ (exists (by:bytes)
                  (bit1:bytes)(bit2:bytes)(v:vote) (r:bytes).
                    Encryption v r pkT bit2
```

```
                    ∧ Encryptioncr by r pbT bit1
                    ∧ BallotForm bit1 bit2 b
                    ∧ b = concat bit1 bit2
                    ∧ v = nth final i )
            )
        )}
```

After that we have the Vote, Myballot (constructor for ballot), and Voter Happy assertion. The VoterHappy assertion ensures that when the voter casts his vote then his vote in present in atleast one of the ballots. We also have function (with external implementation) for reading data from the public bulletin boards. Since we have fake votes also being cast we have two types of main voting function. The real voting function takes a string as credential (for which faker returns false) and casts a vote and asserts that the vote is present in atleast one of the ballot boxes. The fake voting function takes string for which faker is true so just casts the vote and does not asserts voter happy condition.

```
type Votes : prin → vote → Type
type MyBallot : bytes → prin → vote → ballot → Type
type VoterHappy : bytes → prin → vote → list ballot → Type

val voterreal: by:bytes{faker by = false}
      → prin → vote → unit
let voterreal by p v =
    assume (Votes p v);
    let r = mkBytes() in
    let bit2 = enc pkT v r in
    let bit1 = enccr pbT by r in
    let b = concat bit1 bit2 in
    assume (MyBallot by p v b);
    assume (BallotForm bit1 bit2 b);
    assert (Wrap by v b);
    assert (forall (by1:bytes) (by2:bytes)(v1:vote) (v2:vote).
        (Wrap by1 v1 b ∧ Wrap by2 v2 b) ⟹ v1 = v2 ∧ by1 = by2);
    send (marshall b);
    let bb = (unmarshall recv) in
    if mem (b <: ballot) bb then
      assert ( VoterHappy by p v bb )
      ()
    else
      ()

```

```
val voterfake: by:bytes{faker by = true}
    → prin → vote → unit
let voterfake by p v  =
    assume (Votes p v);
    let r = mkBytes() in
    let bit2 = enc pkT v r in
    let bit1 = enccr pbT by r in
    let b = concat bit1 bit2 in
    assume (MyBallot by p v b);
    assume (BallotForm bit1 bit2 b);
    assert (Wrap by v b);
    assert (forall (by1:bytes) (by2:bytes)(v1:vote) (v2:vote).
        (Wrap by1 v1 b ∧ Wrap by2 v2 b) ⟹ v1 = v2 ∧ by1 = by2);
    send (marshall b);
      ()
```

We have used the term fake ballot which we define as a ballot in which the credential present is fake credential or more correctly it is not a real credential. We also have GoodCounting and GoodSanitization for ensuring that the mixnet works properly and the votes are counted correctly from the list of ballots. The major difference from Helios mixnet is again the remove_fake function which removes all the fake ballots from the list of ballots.

The judge function takes data from all the bulletin boards and checks all the properties. It checks whether duplicates have been removed correctly and then there is proper permutation being performed and then each and every one of the fake ballots have been removed or not. Finally it also checks whether the result has been calculated correctly from the list of ballots or not. It also asserts JudgeHappy which ensures GoodCounting and GoodSanitization and also fake removal from the list of ballots. All this ensures proper calculation of results has been done.

```
val judge: list ballot → result → unit
let judge bb final =
    let bbdr = (unmarshall recv) in
    let bbdran  = (unmarshall recv) in
    let bbdranfk  = (unmarshall recv) in
    let zkp = (unmarshall recv) in
    if ( bbdr = remove_duplicates bb)
        && ( ispermutation bbdr bbdran = true )
            && ( bbdranfk = remove_fake bbdran)
                && (check_zkp zkp bbdranfk final) then
    assert (JudgeHappy bb final)
```

```
12      ()
13      else ()
```

We have verified Civitas in Fstar. We have modeled an adversary that is able to cast votes try to coerce the voter. Also it has all the abilities of the adversary of Helios. But the system still type checks. We have modeled the JudgeHappy condition to check that all duplicate have been removed and final ballot list contain only real credential votes. And also counting has been done properly. Since the program asserts the JudgeHappy condition thus we have an automated verification of Civitas Verifiability.

# 5 Conclusion

We have modeled two protocols - Helios and Civitas in Fstar for verifying their security properties. We have seen that Helios mainly preserves the voter privacy. It is used mainly in the university elections where coercion is not a major problem. On the other hand Civitas mainly aims for coercion resistance. It has been verified to be resistant to coercion and also preserves voter privacy. Till now the usage of both the protocols have been limited to small scale elections.

But as we know that e-voting is coming to reality, Civitas holds a great potential to the chief protocol for large scale e-voting. But it also has its flaws. The time complexity of Civitas is quadratic in number of voters which limits its usage. This can be improved in future which is will increase the usage of Civitas and thus will make the voting system more secure and increases the verifiability of the election process.

# References


[1] http://heliosvoting.org.

[2] http://rise4fun.com/fstar/tutorial/guide.

[3] https://fstar-lang.org/tutorial/.

[4] https://github.com/fstarlang/fstar/tree/master/lib.

[5] Ben Adida. Helios: Web-based open-audit voting. *17th USENIX Security Symposium*, 2008.

[6] Andrew C. Myers Michael R. Clarkson, Stephen Chong. Civitas: Toward a secure voting system. *IEEE Symposium on Security and Privacy*, 2008.

[7] Steve Kremer Matteo Maffei Cyrille Wiedling Véronique Cortier, Fabienne Eigner. *Principles of Security and Trust*, chapter Type-Based Verification of Electronic Voting Protocols. Springer Berlin Heidelberg, 2009.