5-2017

# Formal Verification of Receipt Validation in Chaum's Scheme

Kyle Savarese
kms7341@rit.edu

# Formal Verification of Receipt Validation in Chaum's Scheme

by

Kyle Savarese

THESIS

Presented to the Faculty of

the Golisano College of Computer and Information Sciences

Computer Science Department

Rochester Institute of Technology

in Partial Fulfillment of the Requirements for the Degree of

Master's of Science in Computer Science

Rochester Institute of Technology

May 2017

# Formal Verification of Receipt Validation in Chaum's Scheme

APPROVED BY
SUPERVISING COMMITTEE:

_____

Dr. Matthew Fluet, Chair

_____

Dr. Edith Hemaspaandra, Reader

_____

Dr. Stanislaw Radziszowski, Observer

# Abstract

In the aftermath of the United States Presidential election, more and more frequently there are calls for voters to be able to place their votes from the comfort of their own home. However, many studies have found prototype systems to be either insecure or insufficiently defined for the purposes of an election on a national scale.

In this paper I will examine the security of voting applications from a different angle: the validation and verification of compiled code. There are the obvious concerns about unverified code, that we have no guarantee the protocol described by the voting procedure is the one being executed. Using work by Appel [3] as a model, it can be seen that even advanced cryptographic algorithms can be verified. Using Chaum's scheme, a visual cryptography system intensely examined in Staub's work [1], and originally described in Chaum's paper [5], as our target enables us to have a secure algorithm that we can properly verify. Our goal will be to establish a verified code implementation for Chaum's scheme that could be deployed to voters to confirm their votes.

# Table of Contents

# Chapter 1: Introduction

There has been a great deal of discussion in recent years about the security of current voting procedures and how we can both increase security and public confidence in our election processes. These concerns have led to a number of potential voting systems to be proposed that would cover those concerns, normally involving a series of cryptographic procedures that would provide the qualities needed for votes to be processed securely.

A voter needs to be assured that those in charge of processing the election tally cannot determine who they voted for, while also being able to know that their vote will be processed correctly. The scheme we've chosen to study is Chaum's scheme, which utilizes visual cryptography to provide these guarantees to voters. However, the changes that switching to Chaum's scheme would create in the voting process would be likely to generate distrust and concern among the general public. With public confidence and trust in the process being important factors in the successful completion of an election process, providing additional assurances to voters that their votes are acting just as they had before.

With this in mind we can look towards formal verification methods to produce programs that have a guarantee of their execution behavior provided with them. One of these in particular is the Verified Software Toolchain, a connection of a certified C compiler with a proof script written in the Coq proof language. This takes a C program and a series of written proofs regarding the

behavior of the functions and commands within the program to as close to a guarantee of execution behavior as we can get for now.

With this in mind, we can apply the processes described in the Verified Software Toolchain to portions of Chaum's scheme to provide voters with a reassurance that despite the changes their votes are processed just the same as before. This is the thrust of this paper, to provide a formal verification of the vote validation process within Chaum's Scheme. Chapter two will provide a detailed background on both Chaum's Scheme and the Verified Software Toolchain, including a demonstrative proof of a simpler program, with chapter three providing the formally stated thesis description along with a description of the C language source code of our program. Chapter four will provide a description of the difficulties encountered during the process of proving our goal, and then chapter five explains some design decisions along with a group of future work that would help further the value of the proofs within this thesis.

# Chapter 2: Background

## 2.1 Chaum's Scheme

First we should establish some terminology regarding modern voting technologies: Internet voting is the use of an online portal or forms to cast a vote in an election, whereas e-voting or electronic voting is the use of any electronic process, be it an online technology or the booths used at many polling stations today.

As democracy expands and voters' lives become progressively more busy, countries are looking for methods to make it easier than ever to vote in their important elections. Many have called upon governments to build a method to vote via the Internet [12], or expand current e-voting procedures. Meanwhile security experts continue to cite a lack of transparency in e-voting machines' code as a continuing and growing security risk [8,9], and many have warned heavily against the idea of online voting, saying it is too insecure and the stakes are too high for failure [10].

Source code of electronic voting machines is considered proprietary by many companies, making it impossible for individuals and experts to properly certify the back-end code. This is inherently dangerous in national voting schemes, as there is no way for outside third parties to be certain that no changes or modifications to votes are taking place, placing the credibility of such elections in doubt [9]. While there has yet to be a proven case of fraud using e-voting machines (also known as Direct Record Electronic machines, or DRE machines), the potential exists in systems without proven cryptographic security [11].

Proposed primarily for e-voting procedures, Chaum's scheme as specified in his original paper [5] and described in a detailed analysis of it in Staub's work [1] is a potential way of putting to rest some of the security concerns with e-voting machines. First and foremost, the algorithm is public, enabling security and cryptography experts to analyze the process and confirm that it is in fact secure. Additionally, it has an easily implemented verification section that allows voters to quickly check that their vote has been properly cast and not modified or corrupted. While each

individual voter only has a fifty percent chance of detecting fraud, they can be nearly certain on a national scale that the votes are being cast correctly due to sheer mathematical probability.

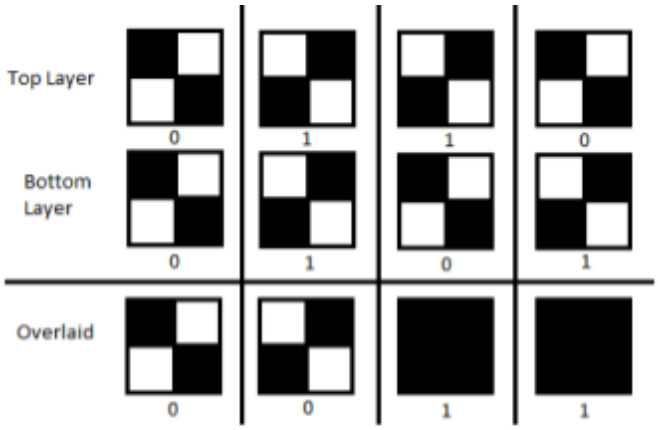
Chaum's scheme is an electronic voting procedure based on visual cryptography [13, 15]. Visual cryptography is formed as follows: two different bit images for 0 and 1 are formed (see

0 1

below) [13, 5]. These bit images are combined together to form a layer, which when combined with another layer can form an actual image, see generated example below (code from Stajano [14]).


Each two by two bit image is represented in the final image, with an all black square being the overlay of two different bit images, and a square with white on it is an overlay of two of the same bit image. Each layer of bit images is random noise to anyone without the other layer, only once combined do the layers hold any value, with this noise being generated in a process described in 2.1.1.

On a high level, Chaum's scheme allows a user to vote using this process. A user selects a desired series of votes, which are then turned into layers of bit images that combined form an image holding those votes. Each layer is then encrypted and, combined with other information, placed into a *receipt* which holds that information that is then put online to be tallied as well as printed for a user to take home. A central committee then decrypts the layers and uses them to produce the final ballots, and with those, the total vote count.




*Columns represent xor operation*
*Rebuilt from information in Staub [1]*

Let us dig deeper into that high level description of what makes Chaum's scheme an effective way to cast ballots in an election. All election schemes require three important properties, anonymity, integrity, and coercion immunity [16, 9]. Anonymity specifies that it is impossible for anyone to determine how another person voted, integrity specifies that a person's vote must be counted the way it is cast, and lastly coercion immunity specifies that it is impossible for a voter to prove who they voted for to anyone else.

Chaum's scheme provides this with a receipt system that holds information which when properly manipulated provides coercion immunity and (statistical) integrity, along with a distributed tally process that protects anonymity. Through manipulation of information provided on the receipt as

described below, along with the manner in which *trustees* (individuals entrusted with parts of the decryption process) do their work, all three conditions are met.

With that in mind, let's now go through a description of the scheme step by step, ignoring some specifics that we will comment on afterwards. The first step is a voter selecting their list of vote choices on a ballot, including write-ins or other election-specific options, just as they might on a current machine. From there, the machine prints out the beginning of a receipt, where the receipt is a set of aligned graphics on the top and bottom of a separable page. An example of this sort of split is shown above, with the letter "e" being split into two layers that look simply like random noise. All letters would be similarly constructed for the purposes of the receipt, forming a total image.

After this initial portion is printed, the voter selects which part of the receipt they want, and which part of the receipt is to be eliminated. Once the layer has been chosen, the printer prints the remaining portions of the receipt (which describes instructions of what to do with each layer of the receipt). The ordering here is crucial for the integrity of the algorithm: if a voter were to select the chosen layer prior to printing, the machine could "cheat" here, knowing that the voter would have no way to verify the layer they did not choose. Therefore it could freely manipulate the layer, changing votes at will.

A voter can now leave the polling station with their chosen receipt, as it is a grid of the bit symbols shown above, and is in effect a one-time pad acting as the key for the discarded layer which now only exists as an encrypted electronic version to be sent to the official counting location. This receipt can be shown to anyone, in any manner, and all that a viewer would be

able to determine is whether the ballot is authentic and legitimate, and not who the vote was for, thus providing coercion immunity. The process for verifying a receipt is described, coded, and the code formally verified throughout this thesis. Additionally, this receipt can be checked against an official website via serial number and checked for accuracy. Further, the website will post a batch of all receipts to be processed, along with the the output of a cryptographic signature taken on the receipts. This can be compared with the results to ensure the same number of items is in each of the posted batch and the result set. Through open-source programs and releases of intermediate batches, any person or group can verify the receipts posted as the input receipt batch do correspond to an (unknown) receipt in the tally batch.

An important question here is how these receipts work, and what properties they give each vote they contain. Any receipt posted online will be included in the tally and cannot be decoded without the keys of the distributed trustees. More importantly, even an attacker with infinite computing power and access to every election computer in an election could only modify each vote in three manners: printing an incorrect layer, repeating serial numbers, or cheating in the tally process. Due to the manner the election is counted, each of these attacks has only a fifty percent chance of succeeding per modified ballot. Changing a mere twenty ballots has a smaller chance of evading detection than one in one million, with the odds scaling exponentially worse and worse for the attackers. This provides integrity through sheer volume - the odds of an attacker being able to flip enough votes to sway an election are incredibly, nearly impossibly low, making the process secure.

The details of constructing the layers will be described below, but let it suffice for now that we build them such that when overlaid they produce the desired ballot image, and both appear to

be (and are)completely random. We then have two layers that are noise, with each acting as a one-time pad of sorts for the other, with only the election trustees having a way to recreate the ballot image from a single receipt layer. A more formal description of this is available in both Staub's [1] and Chaum's [5] work. Is it possible for the receipt layer to encode different choices than the voter sees on their printout? Only if a single layer has been falsified - if both were changed, then the voter would be able to see the fraud immediately. If just one layer was changed, then each instance of a changed ballot would have, as mentioned before, a fifty percent chance of being caught and shown.

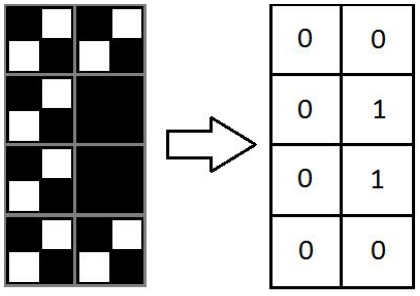
Next we must determine how to get from this set of receipts to a set of final images. The election has a number of trustees, ideally who have either no stake in the election or opposing stakes, and so are highly incentivized to keep the others honest in their roles. On the receipt there is a set of information described below that holds the information on how the votes are to be recovered: a set of two data structures (which can be thought of simply as int arrays), termed *dolls*, that are in fact just encrypted versions of each of the layers. The layers are passed through repeated rounds of encryption, one for each trustee, where a public key corresponding to that trustee is used to encrypt the layer. Once the encryption process is complete, the dolls on our receipt are created. To tally the final vote, trustees apply their private decryption keys in sequence to recover both layers and with the layers the ballot image corresponding to each vote. There are once again details that help ensure the integrity and anonymity of each individual vote within the precise scheme, but for our purposes this description will do.

### 2.1.1 Receipt Generation

There are still some important details for Chaum's scheme to cover for a full understanding of how it all comes together, so we will cover that now. We need to look in depth at the mathematics and process of how layers are constructed, how receipts are constructed, and lastly how receipts are verified, which will be a bit further below in its own section.

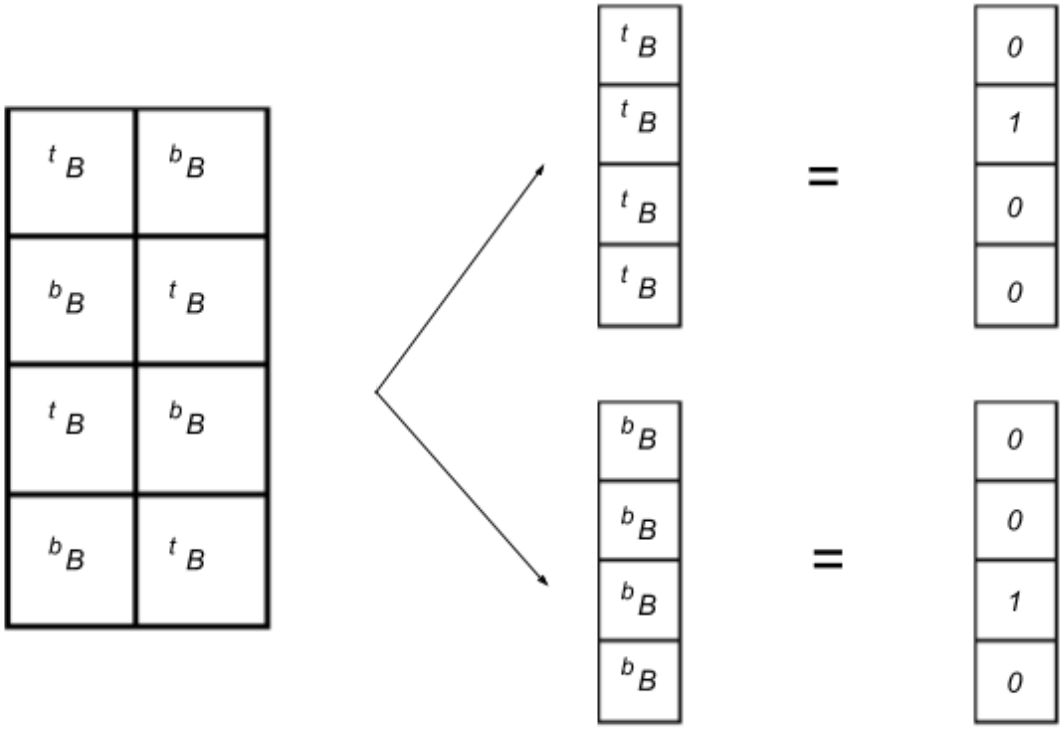
We can begin with how layers are constructed. We already know that the layers, when overlapped and aligned properly, should produce the desired ballot image selected by the voter. Additionally, we know that each of the two layers should appear to be random bits to any user without the other corresponding layer. At the same time, we do need a somewhat deterministic method to generate these layers so that our deterministic machines are capable of making the layers quickly without having to sample atmospheric noise or something similar.

So how do we generate our pseudorandom bits? We will use a small 4x2 ballot section to illustrate the process, shown here, with the image converted to a ballot matrix. From there we split the Ballot into two matrices ${}^{t}B$ and ${}^{b}B$, by *checkerboarding* the ballot as seen on a game board (presented below). This generates



| | |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 0 | 1 |
| 0 | 0 |

${}^{t}B$ = (0100) and ${}^{b}B$ = (0010) for our example.



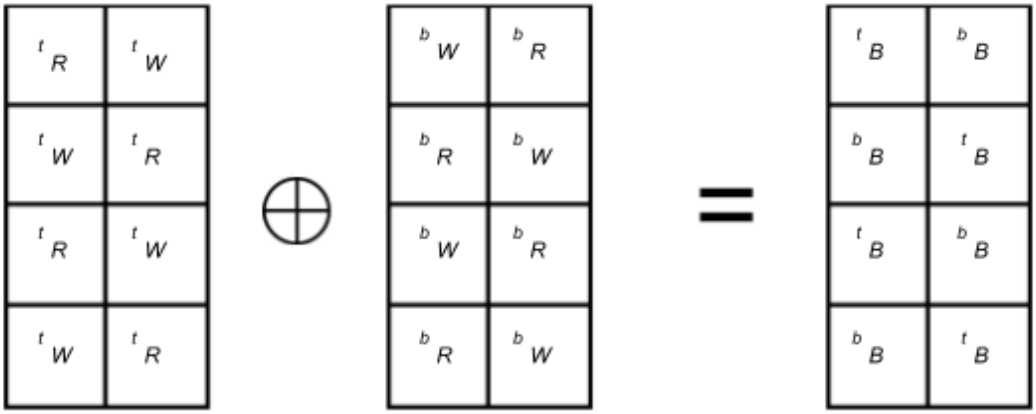

*Created from information in Staub [1]*

The next step is important for all parts of the receipt - layers, dolls, and verification, and involves some additional requirements - a known number of trustees that corresponds to the number of rounds of encryptions/decryptions, along with a pair of hash functions that output a value of the same length as half the number of values in the ballot matrices and a signature function with known public keys for both the top and bottom layers. For our example, we will use four rounds of trustees for both creation and verification. The next step is then as follows - generate four variables for both the top and bottom layers, where a first hash function $h_1$ takes the signed serial number of the receipt using the respective layer along with the number of the current trustee (for us, a value between one and four inclusive). Formally, for each $l$ in $1 \leq l \leq 4$ and for $x = t \text{ or } b$, ${}^{x}d'_{l} = h_1(s_x(q),\ l)$, with $s_x$ representing signing the serial number $q$ with the public key

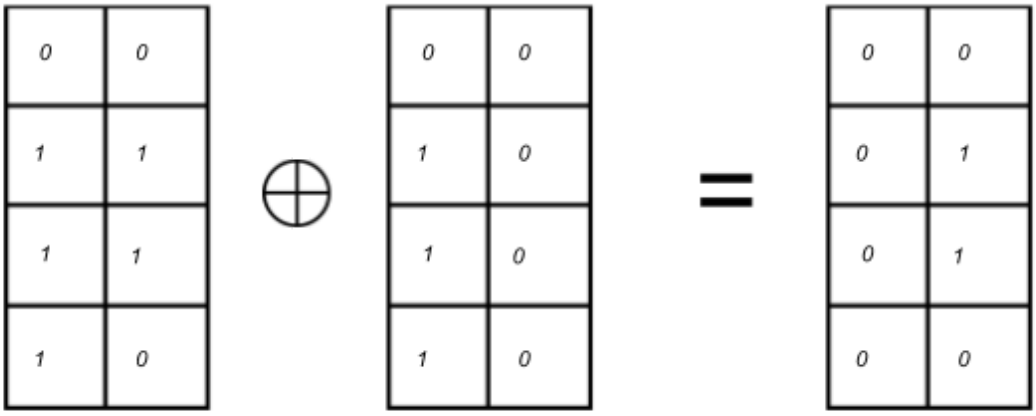
*x* (which is a distinct integer array for both top and bottom). These *d'* variables will be used later to construct the dolls, used in recreating the unknown layer, as well as in confirming the dolls were formed properly while verifying the receipt, see below.

We then take each of these *d'* variables and input them into the second hash function $h_2$. This gives an equal sized set of variables *d* defined as follows: for each $l$ in $1 \leq l \leq 4$ and for $x = t \text{ or } b$ ${}^{x}d_l = h_2({}^{x}d'_l)$. We will form the ballot image using two sets of paired binary strings, ${}^{t}W$, ${}^{b}W$, ${}^{t}R$, and ${}^{b}R$. These strings will be checkerboarded as seen below into each ballot layer. From here we form our first layer as follows: ${}^{t}W = \oplus {}^{t}d_l$ and ${}^{b}W = \oplus {}^{b}d_l$, forming the top and bottom layers by *xor*'ing our generated variables together into a single bit string. We define this output as follows for our example, since we have no particular hash or signature function that we are particularly specified on: ${}^{t}W = (0111)$ and ${}^{b}W = (0010)$. This only provides half the picture, and we need to chose another set of values such that when overlaid with our generated set of values will produce the desired ballot image. Since the overlay acts as an *xor* function, we form the other set of binary strings ${}^{b}R$ and ${}^{t}R$ as follows: ${}^{t}R = {}^{b}W \oplus {}^{t}B$ and ${}^{b}R = {}^{t}W \oplus {}^{b}B$. For this example, this gives us ${}^{t}R = (0010) \oplus (0100) = (0110)$ and ${}^{b}R = (0111) \oplus (0010) = (0101)$.

There is one last step to take to turn our bit strings into layers, which is to build each final layer partially from the generated bits and partially from the required bits. This is done, as described earlier, by *checkerboarding* the two layers, in a pattern seen below. With the checkerboarding complete, we have our two layers determined and ready to be passed onto the next stage of receipt creation.




Top Layer Bottom Layer Ballot




Next we should describe how the dolls (encrypted layers, so called because each layer of a doll reveals another level of encryption within) are formed. We begin with our $d'$ variables from the layer construction, and an encryption function with public keys for each trustee in the process. We then form the final doll as follows: encrypt each successive $d'$ variable using a trustee's key along with the results of each prior encryption, which formally looks like this for both the top and bottom layers:

$Doll_1 = e_1(d'_1);\ Doll_2 = e_2(d'_2,\ Doll_1);\ Doll_3 = e_3(d'_3,\ Doll_2);$ and $Doll_4 = e_4(d'_4,\ Doll_3) = Doll$.

Put together, we have

$$^{t}Doll = e_4(^{t}d'_4,\ e_3(^{t}d'_3, e_2(^{t}d'_2,\ e_1(^{t}d'_1)))) \text{ and}$$

$$^{b}Doll = e_4(^{b}d'_4,\ e_3(^{b}d'_3, e_2(^{b}d'_2,\ e_1(^{b}d'_1)))).$$

Later, during the decryption and tally process, the layers of encryption will be stripped off one at a time, revealing each of these *d'* variables. Those will be used to recreate the missing *W* layer, and with that in hand half of the bits used in the final image. While this may seem insufficient for forming the final image, we do know approximately what the final images may look like on large images with a defined alphabet of characters. Therefore it is possible to take a very accurate guess as to what the final ballot image is supposed to be.

### 2.1.2 Receipt Verification

Let us begin with a table of notation, in order to help provide context to the precise process of verification. For those unfamiliar with it, Coq is a formal proof management system used to complete our full proof and will be further described later on.

| Variable | C-Type | Coq Type | Description | Accessibility |
|---|---|---|---|---|
| *m* | int | Int | Rows of bit images in image | Public |
| *n* | int | Int | Columns of bit images in image | Public |
| *x* | (none) | (none) | Denotes top or bottom | Public |
| $D^x$ | Int array (*m by n)* | List Z | Doll (encrypted seeds) | Both top and bottom Public |
| *I, j* | Int array (*m by n)* | List Z | Member of singed tuple | Public |
| *q* | Int | Int | Serial Number | Public |
| *r* | Int | Int | Member of signed tuple | Public |

| | | | | |
|---|---|---|---|---|
| *v* | Int | Int | Potentially signed serial number | Public |
| *w* | Int | Int | Member of signed tuple | Public |
| $L^x$ | Int array (*m by n)* | List Z | Layer of image | One public, one destroyed |
| *k* | Int array (*m by n)* | List Z | Member of signed tuple | Public |
| $t_x()$; *x* is layer key | Int -> int -> int | Int -> Int -> Int | Signature function | Private |
| $s^{-1}()$ | Int -> int -> int | Int -> Int -> Int | Inverse signature function | Public |
| $p_x()$; *x* is layer key | Int array -> int -> int array | List Z -> int -> List Z | Signature function | Private |
| $o^{-1}()$ | Int array -> int -> int array | List Z -> int -> List Z | Inverse signature function | Public |
| *I* | int | Int | Trustee identifier, encryption key | Public |
| $e_I()$; *I* is trustee key | Int array -> int -> int array | List Z -> int -> List Z | Encryption function | Public |
| *h()* | Int -> int -> int | Int -> Int -> Int | Hash function | Public |
| $^xd'$ | Bit array (*length* $m * n/2$*)* | List Z | Layer components | Private |
| $^xW$ | Bit array (*length* $m * n/2$*)* | List Z | Layer bits | Private |
| $^xR$ | Bit array (*length* $m * n/2$*)* | List Z | Layer bits | Private |

The final receipt that an individual will have to verify will be an int array with the following information in it (with *x* being either the top or bottom layer, and the corresponding key when seen next to a function):

$L^x$, $q$, $D^t$, $D^b$, $v$, and a tuple of *(k, r, i, j, w)* with $v$ being the output of $t_x(q)$ and the tuple being $k = p_x(L^x); r = p_x(q); i = p_x(D^t); j = p_x(D^b); w = p_x(v)$, with each value being as described in the table above.

To summarize, we have the encrypted versions of both layers, the decrypted version of one, a serial number, and a set of six values (one by itself and the other five grouped) that have had some operation done to them by the machine, which we hope to be the signature function using the correct signature keys for their layer. Other known information is the encryption function each trustee uses, and the public key used with that encryption function. For simplicity, let us assume that there are four trustees, and each of them uses the same encryption function, and the subscript on the function will simply represent the encryption key. Additionally, the inverse signature functions are both available, along with their inverse signature keys for both layers.

Knowing all of this, a voter can then verify their receipt by checking three facts:

1. Take the inverse signature of the signed serial number and confirm that it matches - that is, $q = s_x^{-1}(v)$. The function $s^{-1}$ is a public function (with a public inverse signature key for each layer) that has been stated by those running the election to properly de-sign serial numbers. The value $v$ is the output of the signature function (hopefully for which the published $s^{-1}$ is in fact an inverse), and which we hope is equivalent to $s_x(q)$, which would make our stated goal a tautology.
2. Take the inverse signature of the tuple *(k, r, i, j, w)* with the second public designing function $o^{-1}$ and confirm the values match the corresponding unsigned values in the tuple. Using the previous notation,

$L^x = o^{-1}{}_x(k);\ q = o^{-1}{}_x(r);\ D^t = o^{-1}{}_x(i);\ D^b = o^{-1}{}_x(j);\ and\ \ v = o^{-1}{}_x(w)$. Once again, if $p_x = o_x$ for which $o^{-1}{}_x$ is an inverse then all of these equations reduce to tautologies and we have proven our voting machine to be using the correct signing procedures (and as such the machine is genuine).

3. Compute the hash values of the signed serial number and index for each key that exists. Then, encrypt the values using the public encryption keys in the same order the indices were applied (key one on hash one, key two on hash two and the first encrypted value, key three on hash three and the second encrypted value and so on), and compare the final value to the value of the doll listed on the receipt and confirm that they match. These hash values are the same as those computed in the first step of generating the layers. In notation assuming four trustees as above -
    a. ${}^x d'_l = h_1(v, l)$ for each trustee *l*. Each of these variables is a bit string which has length equal to the number of pixels in the image (length times width) divided by two, which may exceed the size of a standard int, in which case they become an integer array.
    b. $D^x = e_4({}^X d'_4 , e_3({}^X d'_3 , e_2({}^X d'_2 , e_1({}^X d'_1))))$. Each encryption function outputs an integer array to pass to the next function along with each of the trustees variable integer arrays, and the final one should match the doll of the layer we have kept.

The last check confirms that the doll (and thus the vote) was formed correctly. This verification process is what will be written and formally verified in this thesis, using the Verified Software Toolchain, described below. This gives the general flow of Chaum's scheme for voter-verifiable elections. For more details see either Chaum's original paper [5] or a thorough examination of it by Staub[1].

## 2.2 Verified Software Toolchain (VST)

As much as high level programming has abstracted away many of the unspecified behaviors and difficulties of the past, there are times when either the raw performance or the ability to write precise code dictates the way we program. And even in high-level languages, compiler bugs and imperfect implementations can create strange edge cases that may never fully be squashed out. As a result, we need a method to guarantee the behavior we intended is the behavior that happens. The Verified Software Toolchain is one option for it [2], proof-carrying code is another [17].

The solution to this for high priority systems is a process of formal verification, where users of a program can be absolutely certain via formalized logic and proven compilers that the code they wrote does precisely what they anticipate. In his original paper in 2011 [2], Appel describes precisely just a process, the Verified Software Toolchain (VST), whereby through a series of proofs, a programmer or team of programmers can verify their program.

Let us quickly cover some basics needed for the VST to work. Coq, as mentioned before, is a formal proof assistant where users specify *.v* files that hold proof instructions called *tactics* that are sequentially executed to prove theorems and lemmas in a constructive logic environment. For those unfamiliar, the online guide Software Foundations by Pierce, et. al. [7] is an excellent start up guide on the basic workings of Coq. CompCert is a project to make a verified C compiler, that guarantees the same behavior between the C code and their compiled versions.

With that in mind, what is the Verified Software Toolchain? It is a series of formally verified transformations from high level language down to executable machine code. We start by simply writing a C program as for any other coding project. Then we convert the c file into a Coq .v file, which we then write a verification of in Coq. The verification in Coq of the program acts as our formal proof that it functions as we expect, and the CompCert compiler semantically preserves the meaning of the programs.

The first step to the VST proof is formally defining a functional specification of the chosen task within Coq, since without that the program cannot have a formal behavior. Once established, we can use Coq libraries provided with VST along with standard Coq tactics to prove that the converted C code has the same behavior as the defined specifications. By defining pre- and post-conditions along with function definitions, we can also easily split the proof into specific concerns rather than a single large proof.

An important part of the VST is the CompCert compiler by Leroy, which enables the step between high-level code and machine code to be certified. The CompCert compiler for CLight provides a formally proven compiler where the input program is guaranteed to have the same behavior as the machine code output, verified using the Coq proof assistant again. For VST, what this means is that if we can prove that the code has a specific behavior (as we prove in our own Coq .v files), then the compiled code from the CompCert compiler will also have that behavior. This behavior includes type checking, memory assignment, and bounds checking.

### 2.2.1 Hoare Logic

The Coq proofs rely on a particular feature of the chosen program logic and target language within CompCert: Hoare Logic, and specifically Hoare triples [6]. Hoare triples are a composition of a statement *c* and two predicates *P* and *Q* on states: {*P*} *c* {*Q*}, where if the state before *c* executes satisfies *P*, then after *c* executes the state satisfies *Q*. This is the root of the functional specifications seen later, along with the basis for how we can do symbolic execution on each C statement to step through the proof.

For those unfamiliar, here is an example of how the sequential composition rule of Hoare logic works. If we start with the predicate *P = { x => 3 }* and the commands *y = x - 2; z = y + 1;* and *a = x*, then the set of predicates *Q = { y => 1; z => 2 }* would hold on the new state after the commands executed, as would any set of predicates with the variables set to their proper values.

### 2.2.2 Tutorial

To assist in understanding the basics of the VST, we have prepared a tutorial that can be used alongside the one described on the VST home page [6]. We will focus here on how to use the Coq tactics introduced by VST to progress through the necessary proofs, broken into a few large categories: Hoare logic commands, arithmetic commands, memory management, and entailments.

First let us show the program that we are working on proving to provide context to our example proof.

```
void addVal(int bits[], int n, int size) {
```

```
  int i, x;
  i = 0;
  x = 0;
  while (i < size) {
    x = bits[i];
    x = x + n;
    bits[i] = x;
    i++;
  }
  return;
}

int four[4] = {1,2,3,4};

int main(void) {
  addVal(four, 4, 4);
  return 0;
}
```

As we can see it is a fairly simple program, which does array reads and writes along with function calls and a loop. While simple it ultimately does cover the basics of all programs. Before we get into the tactics there is some background Coq work that we should examine. First and foremost we need to examine functional specifications. Functional specifications are the way we tell Coq what functions are expected to do. This entails preconditions, postconditions, changes to data passed in, local variables used, and the returned values. These are then proven by the VST to be the basis of our formal proofs. An example of these appear below:

```
Definition addVal_spec :=
  DECLARE _addVal
    WITH bits: val, sh: share, contents : list Z, n : Z, size: Z
    PRE [ _bits OF (tptr tint), _n OF tint, _size OF tint ]
          PROP  (readable_share sh; writable_share sh; 0 <= size < Int.max_signed;
                 size =  Zlength contents;
                 Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) contents)
          LOCAL (temp _bits bits; temp _n (Vint (Int.repr n));
                 temp _size (Vint (Int.repr size)))
          SEP   (data_at sh (tarray tint size) (map Vint (map Int.repr contents)) bits)
    POST [tvoid]
```

```
PROP () LOCAL ()
SEP (data_at sh (tarray tint size)
  (map Vint (map (fun i => Int.add i (Int.repr n)) (map Int.repr contents))) bits).
```

The *WITH* clause describes the Coq inputs to the function. The arrays following the *PRE* and *POST*-conditions declarations describe the C parameters and return value respectively, with the *PROP, LOCAL,* and *SEP* clauses describe requirements of the inputs and the states at the beginning and end of the function. The *PROP* clause is a set of formal Coq propositions that must be proven upon calls to the function, the *LOCAL* clause holding any variables that would be necessary, and the *SEP* clause holding any information regarding memory management. When the proof of this function begins, we will have the hypotheses in the *PROP* clause as assumptions in our proof state, with the *SEP* clause commonly holding *data_at* clauses that state the values held in memory.

What does our specific precondition mean though? The first two statements in the *PROP* clause describe that the *share* which describes how data in arrays can be accessed, and in particular that the function has both read and write access. The third and fourth statements tell us that the *size* parameter is a nonnegative 32-bit signed integer which has the same value as the length of the array that has been passed in. Lastly, each of the values in the array can be represented by a 32-bit signed integer. The *LOCAL* clause describes how the function parameters match up to the Coq values in the *WITH* clause, and in our case that the function parameter *_bits* holds the value of *bits*, and that *_n* and *_size* hold the integer representations of the parameters *n* and *size* respectively. Lastly, our *SEP* clause tells us that there is an array at address *bits*, that can be accessed with share *sh* and containing *contents*. Our postcondition is far simpler, the

function returns no value and the array now holds the value of *Int.add (Int.repr n)* applied to each value in the original array.

With that in hand, we can begin with the set of commands related to Hoare logic - those focused around processing a C command (normally a line of code) and advancing the program state based on that command. This is a small group of tactics, but often involves other portions of the proof as their parameters. The tactics include *forward, forward_if, forward_call,* and *forward_while*. The *forward* tactic handles simple things like variable assignment, return statements, memory loads and assignments, continue statements, and break statements. Of those, anything interacting with memory will require bounds checking subgoals, which we will look at later. The remaining tactics require inputs with their use. The *forward_if* tactic requires a postcondition that describes the final state after either branch executes. The *forward_call* tactic requires the functional specification for the function being called. Lastly the *forward_while* tactic requires a loop invariant, which describes what each step of the loop should do. That will be described in detail on its first usage.

Our proof centers around proving each function in our program to be formally correct. This is done by first providing functional specifications, as described above, followed by proving that each step in the program is correct via the Hoare logic tactics. The proof just showing the Hoare logic tactics follows, with expanded commentary afterwards.

```
Lemma body_addVal:  semax_body Vprog Gprog f_addVal addVal_spec.
Proof.
start_function.
forward. (* i = 0 *)
forward. (* x = 0 *)
```

```
forward_while (addVal_Inv bits sh contents n size).
* (* 1. Prove that current precondition implies loop invariant *)
Omitted tactics
* (* 2. Prove that loop invariant implies loop ending condition will occur *)
Omitted tactics
* (* 3. Prove postcondition of loop body implies loop invariant *)
(* start with x = bits[i] *)
Forward.
  (* bounds check requirement, index manipulation work *)
(* x = x + n *)
forward.
(* bits[i] = x *)
forward.
(* i++; *)
forward.
(* end of for loop *)
(* now we have to prove that we have created the next instance of the loop invariant *)
  (* array manipulation work, see below *)
* (* 4. prove the code that is after the loop *)
forward.(* return; *)
(* post_condition satisfied check *)
Omitted tactics
Qed.
```

The opening lines up to *start_function* are boilerplate code for all functions and they generate the environment for the Hoare logic tactics to work in, and provides it with known hypotheses (pulled from the preconditions) along with the C command and local variables that we are currently working with. Lastly, it creates a series of commands within *Ssequence* clauses to be individually handled and processed. From here we can follow through our C code verification, one step at a time.

Our program begins with the following set of lines: `int i, x; i = 0; x = 0;`. As mentioned earlier, simple assignment statements are handled with the *forward* tactic, but what does it do in

this context? Each one peels off an *Ssequence* clause to see the command within, and if the tactic matches the command, the tactic successfully executes. In this case, the *Sset* statement adds the internal variable to the *LOCAL* clause and then sets the next state up. After those lines have been executed, we now arrive at our *while* loop, which requires a different set of information.

As stated before, the *forward_while* tactic requires a loop invariant to process. For our *addVal* function, the loop within has an invariant as shown below:

```
Definition addVal_Inv bits sh original_contents n size :=
  EX i: Z,
    PROP (0 <= i <= size)
    LOCAL (temp _bits bits;
           temp _i (Vint (Int.repr i));
           temp _n (Vint (Int.repr n));
           temp _size (Vint (Int.repr size)))
    SEP (data_at sh (tarray tint size)
           ((map Vint (map (fun i => Int.add i (Int.repr n))
                           (map Int.repr (sublist 0 i original_contents)))) ++
           (map Vint (map Int.repr (sublist i size original_contents)))) bits).
```

This describes what we expect our program state to be in when we begin processing a loop. We have a precondition that describes the bounds on the loop, a set of local variables we expect to manipulate or use during the loop, and a description of any memory we use during the loop, which can describe changes made during the loop. This invariant is passed to the *forward_while* loop, along with the parameters needed by the invariant, in our case the *bits, sh, contents, n,* and, *size* fields. Thus our final command ends up being

```
forward_while (addVal_Inv bits sh contents n size).
```

We can pause for a moment to look at the *SEP* clause and examine what it is saying about this loop. This *data_at* clause holds that everything below the index we are currently working on (*i)* has had its value modified by the addition function, with everything above it still holding its original value. With this we describe how we modify values step by step.

The *forward_while* tactic does not simply generate the next step in the proof though, it generates four distinct subgoals to handle. First the current program state needs to satisfy the loop invariant, which will be briefly touched on later, along with the second proof goal, which claims that (in this case) the loop condition is guaranteed to successfully execute. The third goal is of interest currently though, it describes stepping through the body of the while loop and proving that at the conclusion of the loop the program state satisfies the loop invariant again. The final subgoal will be to prove the code after the loop with the final instantiation of the invariant as the precondition.

From here we can focus on the loop body section. Our first command is `x = bits[i];` which is an assignment so we can process it simply using *forward*. However, one might note that an access of a C array can have undefined behavior if the index of the array is out of bounds. For that reason, when we attempt this goal we get a subgoal to prove that the index is in bounds. This specific subgoal proof will be briefly covered when we discuss memory management. Discharging this responsibility for now, we can continue proving the loop body.

This leaves us with three remaining statements: $x = x + n;\ bits[i] = x;\ i++;$ which each simply process using *forward*. With those statements concluded, we are left with a goal to prove that the current program state is once again an instantiation of the loop invariant. Parts of this will be covered later within the memory management and entailment sections, but we can again move on for now, leaving us at the program state after the *while* loop. Lastly, we clear our a final *return* via *forward*, leaving us a separation logic proof that the postcondition of the function has been fulfilled, concluding the proof of the *addVal* function body.

The Hoare logic proof is not finished yet: We also need to prove the body of the *main* function in order to have proven the total behavior of our program. The *main* function utilizes an external array definition, so we define it in our proof script as follows:

```
Definition four_contents := [1; 2; 3; 4].
```

From there we begin our proof the same as for the *addVal* function, with a statement added to map the definition we just added:

```
Lemma body_main:  semax_body Vprog Gprog f_main main_spec.
Proof.
name four _four.
Start_function.
forward_call (*  addVal(four,4,4); *)
  (four,Ews,four_contents,4,4).
 split3; auto. split. computable. repeat constructor; computable.
forward. (* return 0; *)
Qed.
```

From there we use the *forward_call* Hoare logic tactic, which uses the *PROP* precondition from the functional specification to provide new subgoals, and confirms that all values passed in match those used by the *WITH* clause in the specification. Our call requires an array (the global variable stored in *four*), a *share* which describes who is allowed to access the data (in our case

a defined value *Ews* which is a share that marks both reading and writing allowed), and two integers, one the size of our data, and one the value to add to each value in the array (both 4 for us).

Our precondition had five propositions within it, and we must prove them before we move on. Two of them are related to whether our chosen share is readable and writable (which we have chosen to be true) and are each dismissed with a usage of *auto* (a tactic that applies a set of known lemmas). The next two relate to the size of our given array, that it is less than the maximum integer value, which we discharge using arithmetic tactics, and that the length is indeed precisely 4, which is by definition of the construction of an array, which is represented in the *constructor* tactic. Repeating that same tactic with *computable* clears out the remaining necessary precondition that all values in the array are within the bounds of signed integers. With that requirement discharged, we now have a new proof state with the postcondition of the functional specification replacing the values as expected. With no other call other than to *return* from the function, we call *forward* one final time to complete the proof of the *main* function.

The last thing to do is to tie the functions together with a final Lemma. It begins the same for all programs:

```
Lemma all_funcs_correct:
  semax_func Vprog Gprog (prog_funct prog) Gprog.
Proof.
unfold Gprog, prog, prog_funct; simpl.
```

From there the remaining task is simple: in the same order as they were declared and proven, apply *semax_func_cons* to the lemma proving the function body:

```
semax_func_cons body_addVal.
```

```
semax_func_cons body_main.
Qed.
```

With that complete, the high level proof of the functional specifications and function bodies is complete, ignoring some details.

One can note that if we desire to treat a specific function as a black box where we do not care about the actual function body we could simply declare our desired functional specification and assert the body proof, meaning we can get intermediate levels of verification depending on whether the man hours are available to formally verify more complicated functions (such as cryptographic functions, an example of this will be seen in the proof of our program).

Next we should discuss entailments, and what they mean within Hoare logic. An entailment is a statement in the form $P \vdash Q$, where $P$ and $Q$ are sets of predicates on the state of the program. States include local variables, the contents of the heap, and potentially other external states, though those are currently not well supported within VST. Within the context of VST, $P$ and $Q$ will take a form similar to that of functional specifications, with the following form, which in essence is another way of writing $P \Rightarrow Q$:

```
ENTAIL Delta,
        PROP(P)
        LOCAL(Q)
        SEP(R)
|-- EX i : Z,
        PROP(P')
        LOCAL(Q')
        SEP(R')
```

When seen during a Coq proof, we handle these with the *entailer!* tactic, which will automatically process as much of the proof as possible, with the propositions of *P'* being checked against known hypotheses and common rewrites done (those without generated subgoals commonly). Frequently the use of *entailer!* will leave subgoals regarding *data_at* clauses, which covers the values in memory. There is a second form of entailment seen while proving loops, as stated above: that of proving that the negation of the loop-test expression. The statement must execute without crashing, all variables it references exist and are initialized, the statement doesn't divide by zero, among other requirements.

This leaves two types of tactics we need to use left to discuss: arithmetic/automation tactics and memory management tactics. Arithmetic tactics are straightforward, if sometimes a bit awkward to use. The tactics include *simpl, computable, auto,* and *omega*, all of which simplify obvious math steps or proof goals that have their statement within the proven hypotheses. Using the *replace ... with ... by ...* tactic is advised to help do simplifications of mathematical statements within proofs to avoid having to *assert* and *rewrite* by the new hypothesis.

This leaves memory management goals and list manipulation type tactics. The tactics here are too numerous to list, but in general can be discovered via *SearchAbout* within Coq and have somewhat intuitive names. We will go through an example of these sorts of proofs with a proof of an array access, the first command of the loop body.

The generated goal to prove regarding the array access is shown below (note that ++ represents a list append):

```
is_int I32 Signed
  (Znth i
```

```
     (map Vint (map (fun i0 : int => Int.add i0 (Int.repr n)) (map Int.repr (sublist 0 i
contents))) ++ map Vint (map Int.repr (sublist i (Zlength contents) contents))) Vundef)
```

In C-code, we can think of the *Znth* portion of the initial state as the following:

$(bits\_contents[0 : i] \; ++ \; bits\_contents[i : size])[i]$

The full proof body is shown below, and as above for the Hoare logic proof, we will walk through the steps to describe the changes.

```
rewrite app_Znth2.
(*  (bits_contents[i : size])[i − length(bits_contents(0 : i))] *)
repeat rewrite initial_world.Zlength_map.
rewrite Zlength_sublist; try omega.
(*  (bits_contents[i : size])[i − i] *)
rewrite Znth_map with (d':=Int.zero). hnf; auto.
(* bounds check from Znth_map *)
rewrite initial_world.Zlength_map. rewrite Zlength_sublist; try omega.
(* confirm that i is in fact more than or equal to i... from app_Znth2 *)
rewrite initial_world.Zlength_map.
rewrite map2_len.
rewrite initial_world.Zlength_map.
rewrite Zlength_sublist; try omega.
```

We also have in hand a number of assumptions, including one in particular that $i < Zlength\ contents$. In essence, this boils down to asking if the value found in a list at index *i* is an integer (i.e. in bounds). The proof may seem obvious as there is a sublist from index *i* to a value that is larger than *i*, but this is the burden of a formally verified proof: all goals, no matter how obvious, must be fully discharged to have the guarantee. A side note is that many memory

related tactics will spawn off subgoals of their own, many of which will process easily at the conclusion of the proof of the main goal.

With this in mind, let's step through the tactics used. First we should strip out the first portion of our array, since we will be able to prove that the index lies in the second part of the list, we do this via *app_Znth2*. This spawns a subgoal to prove the following:

```
i >= Zlength (map Vint (map (fun i => Int.add i (Int.repr n)) (map Int.repr (sublist 0 i
contents)) (Int.repr n)))
```

Along with progressing our current proof to the following:

```
is_int I32 Signed
  (Znth (i - Zlength (map Vint (map (fun i => Int.add i (Int.repr n)) (map Int.repr (sublist 0
i contents)) (Int.repr n))))
     (map Vint (map Int.repr (sublist i (Zlength contents) contents))) Vundef)
```

Continuing our C analogy, this gives us $(bits\_contents[i : size])[i - length(bits\_contents(0 : i))]$ as our inside state.

Next we focus on peeling the length of the sublist out of the chain of function calls, but we can not simply do this immediately. First we peel back layers of mapped functions with *initial_world.Zlength_map* usage. This leaves our sublist directly within the *Zlength* function, where we can use *Zlength_sublist* to pull out the length of the list. This tactic does produce two simple arithmetic subgoals that we are able to clear away by trying to apply *omega* to all generated subgoals. This leaves us at the following proof goal state:

```
is_int I32 Signed (Znth (i - (i - 0)) (map Vint (map Int.repr (sublist i (Zlength contents)
contents))) Vundef)
```

, which roughly equates to $(bits\_contents[i : size])[i - i]$.

From there we get rid of the *Vint* mapping a rewriting of *Znth_map*, which requires a parameter of converting the default value if the index is out of range (not an issue for us, but still a formality). The tactic *hnf* (which is a Coq tactic that reduces the current goal to its head normal form) converts the remaining goal to *True*, and it gets finished by *auto*. Since goals are added to the top of the list of subgoals, we deal with two subgoals (from *Znth_map* and *app_Znth2*) before we can move on. The *Znth_map* goal checks that the index is in range of the sublist, and clears out with *initial_world.Zlength_map* and *Zlength_sublist*. The *app_Znth2* subgoal checks that the provided index is larger than the size of the first half of the appended lists, and clears out much the same. With those subgoals handled, the memory check is complete.

Users looking for further examples can review the associated proof script and step through the goals and tactics to see the process in detail.

# Chapter 3: Full Receipt Validation Program

Now with a deeper understanding of Chaum's scheme and a fairly detailed description of how proofs proceed in the Verified Software Toolchain, we can pursue our primary focus: creating a C program to implement the receipt verification portion of Chaum's scheme, and then formally verify it using the same process just covered in the tutorial. The outcome of this effort is simply the C program to verify, the Abstract Syntax Tree created from the CompCert tool Clightgen, and the Coq .v file used to formally verify the AST from Clightgen. When the Coq file is processed start to finish, each proof will complete successfully, and thus the file as a whole will pass verification.

In the process of proving the file as a whole, we will focus primarily on a particular function: the *certifyReceipt* function. This function takes the contents of a receipt as described in chapter two as input and returns true if and only if the first two sections of the receipt verification process described in 2.1.2 are accurate for the given input (the third portion, regarding the construction of the requisite doll, is a topic for a future study). This in turn utilizes the other functions of the file, all the way down to a *deSignInt* function, which each have their behaviors proved as well. The *deSignInt* could be either a fully specified cryptographic function or simply an assumed one if we want to keep it in the trusted base.

Thus with the full stack of functions having a set of associated proofs of their function bodies and functional specifications, we can make a meaningful statement about our program's behavior. Interestingly, the final statement that we can make about our our proofs comes after defining the boundaries of our proof: the postcondition of our functional specification is the output of the function body. Ultimately, what we are interested in proving can be summed up as follows: the output of the *certifyReceipt* function is 1 if and only if the value representing the signed serial number can have the inverse signature function applied to get the original serial number and the tuple of values included as the second portion of the receipt can be designed to get their corresponding unsigned values.

We now quickly go over the source file. There is a wide array of technical detail in the functions, from a simple *xor* for the designing function (to be further touched on later) to checking the values across arrays. In terms of C code, it is not anything groundbreaking, but certain portions were constructed with the proof necessities in mind. Boolean flags were used to carry values

through a series of comparisons rather than immediately returning, and rather than branching off into two paths of computation a set of variables are assigned whose differing values create the change. Each of those choices were made to drastically reduce the complexity of the branch points within the proofs of the function body, as while they are certainly not what we might normally do when writing code, they function just the same (albeit slower) and make the proofs more manageable. The full source code can be found in appendix A.

# Chapter 4: Verification of the Program

As mentioned before, certain sections of the C code were determined by how to make the proofs easier. That said, certain parts of the VST Coq structures still provide limits and constraints on what we can do within the proof proper. Additionally, as the proof got stronger and stronger as we extended what the function is meant to prove, the pre- and post-conditions get progressively more complicated, which will cause some issues as seen below.

## 4.1 Coq File

For our verification, much of it closely follows the tutorial proof above. We use the *clightgen* utility to transform the C file into the Abstract Syntax Tree in Coq, then importing that into our verification file. From there, we define our set of functional specifications, which bound the strength of our proof bodies. These functional specifications vary from the simple as seen in portions of the tutorial to the complex, which we see when we get to the functional specifications of the *certifyReceipt* function. Interspersed within our functional specifications are a set of a Coq

definition and lemmas that help define what our goals are within those specifications. This definition is an integer array equality definition, and have had various lemmas proven to assist in their operation while in the middle of a list.

### 4.1.1 Functional Specifications

The functional specifications for the set of functions describing the first portion of our desired proof (that the serial is the same as the value that should represent the signed serial with the designature function applied) are straightforward: since we've defined our *deSignInteger* function to be an integer *xor* at the top of the Coq file, the *deSignInt* function returns the *xor* of the input value and the provided key, and the *checkSerial* function takes the original serial, the provided value, and the needed key, see below.

```
Definition deSignInt_spec :=
  DECLARE _deSignInt
   WITH bit : int, invKey : int
   PRE [_bit OF (tint), _invKey OF (tint)]
       PROP ()
       LOCAL (temp _bit (Vint bit); temp _invKey (Vint invKey))
       SEP ()
   POST [tint]
       PROP()
       LOCAL (temp ret_temp (Vint (deSignInteger bit invKey)))
       SEP().

Definition checkSerial_spec :=
  DECLARE _checkSerial
   WITH q : int, qSigned : int, invKey : int
   PRE [_q OF (tint), _qSigned OF (tint), _invKey OF (tint)]
     PROP ()
     LOCAL (temp _q (Vint q); temp _qSigned (Vint qSigned); temp _invKey (Vint invKey))
     SEP ()
   POST [tint]
     PROP ()
```

```
     LOCAL (temp ret_temp (Vint (if Int.eq q (deSignInteger qSigned invKey) then Int.repr 1
else Int.repr 0)))
     SEP ().
```

The next set of functions in our C file describe those needed to compare two arrays of values, as we have three sets of arrays that will need to be checked in the second portion of our target goal (the layer, top doll, and bottom doll, each with their respective list that we expect to represent the signed versions). This requires an additional way of defining a map (one that takes an additional value that we apply on every index in the list), which we borrow from the tutorial. Then we extend the designing function across an array, since if we have an array of integers the function just extends across the array we are in luck here as seen below.

```
Definition deSignArray_spec :=
  DECLARE _deSignArray
    WITH bits: val, sh: share, contents : list Z, invKey : Z, size: Z
    PRE [ _bits OF (tptr tint), _invKey OF tint, _size OF tint ]
          PROP  (readable_share sh; writable_share sh; 0 <= size < Int.max_signed;
                 size = Zlength contents;
                 Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) contents)
          LOCAL (temp _bits bits; temp _invKey (Vint (Int.repr invKey)); temp _size (Vint
(Int.repr size)))
          SEP   (data_at sh (tarray tint size) (map Vint (map Int.repr contents)) bits)
    POST [tvoid] PROP () LOCAL ()
      SEP (data_at sh (tarray tint size)
        (map Vint (map (fun i => deSignInteger i (Int.repr invKey))
                  (map Int.repr contents))) bits).
```

Our functional specification for this function is quite similar to that of the mapAdd function in the tutorial, with a *deSignInteger* swapped in for the *add* of that. The more interesting portion is the checkArray function, where we loop over an array and compile the information piece by piece that the two arrays match at each index.

The functional specification has grown with the number of variables involved in the function, but the function size is still manageable with the use of our custom fixpoint. One thing to note here is that despite the fact that we never directly change the arrays in the *checkArray* function, the data in the arrays still changes since we call a function that does change those values, so we must account for that in the specification. The return value of this function is straightforward as well with our fixpoint, it directly correlates to the output. Below shows just the postcondition of the *checkArray* function, since the precondition can easily be derived from the expected values.

```
Definition checkArray_spec :=
  DECLARE _checkArray
    WITH signedBits: val, originalBits: val, sh: share,
         signedContents : list Z, originalContents : list Z, invKey : Z, size: Z
...
    POST [tint]
      PROP  ()
      LOCAL (temp ret_temp (Vint (if (foralleq (*signedContents originalContents)*)
                  (map (fun i => deSignInteger i (Int.repr invKey)) (map Int.repr
signedContents)))
                  (map Int.repr originalContents)
                then Int.repr 1 else Int.repr 0)))
      SEP   ((data_at sh (tarray tint size)
               (map Vint (map (fun i => deSignInteger i (Int.repr invKey)) (map Int.repr
signedContents))) signedBits);
             (data_at sh (tarray tint size) (map Vint (map Int.repr originalContents))
originalBits)).
```

With that out of the way, we can do the work behind checking that the full tuple has all of the values matching properly. This is where we run into a bit of an odd problem with the VST module within Coq: there is a limit of fourteen arguments to a functional specification. If a function starts to approach that many arguments, this becomes an issue, as *share* values along with the existence of two values in the *WITH* clause for each array value can easily push the total Coq parameters over the limit. The solution we use is simply to combine these values into

tuples. While it makes the specification slightly harder to read, the input values remain the same, we just need to parse them out of the tuples prior to their usage, see the beginning of the functional specification below:

```
Definition checkTuple_spec :=
  DECLARE _checkTuple
    WITH layer_var: (val * list Z), q: Z, topDoll_var: (val * list Z),
         botDoll_var: (val * list Z), v: Z,
         k_var: (val * list Z), r: Z, i_var: (val * list Z),
         j_var: (val * list Z), w: Z,
         sh: share, key: Z, size: Z
```

Once that hurdle has been overcome, the specification does follow closely from the other specifications, with the only change being the chaining of conditionals (since we need to check each of the five values within the signed tuple) using *andb* in the return condition. Everything else becomes a matter of scale: frustrating to deal with, but doesn't present much of a mental hurdle to get past. The postcondition is shown below to show the chainings of conditionals and the multiple modified arrays from the calls within.

```
    POST [tint]
      PROP ()
      LOCAL (temp ret_temp (Vint (if (
              andb (foralleq (map (fun i => deSignInteger i (Int.repr key)) (map Int.repr (snd
k_var)))
                (map Int.repr (snd layer_var)))
              (andb (Int.eq (Int.repr q) (deSignInteger (Int.repr r) (Int.repr key)))
              (andb (foralleq (map (fun i => deSignInteger i (Int.repr key)) (map Int.repr
(snd i_var)))
                (map Int.repr (snd topDoll_var)))
              (andb (foralleq (map (fun i => deSignInteger i (Int.repr key)) (map Int.repr
(snd j_var)))
                (map Int.repr (snd botDoll_var)))
              ((Int.eq (Int.repr v) (deSignInteger (Int.repr w) (Int.repr key))))))))
                then Int.repr 1 else Int.repr 0)))
      SEP   ((data_at sh (tarray tint size)
```

```
              (map Vint (map (fun i => deSignInteger i (Int.repr key)) (map Int.repr (snd
k_var)))) (fst k_var));
            (data_at sh (tarray tint size) (map Vint (map Int.repr (snd layer_var))) (fst
layer_var));
            (data_at sh (tarray tint size)
              (map Vint (map (fun i => deSignInteger i (Int.repr key)) (map Int.repr (snd
i_var)))) (fst i_var));
            (data_at sh (tarray tint size) (map Vint (map Int.repr (snd topDoll_var))) (fst
topDoll_var));
            (data_at sh (tarray tint size)
              (map Vint (map (fun i => deSignInteger i (Int.repr key)) (map Int.repr (snd
j_var)))) (fst j_var));
            (data_at sh (tarray tint size) (map Vint (map Int.repr (snd botDoll_var))) (fst
botDoll_var))).
```

We now have our segments for the two portions of our proof, the only thing left to do is combine them and we have a function to certify receipts. However, there are some additional considerations for certifying a receipt, primarily that we have to be able to verify both a receipt with the bottom layer chosen and a receipt with the top layer chosen. We can pass in a flag that shows which one to use along with values for each of the four keys (one for tuple, one for serial for both top and bottom), but the conditions and changes to the arrays will depend upon the which layer we have been given. While this is handled in five lines of the C code, it makes our functional specification a staggering 99 lines in length. The precondition follows from the other functions in an increase in complexity, but again only in length and the number of variables we need to check on. The postcondition is where the struggles lay though, as the output is dependent on that split. The return variable splits based upon the flag and uses the requisite variables within the composite check of the serial and tuple rather simply, but there is a struggle with the array contents' postcondition.

```
POST [tint]
```

```
      PROP ()
      LOCAL (...)
      if (Int.eq (Int.repr (fst topFlag_size)) (Int.repr 1))
             then
       SEP   ((data_at sh (tarray tint (snd topFlag_size))
               (map Vint (map (fun i => deSignInteger i (Int.repr topTupleKey)) (map Int.repr
(snd k_var)))) (fst k_var));
             (data_at sh (tarray tint (snd topFlag_size)) (map Vint (map Int.repr (snd
layer_var))) (fst layer_var));
             (data_at sh (tarray tint (snd topFlag_size))
               (map Vint (map (fun i => deSignInteger i (Int.repr topTupleKey)) (map Int.repr
(snd i_var)))) (fst i_var));
             (data_at sh (tarray tint (snd topFlag_size)) (map Vint (map Int.repr (snd
topDoll_var))) (fst topDoll_var));
             (data_at sh (tarray tint (snd topFlag_size))
               (map Vint (map (fun i => deSignInteger i (Int.repr topTupleKey)) (map Int.repr
(snd j_var)))) (fst j_var));
             (data_at sh (tarray tint (snd topFlag_size)) (map Vint (map Int.repr (snd
botDoll_var))) (fst botDoll_var)))
             else
      SEP   ((data_at sh (tarray tint (snd topFlag_size))
               (map Vint (map (fun i => deSignInteger i (Int.repr botTupleKey)) (map Int.repr
(snd k_var)))) (fst k_var));
             (data_at sh (tarray tint (snd topFlag_size)) (map Vint (map Int.repr (snd
layer_var))) (fst layer_var));
             (data_at sh (tarray tint (snd topFlag_size))
               (map Vint (map (fun i => deSignInteger i (Int.repr botTupleKey)) (map Int.repr
(snd i_var)))) (fst i_var));
             (data_at sh (tarray tint (snd topFlag_size)) (map Vint (map Int.repr (snd
topDoll_var))) (fst topDoll_var));
             (data_at sh (tarray tint (snd topFlag_size))
               (map Vint (map (fun i => deSignInteger i (Int.repr botTupleKey)) (map Int.repr
(snd j_var)))) (fst j_var));
             (data_at sh (tarray tint (snd topFlag_size)) (map Vint (map Int.repr (snd
botDoll_var))) (fst botDoll_var))).
```

Attempting to split within the *SEP* clause generates unexpected and strange errors that are difficult to adequately address, which is when it becomes valuable to learn that the *POST* clause can have an *if* statement return different *SEP* clauses and not be an issue. With that in hand, we can now split our newly mapped arrays based on the appropriate key. Since the *main* function's functional specification is the same for all files, this completes the first portion of our proof discussion, leaving just the bodies of the proofs to examine. See appendix B for the full functional specifications.

As stated in the tutorial, we prove the functions in the order they are defined, which means we start out with some proofs that are very similar to those we saw in the tutorial, or are otherwise fairly simple.

### 4.1.2 Validation Program Proof

With our functional specifications in hand, we can go about proving each of our functions to have that specification. We will skim over the opening proofs of *deSignInt*, *checkSerial*, and *deSignArray*, as each of those are either simply very straight forward or derivative of work seen in the tutorial. The code becomes more interesting after that though, with *checkArray* requiring a new property to work with and more advanced loop behavior. The property *foralleq* is true if and only if all values in two lists are the same, and we want to build this property up by iterating through the lists. This is straightforward in the C code, but the list manipulation in Coq presents issues for us, mostly due to needing to prove that both arrays provide an integer to compare. Ultimately, unfolding values within one of the hypotheses enabled us to get to a point where the proof goal can simplify easily. With that in hand, the rest of the proof follows much as the others.

The other two proofs that we have to focus on are our *checkTuple* function and our *certifyReceipt* functions. The *checkTuple* function proof is in fact as straightforward as we could possibly hope, we call each function in turn as in the source code and use the preconditions of the *checkTuple* function to prove each of the called functions' preconditions. To prove the *checkTuple* function, we simply destruct on each of the conditions of the tuple check. By chaining the commands together, we fortunately can avoid having to prove each case individually, and this applies to *certifyReceipt* as well.

We do largely the same thing for *certifyReceipt,* with a difference being that we need to decide which of our keys to be used for checking the tuple and serial number. Once we have done that, we can continue, with conditionals determining which of our keys to use. We once again destruct on each of our conditionals, for the return value, then again on determining the final values of the arrays. With that finished, our proof is complete.

# Chapter 5: Future Work

Some may note that we have used a simple *xor* function as our function to design integers and arrays and that as such the code is not cryptographically secure. However, it can be noted that this still fulfills the specification of Chaum's scheme in the manner of the signing functions for both the serial number and the tuple of data. We can simply take the *xor* function to be the provided designing function, as both the $o^{-1}$ and $s^{-1}$ functions and our defined hardcoded keys to be the provided keys of those functions. While this is not necessarily an implementation of the verification and scheme that would want to be used in a production server, it does serve an important role in the path of progress. With this proof now in hand others can improve on the

functions used, replacing the insecure *xor* with a full fledged cryptographic function, and so long as the functional specification still fits (which it will for any function of type *Int -> Int*), the only thing that will need to be updated is the proof of that designing function.

Additionally, with this proof there is a motivation to provide such a proof for a cryptographic designing function - it immediately provides value when put in combination with this proof, naturally expanding the value that this initial set of proofs has provided. The nice thing is that for our purposes the signing function (along with the decryption function) does not require a proof - those are taken as black boxes that we hope are working properly but for our purposes it does not matter, if not then our verification function will simply fail.

In order to bring the code and proof to fully implement a more secure Chaum's scheme, replace the *deSignInteger* function definition with a full designing algorithm. When proving the function body of *deSignInt*, simply expand the function definition to get the entire description available and to sanity check that the result of the function is matching the progress of the proof. The rest of the proof body should remain constant, without any need to consider the change.

# Chapter 6: Conclusion

Our goal when we began this thesis was the formal verification of a program designed to verify portions of a receipt given as the output of Chaum's scheme. In doing so, we hoped to provide a framework for others to use as a launching pad for future endeavors along a similar branch of research and produce an example of VST being applied to a practical real-world algorithm that has a more entry level friendly description than that of the SHA-256 paper.

In this we have mostly succeeded - we have finished a more thorough tutorial as designed, however we fell a touch short in our hopes for the verification of the validation function. An observant eye would note that there are three pieces necessary to validate a receipt, and our *certifyReceipt* function only has the first two of those. Ultimately this would not solve any additional interesting problems though, as we would once again be using placeholders for both the cryptographic hash function and the encryption function and the proofs for them would be time consuming but in the same vein as those already completed.

# Appendix A: verify.c Source Code

```
/*
 * Verification program for Chaum's scheme
 * @author kms7341@rit.edu
 */

// note on array lengths: chaum's scheme can operate at any scale
// we've chosen size = 8 since that is reasonably large without
// imposing undue costs of size and time


int deSignInt(int bit, int invKey) {
  return bit ^ invKey;
}

// only need deSign for first two portions

int checkSerial(int q, int qSigned, int invKey) {
  int q2;
  q2 = deSignInt(qSigned, invKey);
  if (q2 == q)
    return 1;
  return 0;
}

//next set of conditions: needs designArray

void deSignArray(int bits[], int invKey, int size) {
  int i = 0;
  int bit = 0;
  int ans = 0;
  while (i < size) {
    bit = bits[i];
    ans = deSignInt(bit, invKey);
    bits[i] = ans;
```

```
    i++;
  }
}

int checkArray(int signedBits[], int originalBits[], int invKey, int size) {
  deSignArray(signedBits, invKey, size);
  int i = 0;
  int bit1 = 0;
  int bit2 = 0;
  int flag = 1;
  while (i < size) {
    bit1 = signedBits[i];
    bit2 = originalBits[i];
    if (bit1 != bit2) {
      flag = 0;
    }
    i++;
  }
  return flag;
}

int checkTuple(int layer[], int q, int topDoll[], int botDoll[], int v,
       int k[], int r, int i[], int j[], int w, int key, int size) {
  int flag = 1;
  int val = 0;
  val = checkArray(k, layer, key, size);
  flag = flag & val;
  val = checkSerial(q, r, key);
  flag = flag & val;
  val = checkArray(i, topDoll, key, size);
  flag = flag & val;
  val = checkArray(j, botDoll, key, size);
  flag = flag & val;
  val = checkSerial(v, w, key);
  flag = flag & val;
  return flag;
}

int certifyReceipt(int topFlag, int layer[], int q, int topDoll[], int botDoll[], int v,
       int k[], int r, int i[], int j[], int w, int size, // size is constant for all arrays
       int topSerialKey, int botSerialKey, int topTupleKey, int botTupleKey) {
  int flag = 1;
  int val = 0;
  int serKey = 0;
  int tupKey = 0;
  if (topFlag == 1) {
    serKey = topSerialKey;
    tupKey = topTupleKey;
  } else {
    serKey = botSerialKey;
    tupKey = botTupleKey;
  }
  val = checkSerial(q, v, serKey);
  flag = flag & val;
  val = checkTuple(layer, q, topDoll, botDoll, v,
    k, r, i, j, w, tupKey, size);
  flag = flag & val;
  return flag;
}

// arbitrary values
```

```
int qVals[8] = {15, 5463, 12, 75, 231, 1431, 735, 134};
int processedQVals[8] = {15, 5463, 12, 75, 231, 1431, 735, 134};

int main(void) {
  int qsigned, ser;
  int q = 15687;
  int key = 4231;
  qsigned = deSignInt(q, key); // xor makes algo same both ways
  ser = checkSerial(q, qsigned, key);
  deSignArray(processedQVals, key, 8);
  int arr = checkArray(processedQVals, qVals, key, 8);
  return ser;
}
```

# Appendix B: Functional Specifications

```
Definition deSignInt_spec :=
  DECLARE _deSignInt
   WITH bit : int, invKey : int
   PRE [_bit OF (tint), _invKey OF (tint)]
       PROP ()
       LOCAL (temp _bit (Vint bit); temp _invKey (Vint invKey))
       SEP ()
   POST [tint]
       PROP()
       LOCAL (temp ret_temp (Vint (deSignInteger bit invKey)))
       SEP().

Definition checkSerial_spec :=
  DECLARE _checkSerial
   WITH q : int, qSigned : int, invKey : int
   PRE [_q OF (tint), _qSigned OF (tint), _invKey OF (tint)]
     PROP ()
     LOCAL (temp _q (Vint q); temp _qSigned (Vint qSigned); temp _invKey (Vint invKey))
     SEP ()
   POST [tint]
     PROP ()
     LOCAL (temp ret_temp (Vint (if Int.eq q (deSignInteger qSigned invKey) then Int.repr 1
else Int.repr 0)))
     SEP ().

Local Open Scope logic.

Definition deSignArray_spec :=
  DECLARE _deSignArray
    WITH bits: val, sh: share, contents : list Z, invKey : Z, size: Z
    PRE [ _bits OF (tptr tint), _invKey OF tint, _size OF tint ]
          PROP  (readable_share sh; writable_share sh; 0 <= size < Int.max_signed; size =
Zlength contents;
                 Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) contents)
          LOCAL (temp _bits bits; temp _invKey (Vint (Int.repr invKey)); temp _size (Vint
(Int.repr size)))
          SEP   (data_at sh (tarray tint size) (map Vint (map Int.repr contents)) bits)
    POST [tvoid] PROP () LOCAL ()
      SEP (data_at sh (tarray tint size)
        (map Vint (map (fun i => deSignInteger i (Int.repr invKey)) (map Int.repr contents)))
bits).

Fixpoint foralleq (al: list int) (bl: list int) : bool :=
```

```
  match al with
    | a::al' => match bl with
                | b::bl' => (Int.eq a b) && (foralleq al' bl')
                | _ => false
                end
    | _ => match bl with
           | b::bl' => false
           | _ => true
           end
  end.

Definition checkArray_spec :=
  DECLARE _checkArray
    WITH signedBits: val, originalBits: val, sh: share,
         signedContents : list Z, originalContents : list Z, invKey : Z, size: Z
    PRE [ _signedBits OF (tptr tint), _originalBits OF (tptr tint), _invKey OF tint, _size OF
tint ]
          PROP  (readable_share sh; writable_share sh; 0 <= size < Int.max_signed;
                 size = Zlength signedContents; size = Zlength originalContents;
                 Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) signedContents;
                 Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) originalContents)
          LOCAL (temp _signedBits signedBits; temp _originalBits originalBits;
                 temp _invKey (Vint (Int.repr invKey)); temp _size (Vint (Int.repr size)))
          SEP   ((data_at sh (tarray tint size) (map Vint (map Int.repr signedContents))
signedBits);
                 (data_at sh (tarray tint size) (map Vint (map Int.repr originalContents))
originalBits))
    POST [tint]
      PROP  ()
      LOCAL (temp ret_temp (Vint (if (foralleq (*signedContents originalContents)*)
                  (map (fun i => deSignInteger i (Int.repr invKey)) (map Int.repr
signedContents)))
                  (map Int.repr originalContents)
                 then Int.repr 1 else Int.repr 0)))
      SEP   ((data_at sh (tarray tint size)
                (map Vint (map (fun i => deSignInteger i (Int.repr invKey)) (map Int.repr
signedContents))) signedBits);
             (data_at sh (tarray tint size) (map Vint (map Int.repr originalContents))
originalBits)).

Definition checkTuple_spec :=
  DECLARE _checkTuple
    WITH layer_var: (val * list Z), q: Z, topDoll_var: (val * list Z),
         botDoll_var: (val * list Z), v: Z,
         k_var: (val * list Z), r: Z, i_var: (val * list Z),
         j_var: (val * list Z), w: Z,
         sh: share, key: Z, size: Z
         (* layer: val, layer_bits: list Z, q: Z, topDoll: val, topDoll_bits: list Z,
         botDoll: val, botDoll_bits: list Z, v: Z,
         k: val, k_bits: list Z, r: Z, i: val, i_bits: list Z,
         j: val, j_bits: list Z, w: Z,
         sh: share, key: Z, size: Z *)
    PRE [ _layer OF (tptr tint), _q OF tint, _topDoll OF (tptr tint), _botDoll OF (tptr tint),
_v OF tint,
          _k OF (tptr tint), _r OF tint, _i OF (tptr tint), _j OF (tptr tint), _w OF tint,
          _key OF tint, _size OF tint ]
          PROP  (readable_share sh; writable_share sh; 0 <= size < Int.max_signed;
                 size = Zlength (snd layer_var); size = Zlength (snd topDoll_var);
                 size = Zlength (snd botDoll_var);
                 size = Zlength (snd k_var); size = Zlength (snd i_var); size = Zlength (snd
j_var);
```

```
                Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) (snd layer_var);
                Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) (snd
topDoll_var);
                Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) (snd
botDoll_var);
                Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) (snd k_var);
                Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) (snd i_var);
                Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) (snd j_var))
          LOCAL (temp _layer (fst layer_var); temp _q (Vint (Int.repr q)); temp _topDoll (fst
topDoll_var);
                temp _botDoll (fst botDoll_var); temp _v (Vint (Int.repr v));
                temp _k (fst k_var); temp _r (Vint (Int.repr r)); temp _i (fst i_var);
                temp _j (fst j_var); temp _w (Vint (Int.repr w));
                temp _key (Vint (Int.repr key)); temp _size (Vint (Int.repr size)))
          SEP ((data_at sh (tarray tint size) (map Vint (map Int.repr (snd layer_var))) (fst
layer_var));
               (data_at sh (tarray tint size) (map Vint (map Int.repr (snd topDoll_var))) (fst
topDoll_var));
               (data_at sh (tarray tint size) (map Vint (map Int.repr (snd botDoll_var))) (fst
botDoll_var));
               (data_at sh (tarray tint size) (map Vint (map Int.repr (snd k_var))) (fst
k_var));
               (data_at sh (tarray tint size) (map Vint (map Int.repr (snd i_var))) (fst
i_var));
               (data_at sh (tarray tint size) (map Vint (map Int.repr (snd j_var))) (fst
j_var)))
    POST [tint]
      PROP ()
      LOCAL (temp ret_temp (Vint (if (
              andb (foralleq (map (fun i => deSignInteger i (Int.repr key)) (map Int.repr (snd
k_var)))
                (map Int.repr (snd layer_var)))
              (andb (Int.eq (Int.repr q) (deSignInteger (Int.repr r) (Int.repr key)))
              (andb (foralleq (map (fun i => deSignInteger i (Int.repr key)) (map Int.repr
(snd i_var)))
                (map Int.repr (snd topDoll_var)))
              (andb (foralleq (map (fun i => deSignInteger i (Int.repr key)) (map Int.repr
(snd j_var)))
                (map Int.repr (snd botDoll_var)))
              ((Int.eq (Int.repr v) (deSignInteger (Int.repr w) (Int.repr key))))))))
                then Int.repr 1 else Int.repr 0)))
      SEP   ((data_at sh (tarray tint size)
                (map Vint (map (fun i => deSignInteger i (Int.repr key)) (map Int.repr (snd
k_var)))) (fst k_var));
             (data_at sh (tarray tint size) (map Vint (map Int.repr (snd layer_var))) (fst
layer_var));
             (data_at sh (tarray tint size)
                (map Vint (map (fun i => deSignInteger i (Int.repr key)) (map Int.repr (snd
i_var)))) (fst i_var));
             (data_at sh (tarray tint size) (map Vint (map Int.repr (snd topDoll_var))) (fst
topDoll_var));
             (data_at sh (tarray tint size)
                (map Vint (map (fun i => deSignInteger i (Int.repr key)) (map Int.repr (snd
j_var)))) (fst j_var));
             (data_at sh (tarray tint size) (map Vint (map Int.repr (snd botDoll_var))) (fst
botDoll_var))).

Definition certifyReceipt_spec :=
  DECLARE _certifyReceipt (* long spec *)
    WITH layer_var: (val * list Z), topDoll_var: (val * list Z),
         botDoll_var: (val * list Z), q_v: (Z * Z),
```

```
        k_var: (val * list Z), i_var: (val * list Z),
        j_var: (val * list Z), r_w: (Z * Z), topFlag_size: (Z * Z),
        sh: share, topSerialKey: Z, botSerialKey: Z, topTupleKey: Z, botTupleKey: Z
    PRE [ _topFlag OF tint,
          _layer OF (tptr tint), _q OF tint, _topDoll OF (tptr tint), _botDoll OF (tptr tint),
_v OF tint,
          _k OF (tptr tint), _r OF tint, _i OF (tptr tint), _j OF (tptr tint), _w OF tint,
_size OF tint,
          _topSerialKey OF tint, _botSerialKey OF tint, _topTupleKey OF tint, _botTupleKey OF
tint ]
          PROP  (readable_share sh; writable_share sh; 0 <= (snd topFlag_size) <
Int.max_signed;
                 (snd topFlag_size) = Zlength (snd layer_var);
                 (snd topFlag_size) = Zlength (snd topDoll_var);
                 (snd topFlag_size) = Zlength (snd botDoll_var);
                 (snd topFlag_size) = Zlength (snd k_var);
                 (snd topFlag_size) = Zlength (snd i_var);
                 (snd topFlag_size) = Zlength (snd j_var);
                 Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) (snd layer_var);
                 Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) (snd
topDoll_var);
                 Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) (snd
botDoll_var);
                 Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) (snd k_var);
                 Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) (snd i_var);
                 Forall (fun bit => Int.min_signed <= bit <= Int.max_signed) (snd j_var))
          LOCAL (temp _layer (fst layer_var); temp _q (Vint (Int.repr (fst q_v)));
                 temp _topDoll (fst topDoll_var);
                 temp _botDoll (fst botDoll_var); temp _v (Vint (Int.repr (snd q_v)));
                 temp _k (fst k_var); temp _r (Vint (Int.repr (fst r_w))); temp _i (fst
i_var);
                 temp _j (fst j_var); temp _w (Vint (Int.repr (snd r_w)));
                 temp _topFlag (Vint (Int.repr (fst topFlag_size)));
                 temp _topSerialKey (Vint (Int.repr topSerialKey));
                 temp _botSerialKey (Vint (Int.repr botSerialKey));
                 temp _topTupleKey (Vint (Int.repr topTupleKey));
                 temp _botTupleKey (Vint (Int.repr botTupleKey));
                 temp _size (Vint (Int.repr (snd topFlag_size))))
          SEP ((data_at sh (tarray tint (snd topFlag_size))
                 (map Vint (map Int.repr (snd layer_var))) (fst layer_var));
               (data_at sh (tarray tint (snd topFlag_size))
                (map Vint (map Int.repr (snd topDoll_var))) (fst topDoll_var));
               (data_at sh (tarray tint (snd topFlag_size))
                (map Vint (map Int.repr (snd botDoll_var))) (fst botDoll_var));
               (data_at sh (tarray tint (snd topFlag_size))
                (map Vint (map Int.repr (snd k_var))) (fst k_var));
               (data_at sh (tarray tint (snd topFlag_size))
                (map Vint (map Int.repr (snd i_var))) (fst i_var));
               (data_at sh (tarray tint (snd topFlag_size))
                (map Vint (map Int.repr (snd j_var))) (fst j_var)))
    POST [tint]
      PROP ()
      LOCAL (temp ret_temp (Vint (if (Int.eq (Int.repr (fst topFlag_size)) (Int.repr 1))
            then (if
              (andb (Int.eq (Int.repr (fst q_v)) (deSignInteger (Int.repr (snd q_v)) (Int.repr
topSerialKey)))
              (andb (foralleq (map (fun i => deSignInteger i (Int.repr topTupleKey)) (map
Int.repr (snd k_var)))
                (map Int.repr (snd layer_var)))
              (andb (Int.eq (Int.repr (fst q_v)) (deSignInteger (Int.repr (fst r_w)) (Int.repr
topTupleKey)))
```

```
              (andb (foralleq (map (fun i => deSignInteger i (Int.repr topTupleKey)) (map
Int.repr (snd i_var)))
                (map Int.repr (snd topDoll_var)))
              (andb (foralleq (map (fun i => deSignInteger i (Int.repr topTupleKey)) (map
Int.repr (snd j_var)))
                (map Int.repr (snd botDoll_var)))
              ((Int.eq (Int.repr (snd q_v)) (deSignInteger (Int.repr (snd r_w)) (Int.repr
topTupleKey)))))))))
                 then (Int.repr 1) else (Int.repr 0))

            else (if
              (andb (Int.eq (Int.repr (fst q_v)) (deSignInteger (Int.repr (snd q_v)) (Int.repr
botSerialKey)))
              (andb (foralleq (map (fun i => deSignInteger i (Int.repr botTupleKey)) (map
Int.repr (snd k_var)))
                (map Int.repr (snd layer_var)))
              (andb (Int.eq (Int.repr (fst q_v)) (deSignInteger (Int.repr (fst r_w)) (Int.repr
botTupleKey)))
              (andb (foralleq (map (fun i => deSignInteger i (Int.repr botTupleKey)) (map
Int.repr (snd i_var)))
                (map Int.repr (snd topDoll_var)))
              (andb (foralleq (map (fun i => deSignInteger i (Int.repr botTupleKey)) (map
Int.repr (snd j_var)))
                (map Int.repr (snd botDoll_var)))
              ((Int.eq (Int.repr (snd q_v)) (deSignInteger (Int.repr (snd r_w)) (Int.repr
botTupleKey)))))))))
                 then (Int.repr 1) else (Int.repr 0)))))
      if (Int.eq (Int.repr (fst topFlag_size)) (Int.repr 1))
            then
       SEP   ((data_at sh (tarray tint (snd topFlag_size))
                (map Vint (map (fun i => deSignInteger i (Int.repr topTupleKey)) (map Int.repr
(snd k_var)))) (fst k_var));
              (data_at sh (tarray tint (snd topFlag_size)) (map Vint (map Int.repr (snd
layer_var))) (fst layer_var));
              (data_at sh (tarray tint (snd topFlag_size))
                (map Vint (map (fun i => deSignInteger i (Int.repr topTupleKey)) (map Int.repr
(snd i_var)))) (fst i_var));
              (data_at sh (tarray tint (snd topFlag_size)) (map Vint (map Int.repr (snd
topDoll_var))) (fst topDoll_var));
              (data_at sh (tarray tint (snd topFlag_size))
                (map Vint (map (fun i => deSignInteger i (Int.repr topTupleKey)) (map Int.repr
(snd j_var)))) (fst j_var));
              (data_at sh (tarray tint (snd topFlag_size)) (map Vint (map Int.repr (snd
botDoll_var))) (fst botDoll_var)))
            else
      SEP   ((data_at sh (tarray tint (snd topFlag_size))
                (map Vint (map (fun i => deSignInteger i (Int.repr botTupleKey)) (map Int.repr
(snd k_var)))) (fst k_var));
              (data_at sh (tarray tint (snd topFlag_size)) (map Vint (map Int.repr (snd
layer_var))) (fst layer_var));
              (data_at sh (tarray tint (snd topFlag_size))
                (map Vint (map (fun i => deSignInteger i (Int.repr botTupleKey)) (map Int.repr
(snd i_var)))) (fst i_var));
              (data_at sh (tarray tint (snd topFlag_size)) (map Vint (map Int.repr (snd
topDoll_var))) (fst topDoll_var));
              (data_at sh (tarray tint (snd topFlag_size))
                (map Vint (map (fun i => deSignInteger i (Int.repr botTupleKey)) (map Int.repr
(snd j_var)))) (fst j_var));
              (data_at sh (tarray tint (snd topFlag_size)) (map Vint (map Int.repr (snd
botDoll_var))) (fst botDoll_var))).
```

```
Definition main_spec :=
 DECLARE _main
  WITH u : unit
  PRE  [] main_pre prog u
  POST [ tint ] main_post prog u.
```

# References

[1] : Staub, Julie Ann. "AN ANALYSIS OF CHAUM'S VOTER-VERIFIABLE ELECTION SCHEME." Thesis. University of Maryland, 2005. Web

[2]: Appel, A. W, Verified software toolchain. In ESOP '11: Proceedings of the 20th European Conference on Programming Languages and Systems, 2011.

[3] : Appel, A.W., "Verification of a Cryptographic Primitive: SHA-256", ACM Transactions on Programming Languages and Systems (TOPLAS), 2015.

[4]: Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In Symp. on Formal Methods, pages 460–475, 2006

[5]: D. Chaum. Presentation: Secret-Ballot Receipts: True Voter-Verifiable Elections. DIMACS Workshop on Electronic Voting, 2004. Available at https://people.csail.mit.edu/rivest/voting/papers/Chaum-SecretBallotReceiptsTrueVoterVerifiableElections.pdf

[6]. Appel, Andrew W. *Program Logics for Certified Compilers*. New York, NY: Cambridge UP, 2014. *Verified Software Toolchain*. Princeton University, 27 July 2016. Web. 18 May 2017. http://vst.cs.princeton.edu/download/VC.pdf

[7]. Pierce, Benjamin C. "Software Foundations." *Software Foundations*. University of Pennsylvania, Spring 2017. Web. 18 May 2017. http://www.cis.upenn.edu/~bcpierce/sf/current/index.html

[8]. *VITA Security Assessment of WINVote Voting Equipment* http://www.elections.virginia.gov/WebDocs/VotingEquipReport/WINVote-final.pdf

[9]. Bruce, Schneier. "The Problem with Electronic Voting Machines." *Blog*. N.p., 10 Nov. 2004. Web. 18 May 2017. https://www.schneier.com/blog/archives/2004/11/the_problem_wit.html

[10]. Horwitz, Sari. "More than 30 States Offer Online Voting, but Experts Warn It Isn't Secure." *The Washington Post*. WP Company, 17 May 2016. Web. 18 May 2017.

[11]. Wolchok, Scott, Eric Wustrow, Dawn Isabel, and Alex J. Halderman. "Attacking the Washington, D.C. Internet Voting System." *Financial Cryptography and Data Security Lecture Notes in Computer Science* (2012): 114-28. Feb. 2012. Web. 18 May 2017.

[12]. Syal, Rajeev. "John Bercow Calls for Online Voting in 2020 General Election." *The Guardian*. Guardian News and Media, 26 Jan. 2015. Web. 18 May 2017.

[13]. M. Naor and A. Shamir, "Visual Cryptography," Proc. Advances in Cryptology (Eurocrypt 94), A. De Santis, ed., LNCS 950, Springer-Verlag, 1995, pp. 1–12.

[14]. Stajano, Frank. *Visual Cryptography Kit*. N.p., n.d. Web. 18 May 2017.

[15]. Ateniese, Giuseppe, Carlo Blundo, Alfredo De Santis, and Douglas R. Stinson. "Extended Capabilities for Visual Cryptography." *Extended Capabilities for Visual Cryptography - ScienceDirect*. N.p., 6 Jan. 2001. Web. 18 May 2017.

[16]. B. Lee and K. Kim. Receipt-free Electronic Voting Scheme with a Tamper-Resistent Randomizer. In ICISC `02, pgs. 405-422, 2002.

[17]. Necula, George C., and Peter Lee. "Proof-Carrying Code." (n.d.): n. pag. Nov. 1996. Web. 18 May 2017. http://www.utdallas.edu/~kxh060100/Papers/necula96.pdf