Department Of Computer Science

University of Bristol

Compact E-Cash

Alireza Hajabbasgholi

A.Hajabbasgholi.05@bristol.ac.uk

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering

Declaration

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Alireza Hajabbasgholi, April 2006

Acknowledgements

Firstly and foremostly, I would like to thank Professor Nigel Smart, who has been the best tutor and supervisor one could wish for. As well as going out of his way to help me when difficulties arose, his moral support kept me on the right path. I am very grateful for having had his supervision throughout my degree.

Secondly, I would like to thank Dr. Tatsuaki Okamoto of NTT labs, for his prompt assistance and recommendations on his paper.

Finally, I would like to thank all of the authors mentioned in the references section of this thesis. Completing this thesis would have not been possible without access to their previous work(s).

Abstract

E-cash or electronic cash has been the focus of financial cryptographers since the 1980's. Every year, one or more E-Cash protocols are presented at major cryptography conferences around the world. A goal common to all these schemes is to improve upon the efficiency of their predecessors while offering the same level of anonymity. But how efficient are these protocols?

We aim to analyse this and other trends that E-Cash has seen since its conception. Originally, the analysis method considered for this project was to verify the efficiency and security of the most recent protocol as an aid to our analysis. However, implementing the most recent protocols proves to be extremely time-consuming. Therefore, our new strategy aims to implement an older protocol that has never been implemented before. We shall also use it as a basis for a brief analysis of some more recent schemes. We shall provide an implementation of one of the most influential protocols in divisible, transferable and offline E-Cash's history on which many other protocols are based. We shall also provide the reader with a brief analysis of the performance of other important protocols, including the most recent paper presented at Eurocrypt'05. This allows the reader to draw objective conclusions on the need for new schemes as well as the future of E-Cash.

Contents

List of Figures

1 Background

1.1 Introduction

1.1.1 Definition and History

E-Cash or as more generally known 'Digital Cash/Money'[1] has been extensively studied since around 25 years ago. E-Cash was invented by David Chaum et al [1, 2]. Most E-Cash schemes are built around the core of Chaum's invention of blind signatures. Since its invention, E-Cash has received great attention by cryptographers and a great deal of research has been undertaken in making it more secure and efficient e.g. [12, 4, 5, 6, 7, 8, 9, 10, 11].

The main idea behind E-Cash is to replace hard money by a series of binary form of data. In its simplest form (Figure 1[2]), circulation of E-Cash is as follows [3][3]:

1. A user obtains a wallet with some money from her bank.
2. The user then spends the money with a merchant.
3. The merchant retrieves the money from the bank later.

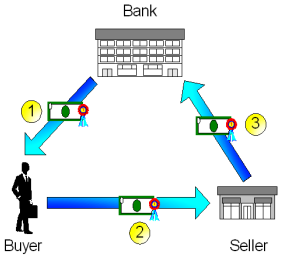

Figure 1: Basic E-Cash Circulation

It can be clearly seen that there are many security threats associated with this model. We shall take a look at cryptographically more advanced protocols later in this paper.

There are two types of E-Cash that have so far been developed: online and offline. In the online scenario, we already have more security as we have a Bank β online who supervises transactions. However, in this paper, we are mainly concerned with the offline scenario of E-Cash[4]. In the offline scenario, we eliminate the bank's supervision by introducing some cryptographic techniques.

[1] From now on, we use the term E-Cash for simplicity of use.
[2] Source: cui.unige.ch/~deriazm/Softs/ECash/images/ECash.PNG
[3] Here, we are referring to the Offline mode of E-Cash.
[4] From now on, by E-Cash we are referring to the offline scenario unless stated otherwise.

The reason for our interest in the offline scenario has arisen from the need for physical cash alike digital money as it offers anonymity and efficiency (e.g. network use, communications lag and etc. . .).

A complete E-Cash protocol must satisfy the following six conditions [12]:

1. *Independence:* As E-Cash is principally in the form of data, its security must not depend on any physical conditions.

2. *Security:* E-Cash is secure. I.e. it cannot be duplicated.

3. *Privacy:* Also known as un-traceability; as with physical cash, it must be anonymous; i.e. the bank cannot reveal the identity of the purchaser or merchant unless one of the two has cheated. In a more secure context, even if the bank and merchant collaborate to reveal the identity of an honest purchaser, this should not be possible, unless the purchaser has cheated.

4. *Offline operation:* Transactions between users and merchants should be completed offline, i.e. the merchant does not need to establish a link to the bank.

5. *Transferability:* Just like real cash, E-Cash can be transferred from one holder to another.

6. *Divisibility (multiple denominations):* Although not exactly physical cash alike, divisibility simulates possession of cash with different face values, i.e. if we have a wallet of value **S**, we can divide the wallet into many pieces such that the total value of the small pieces equals **S**.

The typical players in a transaction are:

1. *A user* $\boldsymbol{U}$: This is a customer who withdraws a wallet from his/her bank.

2. *A bank* β: The bank is responsible for signing the money order and giving it to **U.** β is also responsible for giving the payment of the money to the merchant who has received the money order.

3. *A merchant* $\boldsymbol{M}$: The merchant is essentially the same as the user in the sense that they can both hold digital money. The only difference is that the merchant is the receiver of the money from **U**.

Some protocols introduce an extra entity called the "*Trusted third party*" or "*TTP*". A TTP is essentially a supervisor that can trace all the coins of a cheating user. However, this contradicts with the anonymity principal of E-Cash. To a TTP, the user is known and therefore one must achieve the security

of a TTP without a TTP. This however, does come at the cost of increased space complexity for transactions as a single coin is circulated. This effect is demonstrated in our implementation.

1.1.2 Realisations of E-Cash

Over the past 15 years, there have been many companies offering E-Cash services to customers. Examples of these companies include: DigiCash, Visa cash, Mondex, Interac network (Canada), Octopus card (Hong Kong), Micromint, project CAFE and others. Some of these mentioned offer online cash and some offline without relying on online verification.

So far many of these companies have been unsuccessful (such as DigiCash which declared bankruptcy in 1998 and sold 16 patents and the company to E-Cash systems who is experiencing difficulties too) and some have been so successful that the use of their products have surpassed physical cash transactions in their country of operation (such as the Interac network which surpassed cash as a payment method in 2000)[5].

Although there have been many opinions regarding the reasons for failure of those companies, the issues of trust and security top the polls here[6]. Despite this, many online services such as online casinos and small-value online- deliverable items (such as E-Magazines, E-Books and PDA software) accept E-Cash as a payment method.

Therefore, the need for a provably secure and completely anonymous protocol still exists and there is much research to be done in this field.

1.2 Previous Work

In this section we briefly review the research that has been undertaken on E-Cash. We will also describe the simplest protocol that is the core of most advanced E-Cash protocols. Of course the newer protocols have more mathematically and cryptographically advanced ideas behind them and are much more sophisticated. In order to get a better overview of E-Cash, we will introduce some concepts and algorithms that are used most commonly in E-Cash implementations.

1.2.1 A Simple Protocol

Below is an outline of a simple protocol [3][7] that offers anonymity and security to the user and merchant, unless they have cheated.

[5]Source: http://en.wikipedia.org/wiki/Electronic_money
[6]Source: http://osaka.law.miami.edu/~froomkin/articles/ocean.htm
[7]Here, we are concerned with protocol #4 of the book referred to.

1. The user places **n** money orders in an envelope (achieved by using a blinding protocol [1]). Each money order contains a random long string that guarantees the uniqueness of the money order (i.e. two money orders will not have the same string). On each order, there are also **n** pairs of identity bit-strings that when correctly merged, reveal the identity of the user[8].

2. The user then blinds all of her money orders using a blind signature protocol (this can be seen in the account opening part of our implementation).

3. The money orders are then taken to the bank. In order to maintain anonymity of the user and at the same time verify the user's honesty, the bank asks the user to un-blind **n-1** of the orders at random and to reveal all of their identity strings. Therefore the chance of a cheating user getting away with a fake order is **1/n.** If **n** is large, this chance is close to zero.

4. Provided the user verification is successful and the user's honesty is confirmed, the bank signs the one remaining blinded money order and hands it back to the user. The value of the order is then deducted from the user's account at the bank.

5. The user then spends the money with a merchant by un-blinding it and passing it to the merchant.

6. The merchant then checks the signature of the bank on the order. If it is a genuine signature of the bank, the merchant proceeds to the next step.

7. The merchant then gives the user a random **n**-bit long *selector string* (i.e. series of 0s and 1s). The user then reveals either left hand or the right hand bit of each of her identity strings (not both sides; otherwise the user's anonymity is breached).

8. The merchant then takes the order to the bank. The bank verifies that an order with the same uniqueness string has not been spent previously. If no cheating has occurred, the bank deposits the money to the merchant's account. If either the merchant or the user cheat, the bank will know this by checking whether the identity string of the previously used order is the same as the new order. If it is, then the merchant has copied the order, otherwise the user is guilty. In order to reveal the identity of the cheating user, all the bank has to do is to XOR the two identity strings.

Notes:

[8] This can be achieved using secret splitting and bit commitment protocols.

1. The protocol introduced does not satisfy the 6^{th} condition of a complete protocol [12] (however, we can modify the protocol so that we can achieve this). We shall introduce protocols that claim to meet this condition along with the others, later in this paper.

2. Also the reader should note that the efficiency of the protocol above is not feasible. In fact the data requirement for a single purchase would be around 200megabytes[9].

3. In the protocol above, if double spending occurs, it is only discovered when the merchant has taken the money order to the bank. However, we are not concerned about this as the same issue exists with physical cash.

4. Just like real cash, if someone copies and spends the user's money before she does, the user will be identified as guilty.

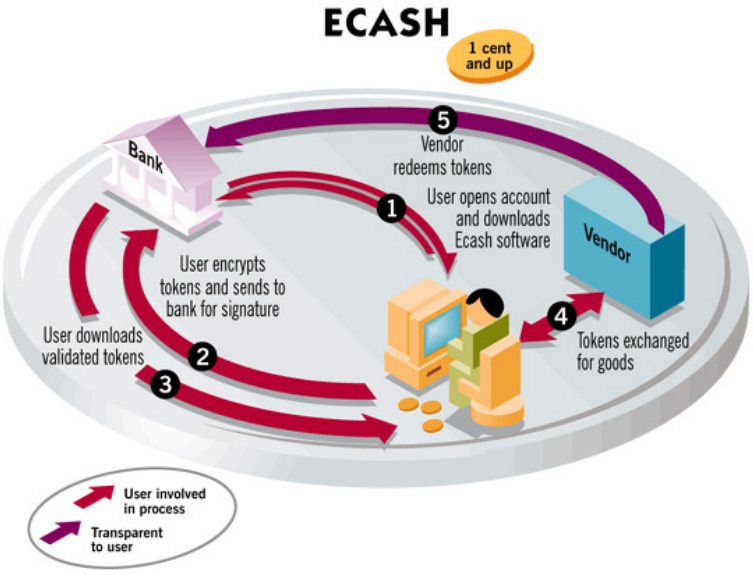

Figure 2: A More Advanced Exchange Protocol

Figure 2[10] shows the presentation of one such scheme . The protocol above does satisfy the conditions for a complete E-Cash protocol (divisibility can be achieved using this protocol but some changes are required in order to do this. Our implementation of Okamoto's scheme in this paper shows how). However we need to do better by improving the efficiency of the protocol. As a result, we will be looking at some other protocols that have been regarded as complete and efficient, and we shall take a more in depth look at one such protocol and

[9]Reference [3], Page 147.
[10]Source: elab.vanderbilt.edu/research/studentprojects/secure.payment.systems/ecash.jpg

analyse it by implementation. We shall present mathematical ideas behind our selected scheme in sub-section 1.4 of this paper, as before one moves to the internals of protocols, obtaining a general overview of each is necessary.

1.2.2 More Advanced Protocols

A quick review of the submissions between 1989 and 2005 reveals a set of most referred-to E-Cash schemes, e.g. [3, 9, 11, 10, 13, 4, and etc...]. As we are concerned with efficiency, we will review the three most relevant submissions: [12, 14, and 15].

In [12], Okamoto claims that his scheme is the first complete and efficient protocol that satisfies all six requirements of E-Cash[11]. He also shows that the security of his scheme relies on the difficulty of factoring[12]. He also demonstrates that divisibility is feasible by introducing *"Money Structures"*. These structures are principally the same as a binary tree with some usage rules. He achieves divisibility by using two identical money structures called Γ and Λ tables respectively. Okamoto then moves on to introduce two protocols of which the second is transferable. Double expenditure is prevented by the idea of "disposable Zero-Knowledge authentication protocol" which was invented by Okamoto himself [8].

Efficiency: In his paper, Okamoto claims that his protocol takes on average 20kilobytes of data transmission to complete a transaction. Moreover, he estimates the average speed of his protocol (assuming a Rabin scheme chip of 30kbps is present) to be several seconds. This will be evaluated in the performance section (2.2.2) of this paper.

The author also confirms in his submission, that his scheme is not unconditionally un-traceable.

In [15], Ferguson claims that his scheme is more efficient than the others previously known[13]. He claims that his *withdrawal protocol* (the part in which users withdraw wallets from their banks) is the first that takes advantage of a direct construction and does not rely on the *cut-and-choose* algorithm[14]. Ferguson also claims that the efficiency of his scheme is due to a single term payment challenge rather than many request-replies between users and merchants. Another efficiency improvement in his scheme is due to eliminating the cut-and-choose use inspired by S. Brands' work based on the representation problem [5].

As it can be seen in his paper, his withdrawal protocol does depend on the

[11] Note that the author of the paper himself has introduced the conditions; hence one might view his claims as subjective.
[12] We shall evaluate these claims later in this paper.
[13] Claim not evaluated for after the year of publication (1993).
[14] For C-A-C refer to [3] p.103

use of RSA and *randomised blind signatures*[15] introduced by Chaum.

Efficiency: Although there is no information by the author about the efficiency of his protocol, he claims that some changes to a variable in his system could reduce the storage size of a coin to 70bytes.

In [14], Brands introduces a primitive he calls "*restrictive blind signatures*" in order to achieve un-traceability. He claims that this, in conjunction with the representation problem in groups of prime order, achieves a highly efficient offline system which does not affect the privacy of the user.

The security of Brands' scheme is derived from the security of "*Schnorr identification scheme*" [16] in which the presence of a smart card is a requirement.

In his paper, the author introduces two schemes of which the second is an extended version of the first to achieve prior restraints of double spenders.

1.3 Our Contribution

As we saw in 1.2.2, there have been many schemes designed that are claimed to be efficient. However, many of these claims are hidden behind complex mathematical ideas which do not reveal much about protocol efficiencies. Therefore, there exists a need for an implementation of these schemes in order for one to analyse their efficiency or even verify their security.

As reviewed in 1.2.2, there have been three schemes most talked about which claim they represent an efficient E-Cash scheme.

In the next parts of this paper, we shall implement one of these schemes most relevant to our aims and interests. We shall take a look at Okamoto-Ohta's E-Cash protocol and implement it in a high level language[16].

Notes: The reason for eliminating Brands' scheme in this paper is that he relies on a physical property (i.e. tamper proof smart cards) in order to achieve security. In our point of view, the use of tamper resistant equipment takes away the first condition of a complete E-Cash scheme. Furthermore, this has sometimes been viewed as an online scheme, as an observer is present during transactions. Clearly, this is not of interest to us. However, Brand's scheme does introduce some valuable ideas for use in the field of financial cryptography. Another reason for not implementing Brands' scheme is that a security flaw (possibility of identity cheat) in the setup stage of the protocol has been identified [17].

[15] He presented a version of this in his paper.
[16] We will use Java for this purpose.

1.4 Preliminary Preparations

Realisations of E-Cash protocols rely heavily on mathematics, in particular number theory and group theory. Therefore some basic knowledge of group theoretical and number theoretical ideas is expected of the reader. However, one might need to refer to this section as a guide in order to understand the meaning of the symbols used. Much effort has been put into preserving the original notations used by the authors of the scheme implemented. This allows easy comparison of this document with the original.

Readers can refer to the usage of these terms in [12] for a more detailed explanation

We shall be using the following mathematical notations:

- **Blum integers:** N is called a Blum integer if: $N = PQ$, where P and Q are primes with $P = 3(\text{mod}4)$ and $Q = 3(\text{mod}4)$.
- **Williams integers**: A special case of Blum integers with: $P = 3(\text{mod}8)$ and $Q = 7(\text{mod}8)$. Williams integers have all the properties of Blum integers.
- **(X/N)**: This denotes the Jacobi symbol when N is a composite number with $N = PQ$, and denotes the Legendre symbol when N is a prime. When N is composite, QR_N and QNR_N classify the extended quadratic residue properties of N and x.
- $\mathbf{QR_N}$: $Z_{(1,1)} = \left\{x \in \mathbb{Z}_N^* | (x/P) = 1, (x/Q) = 1\right\}$
- $\mathbf{QNR_N}$: For $a, b \in \{-1, 1\}$, a & b not both 1:
  $Z_{(a,b)} = \left\{x \in \mathbb{Z}_N^* | (x/P) = a, (x/Q) = b\right\}$
- $[\mathbf{x^{1/2} mod N}]_{\mathbf{QR}} = \mathbf{y}$: y is the quadratic residue of square root of x mod N such that $y^2 = x(\text{mod}N)$. This concerns both 1 and -1 as quadratic residues. The following two concern +1 and -1 QRs of square root mod N of x.
- $[\mathbf{x^{1/2} mod N}]_{\mathbf{1}} = \mathbf{y'}$: Such that $(y'/N) = 1$ and $0 < y'' < N/2$.
- $[\mathbf{x^{1/2} mod N}]_{\mathbf{-1}} = \mathbf{y''}$: Such that $(y''/N) = -1$.
- $< \mathbf{z} >_{\mathbf{QR}}$: For N (Williams integer) and $z \in \mathbb{Z}_n$, $(< z >_{QR} = dz(\text{mod}N))$ such that $d' \in \{\pm 1, \pm 2\}$ and $dz \bmod N \in QR_N$.
- $< \mathbf{z} >_{\mathbf{1}}$: $< z >_1 = d'z \bmod N$ such that $d' \in \{1, 2\}$ and $(d'z/N) = 1$.
- $< \mathbf{z} >_{\mathbf{-1}}$: $< z >_{-1} = d''z \bmod N$ such that $d'' \in \{1, 2\}$ and $(d''z/N) = -1$.

When we talk about the formulae in square brackets, we imply that we are interested in one out of four of the square roots of $x(\mathrm{mod}N)$. It is expected of the reader to be able to follow these conventions in order to appreciate the security and proof of security of the system implemented. Usages for all these functions are illustrated in our implementation of the system. All the above formulae can be calculated in an efficient manner.

2 Analysis

In order to understand Okamoto's E-Cash scheme better, firstly we need to analyse the protocol in a theoretical manner; secondly, we need to execute systems analysis from a programming/design point of view.

2.1 Theoretical Design

As mentioned before, Okamoto was the first to design an electronic cash scheme that was truly offline, i.e. his protocol did not depend on any physical properties such as a tamper proof smart card (although one can be used to store data) or a third party verifier. He also introduced the first divisible digital cash protocol, which meant a single electronic licence (bit-string containing the digital cash) could be used in parts or even transferred to another person. Therefore, his E-Cash scheme is perhaps the most important one built to date. Most of the protocols built today use this and an improved version of it to achieve less time and space complex implementations.

Perhaps the most important novel idea introduced in his paper was the use of a version of binary trees which he calls "Hierarchical Structure Tables".

2.1.1 Hierarchical Structure Tables

A hierarchical structure table is simply a binary tree which has all its left sub-trees mirrored into its right sub-trees. It allows an electronic licence C to be subdivided into many smaller pieces so that it can be re-used.

The tree is structured as follows:

1. Its root node has the total value of the electronic licence stored.
2. Each child, whether left or right, has $1/2$ of its parent's value. So for example, starting from a licence of \$100, the first children hold \$50 and the children of the children hold \$25. We cannot divide this any further by 2, so this will form our tree.
3. When a user pays by a node (the corresponding value in a node), its children become unusable.

4. When a user pays by a node, its parents become unusable.

5. No node is allowed to be used more than once.

The structure above has some implications. Firstly when we use a node, we are disabling all its children and all its parents; however, this does not mean that the children of a disabled node are always un-usable, as shown in the figure below. Secondly, as mentioned in structure notes (part 5), a disabled node cannot be re-used by its owner. This means that any tampering of the structure table is traceable. We shall denote this structure table by Γ.

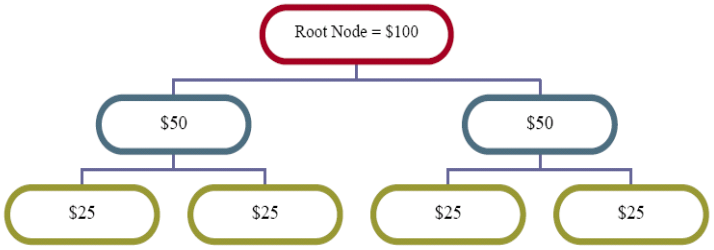

Figure 3: Hierarchical Structure Table

In order to realise the first four restrictions mentioned above, we use table Γ. However, we require another table which we denote by Λ. The second table is used to realise the fifth restriction, i.e. preventing double expenditure. Moreover, Γ and Λ have exactly the same structure and follow a common node and level naming convention as described below.

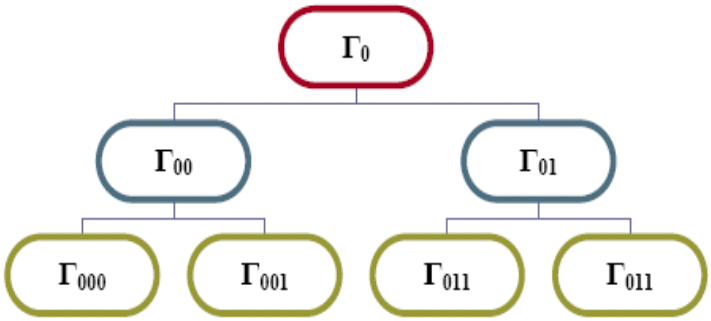

Figure 4: Γ Table

As it can be seen, at each level of the table we start by placing as many zeros as the number of level the nodes are in (at ith level we have 2^i nodes). Then from left to right we number the nodes in binary. Having a look at the protocol on the whole, this naming convention is only in place to identify each node uniquely. It also eases the identification of parent and children nodes when restructuring

tables. However a recursive algorithm to do this could be implemented; but this means that we would be trading time for space, so to find a certain node, we would have to run a recursive search.

Note: The reader should appreciate that at implementation level, we are interested in following the original instructions as closely as possible. In any case, we shall analyse (in 2.2.2) whether using two tables is best in terms of space efficiency when designing this protocol.

2.1.2 Protocol 1 (Basic Divisible Universal Electronic Cash)

Now that we have reviewed the foundations of this scheme, we proceed to describe in brief, the internals of the first protocol (that satisfies all the properties of E-Cash mentioned in section 1.1.1 apart from the transferability property).

Note: Due to lack of space and to avoid repetition of descriptions from the original paper, the reader is recommended to obtain a full description of [12]. However, this and the practical design sections serve as supplements to the original paper. We clarify some algorithms, methods (including a security analysis in 2.1.2) that are illusive to the reader when reviewing the paper. We have also included 2.1.4 which is a result of our theoretical analysis on the efficiency of this protocol. The results from this section will be used later to make comparisons with other E-Cash protocols.

- **Part 0: Preparations**

  Bank A who issues accounts and pays out merchants, publishes some RSA public keys: one for signing electronic licenses and one for each value of electronic cash that it supports. We denote these signing key pairs by (e_A, n_A, d_A) and the rest of the keys by $(e'_A, n'_A, d'_A), (e''_A, n''_A, d''_A)$ and so on.

  In order to fulfil blind-signatures' purpose and to prevent double spending, A publishes f_Γ, f_Λ and f_Ω which are randomised hash functions ([18]) for payments using hierarchical structure tables (note that these functions do not have to be one-way, but we shall use *SHA-256* as it is already provided and is very efficient for hashing). The customer has a public and private key pair denoted by (e_P, n_P, d_P). The security parameter[17] here is clearly $\mathrm{K} = |n_A| = |n'_A| = \ldots$

- **Part I: Opening an Account**

  Suppose a customer P intends to open an account with a bank A. P obtains an electronic license L from the bank. To obtain the licence, the following protocol is executed:

[17] Security parameter is the bit-length of banks' modulus.

1. P generates K Williams integers $N = PQ$ where P, Q are prime Williams integers. He also generates K random values a_i for as blinding factors.

2. P generates and sends K blind candidates to the bank. The blind candidates W_i are calculated as follows: (for i = 0 to i = K-1) $W_i = (r_i^{e_A} \times g(I_i||N_i)) \bmod n_A$ where r is a random integer used as a blinding factor, $g(...)$ is a one-way hash function, $I_i = I_{1,i} \,||\, I_{2,i}$, $I_{1,i} = S_{1,i}^2 (\bmod N_i)$, $I_{2,i} = S_{2,i}^2 (\mathrm{mod} N_i)$, $Si = ID_p \,||a_i||\, (g(ID_p||a_i))^{d_p} (\mathrm{mod} n_p) = S_{1,i} \,||\, S_{2,i}$, ID_p = Unique ID and $||$ denotes concatenation of two bit-strings.

3. A chooses a K/2 random subset of the blind candidates received from P and sends their corresponding i back to the customer.

4. P reveals $a_i, P_i, Q_i, (g(ID_p||a_i))^{d_p} \bmod n_p, ID_p, r_i$ for each requested i in 3. A forms blind-candidates from these and compares them with the ones received in 2. If they are invalid, A refuses to issue a licence to P. This step reduces the probability of ID faking by customers. The probability of a customer getting away with fake details is $1 - \frac{(numberoffakedetails)}{(Numberofblindcandidates)}$.

5. A encrypts the product of all remaining unchecked blind-candidates using his signature RSA private key and public modulus and sends it to P. So: $(\prod W_i)^{d_A} \bmod n_A \rightarrow P$.

6. P extracts his licence from 5. So: $L = (\prod g(I_i||N_i))^{d_A} \bmod n_A$.

Notes: Licence extraction does not depend on the knowledge of d_A as $\prod r_i^{e_A^{dA}} \bmod n_A = (\prod r_i \bmod n_A) \bmod n_A$ so L can be extracted from 5 in polynomial time.

- **Part II: Buying E-Cash**

  Suppose P needs \$d of E-Cash. When P asks the bank for \$d, the bank selects d's corresponding RSA keys $(e^{\cdots'}, d^{\cdots'}, n^{\cdots'})$ and sends it to P. If the bank does not have a key for that value, P selects another value.

  1. P chooses a blinding value b and a random blinding integer factor $r \in \mathbb{Z}_{n'_A}$ and sends $Z = r^{e'_A} g(L||b) \bmod n'_A$ to$A$.

  2. A encrypts Z with (d'_A, n'_A) and charges the customer's account \$d.

  3. The customer's E-Cash can now be extracted from Z by the same method as described in the notes of Part I. We shall denote the electronic bill by C.

- **Part III: Paying by E-Cash**

  Suppose P is to pay a merchant V for an article. The payment protocol executed between the two is as follows:

  1. P generates the root node of his Γ table for each of his K/2 blind candidates. So $\Gamma_{i,0} =< f_\Gamma(C||0||N_i) >_{QR}$.
  2. P also determines the nodes of his Γ tree that he needs to pay with. For example if he has \$100 and wants to pay \$75, then he chooses the nodes corresponding to (\$50, \$25) or (\$25, \$25, 25). We denote the chosen nodes by $\Gamma_{j_1 \ldots j_t}$ where $j_i \in \{0, 1\}$ are the node numbers explained in 2.1.1. The same applies to the nodes of the Λ table.
  3. Having chosen the right nodes, P computes:
     $X_{i,j_1 \ldots j_t} = [(\Omega_{i,j_1 \ldots j_{t-1}}^{2^{t-1} j_t} \times \Omega_{i,j_1 \ldots j_{t-2}}^{2^{t-2} j_{t-1}} \times \ldots \times \Omega_{i,j_1 \ldots j_{t-1}}^{2 j_2} \times \Gamma_{i,0})^{1/2^t} \bmod N_i]_{-1}$
     where $\Omega_{i,j_1 \ldots j_t} =< f_\Omega(C||j_1||\ldots||j_t||N_i) >_1$.
  4. Having computed 3, P transmits to V: $(I_i, N_i, X_{i,j_1 \ldots j_t})$ for each i selected by the bank when opening the account. (L, C) are also sent.
  5. Having received the data in 4 from P, V proceeds to verify their validity,. This can be efficiently done as V can re-compute the data without knowing the customer's private key. Matching the computed data with the received ones is simple. Clearly the Jacobi symbol of X in (3) over N equates to -1 if no faking has been committed. Otherwise another quadratic residue would be output (i.e. $(X_{i,j_1,\ldots,j_t}/N_i) = -1$). Also computing $(X^{1/2} \bmod N)^{-1} \bmod N$ is clearly feasible in polynomial time as $X_{i,j_1 \ldots j_t}^{2^t} = d_i \Omega_{i,j_1 j_{t-2}}^{2^{t-1} j_t} \ldots \Omega_{i,j_1}^{2j} \times f_\Gamma(C||0||N_i) \bmod N_i$ for $d_i \in \{\pm 1, \pm 2\}$. An invalid result will halt the transaction.
  6. For each i, V selects a random bit E and sends it to P. The reason for this is to enforce random alteration between positive and negative quadratic residues in (7) as to prevent alteration of the Λ table.[18]
  7. P computes the following quadratic residues according to the random bits received in 6: $Y_{i,j_i \ldots j_t} = \left[\Lambda_{i,j_1 \ldots j_t}^{1/2} \bmod N_i\right]_{(-1)^{E_i}}$ where $\Lambda_{i,j_1 \ldots j_t} = \langle f_\Lambda(C||j_1||\ldots||j_t||N_i)\rangle_{QR}$.
  8. Now the merchant verifies the Y values by the same method of quadratic residue checking. So: $(Y_{i,j_1 \ldots j_t}/N_i) = (-1)^{E_i}$ and $Y_{i,j_1 \ldots j_t}^2 = d_i' f_\Lambda(C||j_1||\ldots||j_t||N_i) \bmod N_i$, where $d_i' \in \{\pm 1, \pm 2\}$.

  If verification is successful, the payment is accepted, otherwise the transaction is terminated.

[18] This interaction is called "random challenging".

Notes: One might ask how it would be possible to prove the execution of such transaction when V asks for credit from the bank. This is feasible as long as a time stamping method and an ID exchange method is employed. One could simply use a one-way hash function (such as MD5 or SHA-x) to hash ID_v, time T and E_i, and replace the results with E_i. As the results can still be odd or even, V can still randomly challenge P.

- **Part IV: Cashing E-Cash**

  Cashing E-Cash is a simple matter of obtaining the history of part III and checking against the known S_i from part I. If $(1/2)+1$ pieces of S_i are detected, the identity of the customer is then trivially revealed (using bit commitment). H (i.e. customer-merchant transaction history) must also be stored by the bank so that revealing double spenders' IDs is possible.

- **Security proof of protocol 1**

  As the use of quadratic residues above may not be clear to the reader, we shall provide a brief justification.

  Suppose a customer P is not being honest and uses a payment node twice. Even if P gets through the payment process, the bank can use the positive and negative quadratic residues to factorise N from $\left[x^{1/2} \bmod N\right]_{\pm 1}$ (remember that N is a Williams integer, therefore this is feasible in polynomial time.) obtained from the payment history. This is also why merchants challenge customers randomly (by sending random Es) in part III.

  Now what if a customer violates restrictions 3 and 4 of the money structure tables? Suppose P pays with a node and its child which violates the restrictions. If a node $\Gamma_{i,j_1,\ldots j_i}$ is used, then P calculates $X_{i,j_1,\ldots j_i} = [\Gamma^{1/2}_{i,j_1,\ldots j_i} \mathrm{mod} N_i]_{-1}$ $(\forall i)$ and sends them to the merchant (and to the bank via the merchant). Now suppose P uses a child of the node mentioned, say node $\Gamma_{i,j_1,\ldots j_i j_{i+1}}$. Then P sends back to V and therefore to A: $X_{i,j_1,\ldots j_i j_{i+1}} = \left[\Gamma^{1/2}_{i,j_1,\ldots j_i j_{i+1}} \bmod N_i\right]_{-1}$. But as $[\Gamma^{1/2}_{i,j_1,\ldots j_i} \bmod N_i]_{+1} = X^2_{i,j_1,\ldots j_i j_{i+1}} (\bmod N_i)$, by the quadratic residues, the bank can factorise N. From the challenge-responses in part I the identity of P is then revealed.

  Finally, we justify the use of Ω values. This value is used to conceal the identity of users from merchants. Suppose the payment algorithm was executed with a constant value k instead of the variable Ω; therefore the relationship between a child and its parent node would become $\Gamma_{j_1\ldots j_i j_{i+1}} = k(\Gamma_{j_1\ldots j_i})^{1/2}$. As a result of this, when P uses two adjacent nodes, he opens $X_{j_1\ldots j_x 0} = (\Gamma_{j_1\ldots j_x})^{1/4}$ and $X_{j_1\ldots j_x 1} = (k(\Gamma_{j_1\ldots j_x})^{1/2})^{1/2} = \sqrt{k}\sqrt[4]{(\Gamma_{j_1\ldots j_x})}$ where the Jacobi symbols of the two nodes are equal and are in $\{\pm 1\}$.

It is clear that the merchant could calculate $\sqrt{K} = \left(X_{j_1 \ldots j_x 1}/X_{j_1 \ldots j_x 0}\right)$ and compute the factors of N (and therefore the identity of the customer) by this; soon after, the customer would have the pleasure of receiving thousands of marketing letters by post everyday (in line with many other consequences).

2.1.3 Protocol 2 (Transferable Universal Electronic Cash)

Having studied protocol one and analysed its security, we move on to the second scheme which is obtained by applying a slight modification to the first scheme. All we have left to do now is to satisfy the fifth criterion of a complete offline E-Cash scheme (i.e. transferability). As a prerequisite to this protocol, the reader is expected to have appreciated our previous work carried out in 2.1.2.

- **Part I: Opening an Account**

  Suppose Alice and Bob open accounts at bank A. They go through exactly the same steps as described in Part I of 2.1.2. Let us denote their licences by L_{Al} and L_{Bob} respectively.

- **Part II: Buying E-Cash**

  Suppose Alice wants to buy \$100 worth of E-Cash. She could use part II of 2.1.2 to achieve this. Let us denote her E-Cash by C.

- **Part III: Transferring E-Cash**

  Step 1: Suppose Alice owes Bob \$25 and she wants to pay him by E-Cash. All Alice has to do is to think of Bob as the merchant described in 2.1.2 and pay him \$25. So she does.

  Step 2: Alice also sends a transfer certificate T to Bob that says she has paid Bob \$25. T is essentially a digital signature that has C, the payment node string j and the licence of Bob encrypted: $T = (\langle C||j||L_{Bob}\rangle)^{1/2} \bmod N_{j_{Al}}$

- **Part IV: Spending a Transferred E-Cash**

  Now that Bob has some E-Cash in his wallet, he celebrates this by spending it at a merchant V.

  Step 1: Bob has to tell the merchant that this money has been transferred to him from someone else. V requests a history of the transfer. The transfer history is made up of the certificate and the history of payment from 2.1.2. V then verifies the history; if it is genuine, V and Bob proceed to step 2.

  Step 2: Bob can now use the payment part of protocol 1 to pay V.

- **Part V: Cashing Transferred E-Cash**

  Now that V has received his money, he proceeds to cash it at bank A. V presents the history (i.e. all the steps involved in the payment process) to the bank. The bank then verifies this history and verifies V's account provided no cheating has occurred. Otherwise the bank reveals Bob's identity or penalises V for cheating. Having credited V's account, the bank stores the submitted history as evidence of payment.

2.1.4 Theoretical Efficiency

Now that we have acquired an understanding of the internals and the security of this scheme, we proceed to analyse its efficiency.

According to Okamoto and Ohta [12], their protocol is efficient and should run reasonably fast assuming a Rabin scheme chip is available for encryptions and decryptions. However, there is an obvious pitfall. The total amount of data transferred for one payment increases rapidly when a piece of digital cash is spent in multiple denominations. As transferring E-Cash is a similar process to spending it, the same avalanche effect applies to it. To visualise this fundamental issue, assume the total amount of data transferred between two parties when making a single payment is c kilobytes and involves the whole a user's wallet. Now suppose the same user wants to use his wallet to pay with m pieces of smaller coins; then as it can be seen in the payment protocol, the transferred data amounts to $m \times c$ kilobytes. Now suppose the same user transfers his E-Cash to another user. Then the total amount of data transfer after n transfers becomes$(n + 1) \times c$. However, it was shown by Chaum and Pederson that the amount of data per payment needs to grow, at its best in polynomial space of its input (i.e. number of spending and transfers); otherwise banks would not be able to identify cheating users. Therefore, it is reasonable to assume for the second case (i.e. transfers) that we are not breaking the norm. However, if this E-Cash was to circulate many times, then the amount of storage required to keep the history of transactions would be dominated by $n \times c$ and it could potentially take many megabytes.

Suppose that Alice is using this scheme to withdraw cash and make payments. Suppose Alice has \$32 and she wants to spend her money in multiple denominations of as small as \$1. Also suppose that the security parameter used is k = 1024 bits (a realistic choice) . The total amount of data transfers according to the protocol's description is shown below:

1. *Opening an account:* Alice chooses k random values of k/2 bits, 2k number of 1024-bit Williams primes and their multiples. Total storage for Alice $= 2k^2 + k^2/2 = 2621440$ bits = 320Kbytes. Her blind candidates cost

her: $k^2 = 128$Kb as each of which is calculated (mod n) where we ignore any intermediate calculations. She also has to store her ID (1Kb, so it is insignificant) and I_i (she could recalculate this every time she needs it, but this will have a large time overhead). At this stage, Alice stores $k+k^2bits = 128.125\text{kb} \simeq 128\text{kb}$. Alice's license would be about k bits as it is calculated (mod n). Hence the total storage needed for Alice's account data alone is about 577Kbytes.

2. *Buying E-Cash:* Assuming we can ignore intermediate storage needed when E-Cash is being issued, Alice's money order would need k bits of storage again (as calculations are done in mod 1024). Hence, Alice's total storage so far is: 577 + 1 = 578 Kbytes.

3. *Payment of E-Cash:* Assume Alice wants to pay \$18 to a shop for an article. As we have seen, Alice needs to initialise the root node of her Γ table k times. We assume that the randomised hash function used here outputs 256 bits (e.g. SHA-256). So the storage needed is $k\times$ (256 mod 512) = 256k = 32Kbytes. Alice determines which nodes she wants to use. In this case she uses nodes Γ_{00000} and Γ_{01} as it can be seen in Figure 5. She calculates the X challenges for each payment node. As Each X challenge is a (-1) quadratic residue between 0 and $N/2$, in the worst case we have $(N/2)_b$ bits to store for each X. Note that we have k of these challenges; so we need $k \times 1023 \times 2 \simeq 256$Kbytes. We have excluded the calculation of Ωs' (identity blinding) space complexity as the former results do not depend on this value. In general cases, if Alice was to select n nodes, she would have to store $k \times 1023 \times n \simeq 128n$Kbytes. So if $n = 10$ then she had to store a little above 1Mbytes of data. From this point onward, all of Alice's calculations are intermediate; therefore there is no need for her to store these data values. By the end of payment, Alice has stored 32 + 256 = 288 Kbytes of data which she will need in the future. Generalising this, if Alice has n payment nodes to pay with, she would have to store $128n$ + 32 = 32 (4n+1) Kbytes. Hence, the space complexity here is of order constant and grows in proportion with the number of payment nodes that Alice chooses for payment.

Introducing protocol 2 as an extension to protocol 1, we can also calculate the amount of data required for Bob who receives c from Alice. Suppose Alice is to transfer the rest of her money to Bob and she has already used \$18. By part III of protocol 2, Alice selects \$14 and transfers it to Bob along with a transfer certificate. Clearly the only legal combination of nodes for Alice is to choose nodes Γ_{001}, Γ_{0001} and Γ_{00001}. By (3), Bob will have to store 32(12 + 1)

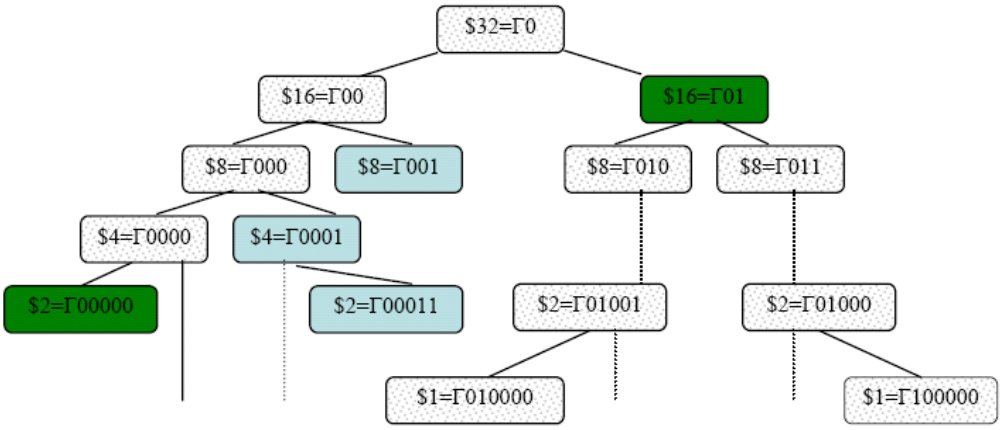

Figure 5: The State of Γ Table after Spending $18

= 416Kbytes for the payment and a certificate which is 1024 bits in worst cases (calculating mod 1024). Alice's payment history is also transferred to Bob so that the bank could verify the E-Cash. So for Bob to have $14 from Alice, 417 + 578 + 32 + 256 = 1283Kbytes of storage is needed. In general cases, after m transfers (assuming no money is spent between each transfer), our storage needs become $mk + 32(4n+1) + |H|$ (where |H| is the size of payments history). This result would grow at a higher rate with intermediate expenditures included between transfers because of n's increase (as we need more nodes to select to make a transfer in this case). In the end we calculate the total storage needed in the scenario explained above; collectively, Alice has to store: 578 + 32 + 256 = 866Kilobytes.

2.2 Practical design

Having acquired an idea of the theoretical issues surrounding Okamoto's scheme, we proceed to practically analyse it.

For the best part of the last 25 years, new E-Cash schemes have been introduced. Although their space efficiency or time efficiency has improved year upon year, so far we have no idea of how practical they will ever be.

In this part we address this exact issue and utilise our results to make comparisons with other famous schemes. The reader should note that even though we are concerned with the security of this protocol, we shall not go into so much detail about identification of cheating users. This however does not mean that we do not need to include security features in our implementations, but merely to concentrate on the tasks that would consume most resources (computation and memory). As a result, our protocol is as secure as its specification. If a

user cheats, it is possible to obtain his ID by a simple calculation (we showed this in our security proof section).

Our implementation is in Java™ using JDK 1.5 and no outside libraries (apart from a previously created menu system); this ensures (a) our code is a good representation of the scheme and (b) efficiency lacks in our program (both in terms of computations and memory usage) are minimised. We have included segments of our code in Appendix 6. A brief user manual is also submitted separately accompanying our code.

2.2.1 Protocols 1 and 2 (Divisible Universal Electronic Cash)

The strategy adopted for program design is as follows:

1. Problem identification, background learning and familiarisation with the mathematical ideas behind the problem.
2. Theoretical problem analysis to obtain an overall view on the best implementation strategy.
3. Coding each step of the program as part of methods and/or classes.
4. Bringing everything together by 3 main packages.

A simple design strategy would consist of designing three classes that communicate together by obtaining an instance of each other. However as this is unlikely to be true in real life situations (e.g. communication lags), it was decided to use sockets over networks as our class communication method.

Our main classes are (i) *Bank*, (ii) *Customer* and (iii) *Merchant*. Classes are designed to have protected fields and methods so that no other class outside their domain could read their important private data (such as their secret keys). There are also supporting classes for each class which we list here:

1. *Bank*:
   - A main class that initialises bank menus.
   - A bank class that manages bank servers and other bank side instructions.
   - A class that holds cryptographic and verification methods.
2. *Customer*:
   - A main class that initialises customer menus.
   - A class that communicates with banks to get bank details (such as name, address, public exponent and etc. . .).

- A cryptographic library which is responsible for mathematical computations in the protocol.
- A hierarchical structure table class (HST) that oversees creation, storage, wallet rebuilding, payment node picking and other processes for user wallets.

3. *Merchant*:
   - A server initialisation class.
   - A server class that handles requests from customers and executes the payment receiving protocol.
   - A verifier class which holds cryptographic and mathematical computations methods needed for merchants' side of protocol.

Extra supporting classes have been created or imported to ensure a friendly and accessible interface. These include:

- PC_Menu[19]: A library imported for easy menu creation.
- Vdu: This library was created to display client/server outputs on the screen, aids users by passing information such as tasks' progress and other extra functionality such as output saving and/or printing.

Now that we have briefly covered our main classes and their methods, we shall cover issues that arise during implementation.

1. **The HST** Okamoto introduced the idea of using hierarchical structure tables as a wallet representation (Figure 3, Figure 4). At first look, one might imagine that a fast and efficient way to implement a wallet would involve binary trees. However, storing a binary tree of l dollars with the smallest coin being δ dollars has $O(\log n)$ complexity, i.e. we have $\log(l/\delta)$ levels in our tree. Even worse, we need $2^{(\log(l/\delta)+1)} - 1$ nodes to store our wallet. Alternatively one could add new nodes as they are needed. This however would introduce resource unfriendly recompilations of user wallets each time they are used. This is even worse when realising the restrictions and rules associated with HSTs' specifications. In the light of the analysis above, it was considered best to deploy user wallets using a slightly modified (in terms of usage) version of the binary tree data structure (see HST.java). This modified version stores three data values in each node; a Boolean that indicates whether a node is usable, an integer that represents the value of coins in each node and a binary number associated with each

[19] Created by Mr. Adrian Lane, Park College, Eastbourne, UK

node (this makes searching for a node much faster; in fact, we can avoid searching a wallet entirely each time we look for a node, its parents and its children). As a result, this model would represent wallets in a more compact and structured manner. Another benefit of using this version is that the rules for HST usage can be realised more efficiently. For example if a coin of value \$32 (binary path Γ_{010}) is used from a \$128 note (Γ_0), our wallet can be normalised by simply flagging all nodes in between the top node and the payment node to "used = true" (i.e. starting from node Γ_0, go to right child Γ_{01} go to left child Γ_{010} marking them "used" on the way). This leaves us with rendering children of the used node invalid; this can be done conveniently by dropping the immediate children of Γ_{010} (i.e. setting them to null). Therefore we store 31 coins less in our wallet; hence, our wallet becomes more compact. Clearly the higher the value of a coin used or the more payments made, the more compact our HST becomes. It is worth mentioning that our strategy to store a wallet was to use a minimum number of symbols so that we acquire an even more compact wallet size.

2. **The Payment Protocol** The payment protocol could be implemented using different methods which result in different execution speeds. As we are trying to follow Okamoto's scheme as closely as possible, we describe two methods here, (1) to compute each X value in the protocol at once and (2), to compute each X and Ω value for each payment node and send them to merchant for verification. Clearly, the first requires a user to compute all values before submission to the merchant. This would be a poor strategy as the merchant is idle when the customer is computing these values. The second method reduces transaction times. When each individual value of X relating to a payment node is being sent to the merchant, the customer can start computing the rest of the values, while the merchant carries out verifications. However, the first strategy proves to be beneficial when customers know how much they will be paying, i.e. the customer can pre-compute these values and send them to the merchant when paying.

3. **Bank Initialisation and Multiple Denominations** As mentioned previously, Okamoto's scheme can handle multiple denominations. However, the scheme requires a bank to have an RSA key pair for each denomination. This does not pose any major issues as these values can be pre-computed rather than computed as needed. As it can be seen in 2.2.2, for current strength RSA keys, key pair generation is relatively computationally expensive. But as we are merely demonstrating the capabilities of the

scheme in this context, pre-computing a set of all key pairs is essential. Alternatively, one could reduce the number of supported E-Cash values by specifying this when creating a new bank (in our program this is accessible by going to "Create a Bank" menu and entering a larger number for minimum denominations).

4. **UI and Securing Data** This part is purely design related; therefore we shall not go into too much detail. As for the user interface, it was decided to design a simple UI to make the program more accessible. This would also allow better visualisation of input/output of our code. Program menus adhere to a simple structure. Windows are also introduced to display the outputs of each entity (i.e. bank, customer, merchant) in our system. These VDUs allow an observer to see the progress of each task and save outputs to files or print them directly for convenience (e.g. when making comparisons). As for the program level security, each method has been assigned an access level by setting privacy settings for them[20]. Therefore, each customer, bank and merchant's private data are invisible to the others.

5. **Other Implementation Issues** The rest of the implementation have been realised mirroring exactly the original protocol descriptions. Our code has been well commented with references to each step of each part of the protocol at hand[21].

Note: We have introduced time logging and memory logging in our implementation to aid analysis (we will use these logs to evaluate efficiency in 2.2.2). The results are immediately available after each part of the program is executed.

2.2.2 Practical Efficiency

In this section, we shall analyse our realisation of Okamoto-Ohta's scheme. The reader should note that results in this section are machine dependant; they have been obtained on an AMD Athlon^{TM} 2000+ with 512 MB of Ram hosting Suse Linux 10.0. Although machine dependent, the results can represent an acceptable estimate of efficiency of the scheme. The reader should also note that prime number generation is an expensive operation requiring many intermediate calculations for primality testing purposes (although primality testing nowadays is quite efficient [20]). As mentioned in 2.1.4, we need to generate a set of RSA keys. This is most expensive for the bank. For instance, if a bank supported 100 different face values when issuing E-Cash, it would need to have 101 key pairs

[20] We use Protected, private and public keywords in Java to achieve this.
[21] A copy of the complete source files have been submitted with this project.

(1 extra for its public modulus and exponent for signatures) in its database. This often causes lengthy pre-computations, especially when generating keys of $|k| = 1024$ bits. For a customer, expensive computations arise when opening an account. Customers need to generate $|k|$ Williams integers of length $|k|$ each of which is obtained by multiplying two Williams primes of length $|k|/2$ bits. Therefore one can expect delays in pre-computations (we only accept Williams integers; hence many other computed values are discarded). This problem can be rectified when a bank has dedicated multiple ALUs (banks should be able to afford these). For the user however, this is often not possible, particularly when using a low powered device such as a PDA. But as this is a one time computation, we can perform our pre-computations using a regular workstation and then transfer our data to other devices.

1. **Our Results:**

   As described in the theoretical efficiency section, it is clear that only some parts of the protocol could pose efficiency issues. These parts are therefore our focus in this section. In any case, for completeness purposes, we shall give a brief benchmark of most of our implementation. Our timings come from the time flags in our code and using profiling. The advantage of time flags is that it allows us to separate unnecessary parameters that would interfere with our results (e.g. screen drawing, saving to disk and etc. . .). We shall run two benchmark tests for each important computation, one with 512 bits of security parameter and the other with a more secure security parameter of 1024 bits.

2. **Preliminary Computations (Code Segment 1)**

   For the bank, as it needs to generate a large number of large primes (one for each value of E-Cash that it supports), we use a random prime generation method already provided[22] (it uses a probabilistic approach to prime generation with p(number is composite) $= 1/2^{30}$). Note that this is a one time operation; reloading our keys takes less than 0.5 seconds once they have been generated. The following statistics demonstrate our results:

   Parameters A: Security parameter $|k| = 512$, maximum withdrawal possible set to \$1024 with denominations of \$1.

   Time to complete operation A: 63331ms $\simeq$ 63 seconds (for generating 513 key-pairs).

   Total memory required A: 313Kbytes (200Kbytes compressed).

[22] We used "BigInteger.probablePrime(numberOfbits, certainty)" in Java

Profiling result A: Profiling showed that 61 seconds were spent on generating our primes which agrees with our expectations.

Parameters B: Security parameter |k| = 1024, maximum withdrawal possible set to $1024 with denominations of $1.

Time to complete operation B: 451813ms $\simeq$ 450 Seconds.

Total memory required B: $\simeq$ 505Kbytes (393Kbytes compressed).

Profiling result B: The profiling result was almost the same as that A (i.e. only 9 seconds were spent on non computational operations).

3. **Account Opening (Code Segment 2)**

   In this part, we observed how much time and space was required for generating blind candidates in order to complete the cut-and-choose method. The results given are excluding of network communication lags, resource usages and server side computations times.

   Parameters A: |k| was the same as parameters A in (1), User ID of length 30 bits, random numbers were 64 bits long and our one-way hash function was of SHA-256 algorithm. Note that we generate |k| = 512 blind candidates.

   Time to open account A: 10719ms $\simeq$ 11 seconds

   Total memory required for licence A: 211Kb, 88Kb compressed. This result includes all blind candidates as the customers need them when making payments.

   Profiling result A: Profiling showed that 10.8 seconds were spent on blind candidate generation. This result is in line with our timings.

   Parameters B: As with parameters of A; but this time we generated 1024 blind candidates.

   Time to open account B: 98750ms $\simeq$ 99 seconds.

   Total memory required for licence B: 551Kbytes, 305Kb compressed.

   Profiling result B: This time the opening procedure took longer (146 seconds). However the actual time spent on computations was significantly less (i.e. 99 seconds) compared to the total duration. Profiling showed our Java sockets used the extra time to communicate the blind candidates back and forth.

4. **Buying E-Cash (Code Segment 3)**

   In this part, the relatively expensive operation was the creation of a wallet (HST). However as we can see, computation times are almost insignificant compared to the rest of the protocol.

Parameters A: |k| = 512, user ID length = 30 bits, random blinding-factor length = 30 bits, one-way hash function used was of SHA-256 algorithm.

Time to buy E-Cash and generate wallet A: 82 ms $\simeq$ 0.08 seconds

Total memory required for HST A: 16Kbytes or 330bytes compressed.

Profiling Result A: Our profiling showed that bottlenecks were caused by the buffers for connections. As the bottlenecks are irrelevant to our goals, they been excluded from our timings.

Parameters B: |k| = 1024 bits, user ID length = 30 bits again, random blinding factor length was the same as A, one-way hash function used was of SHA-256 algorithm.

Time to buy E-Cash and generate wallet B: 105 ms.

Total memory required for HST B: 16.1Kb raw or 313bytes compressed. (the wallet size is almost the same in both A and B).

Profiling Result B: Same as profiling for A.

5. **Payments, Transfers and Deposits (Code Segment 4)**

   This part concentrates mainly on payments. This is because transferring money is principally the same as paying to a merchant where the merchant is a customer at the receiving end. Depositing money back to banks has also been shown to be a linear operation and is not in our focus. Therefore we give a combined result in this part.

   Parameters A: Same customer and bank as in (1A), (2A) and (3A) with a merchant with a 512 bits security parameter.

   Total time for spending ½ of total wallet A: We assume our customer is spending \$501 (first we spend \$1 and then \$500 to observe changes in transaction times and wallet sizes). Hence we have to use more than one node of the HST. The results indicated that for \$500, the user selected 14 coins to pay with which resulted in a transaction time of 10.1 seconds. Compared to our spending of \$1, this took 20 times longer. This is because there are more challenge-response games that need to be played when using more nodes.

   Total transaction history size A: 172Kb compared to 20Kb after spending \$1. When compressed however, these values dropped to 38Kbytes and 5Kbytes respectively.

   Wallet size after transaction A: There was an insignificant change in the wallet size after spending \$1 (remember that \$1 would not have had any children so there is no size loss here), however after spending \$500, our

wallet size reduced to 8Kbytes which is ½ of its original size. Hence we conclude that wallet sizes reduce in relation to the amount of money left in them, which is what we desired in the first place. The compressed wallet size for the same transaction was 276Bytes.

Profiling result A: In our profiling, the most significant bottleneck was the transferring of large challenge response queries from one side to the other. Again this is not of interest to us as we are merely benchmarking the computational parts of our implementation. The computation of X tables also took some time. However routines that were responsible for executing these operations took 4.38 seconds for a 512 bit security parameter.

Parameters B: Same customer and bank as in (1B), (2B) and (3B). This time the merchant is compatible with a 1024 bit security parameter.

Total time for spending ½ of total wallet B: Again we tested two transactions ($1 and $500). However, the total transaction took 42 seconds. The time taken for spending $1 in B was 4 times that of A.

Total transaction history size B: 344Kb compared to 37Kb after spending $1. The compressed history data sizes were 73.2Kb and 8Kb respectively.

Wallet size after transaction B: Almost the same sizes as A with a compressed wallet size of 313bytes after spending $500.

Profiling result B: Profiling for B did not reveal any extra information (any information relevant to our work; we are not interested in how long transferring data would take). Therefore it suffices to rely on our own results.

6. The Significance of Our Results

Now that we have demonstrated our implementation, we shall interpret our efficiency benchmark results.

In test (1) we saw that our key generation computations with an acceptable security parameter took long. In fact it took fact 2^3 times longer than generating keys of 512 bits. This is not necessarily a bad result as most E-Cash schemes, even the most recent versions, suffer from long pre-computations. But pre-computations are just that, meaning that we do not perform them in real time; nevertheless, once we have our pre-computation results, we can save and restore them. In terms of key storage, the results pose no significant (for banks at least) storage issues.

The account opening results almost perfectly followed our expectations, meaning that opening an account with $|k| = 1024$ would take exponentially longer to complete ($Time_B = 3^2 Time_A$). User licences also proved to

take almost 2.5 times as much space in B. Therefore opening an account on a low powered device would be extremely time-consuming. In any case, customers would be more likely to open accounts on their home workstations or at a Bank where there is more computing power to avail.

Part (3) revealed some very promising information regarding fund withdrawals in terms of its efficiency. We saw that it would take less than a second on a home PC to withdraw funds. Even assuming a low powered device, this would not need more than several seconds as we only deal with several exponentiations when buying E-Cash. We also saw that a wallet containing $1024 only needed 16Kbytes of storage, regardless of the security strength of our system. It is even more impressive to see our wallet compress to about 300bytes which fits comfortably on a smart card (even 16Kbyte cards are more than practical these days). The last but not least important result of test (3) was the linear reduction in size of our version of the HST. This demonstrates that we would be able to reload our wallet as soon as we have spent some E-Cash (we now have free space), so we would not have to worry about not having enough E-Cash in our wallet.

In the spending protocol, the time taken to finish our challenge-response rounds for B was exponentially longer than A; it was 2^2 times longer than A. This is perhaps where work needs to be done to improve transaction speeds. It would be very difficult to imagine a low powered computer trying to complete a transaction that requires so much computation power. The transaction histories would be more acceptable when compressed as they would fit on a smart card or a similar device.

2.2.3 Alternative Approaches

Having reviewed the theoretical and practical efficiencies of the divisible E-Cash schemes, in this section, we reanalyse them to see whether a better design approach could be taken to improve efficiency. The first obvious bottleneck in this protocol is the growth of data size of payments and transfers after each payment or transfer. As we saw earlier, regardless of the HST design, using more nodes would result in larger transfer amounts. Even worse, if a coin is transferred to another user, regardless of its face value, we would have to transfer the history of our previous payments. Therefore our wallet which is now smaller in size still carries a large payment history which is space inefficient. Hence one might ask whether matters could be improved by taking a different approach to the design of the protocols. We shall discuss two improvements here. The first is an improvement introduced by [21] and the second was designed by Okamoto himself [22].

Note: The latter paper was recommended by Dr. Okamoto as an extra reading following a discussion with him regarding his protocol[23]. It should be noted that we are bounded by space restrictions here; it is left to the reader to read the papers for a full protocol description; we try to avoid repetition of statements by doing this.

1. In [21] the authors present an improvement to the original protocol by introducing the concept of cheque books rather than wallets. This means that when a customer is spending E-Cash, she will be paying with one page of her book. This has the advantage of not making payments with several coins. In brief, the first and second parts of the improvement exactly follow Okamoto's paper (i.e. preliminary preparations, opening accounts and E-Cash withdrawal). The only difference here is that we make payments with a page of our cheque book rather than a series of smaller coins. When it comes to making payments with E-Cash, rather than dealing with nodes (say Γ_0, Γ_{01}) we deal with a single page. A payment consists of a page j, previous account balance and a timestamp. For example X values can be computed by $X_j = (j, r, T, g_1(g_2(T||ID_p)), Q_{j-1}, Q_j, \{Z_{j,i}\})$ where j is the payment page, r is a random value, T is a time stamp, g_h are hash functions (one-way), Q_j is the value printed in the current page and $Z_{j,i}$ is $\langle g(j||C)\rangle_{QR_N}$ where C is the cheque book itself (equivalent to C in part two of Okamoto). As a result many intermediate coin usages will be avoided.

2. In [22], Okamoto improves the efficiency of E-Cash by introducing a new protocol based on the work of Ferguson [15] and Brands [14]. Okamoto uses "single term coins"[24] instead of the cut-and-choose (for example in our implementation this was choosing k/2 of customers' blind candidates by the bank to verify trustworthiness) methodology which he used in his previous work. Almost all of the work on E-Cash since then has been a fusion of the two ideas. However this comes at the cost of code complexities which make implementations extremely difficult and time consuming. However, the use of randomised blind signatures, secret sharing protocols and secure hashing algorithms has remained the same since Okamoto's paper. As stated before, describing the complete protocol would not be possible as space would not permit. In any case, we shall consider its efficiency in the next section.

[23] Our communication and his recommendations occurred at the time of compiling this thesis.

[24] Single term means that instead of having may coins in our E-Cash we have a single term; this is very similar to [22] but more complex.

Important Note: In Asiacrypt'96, Chan et al presented an attack on Brands' scheme and showed how their attack could be applied to Okamoto's second scheme (remember that [22] uses ideas introduced by Brands in 1993). In fact, they demonstrated two attacks on Brands' and one on Okamoto's protocols by showing how users could misrepresent themselves (without being detected) in the absence of cut-and-choose methodology (which is employed by the protocol we implemented in this document). As a result users could double spend.

3 Result Comparisons

Now that our analysis and benchmarking work is complete, we move on to present brief results of some very important schemes that appeared after [12]. This section also brings all of our work together which assists us in making sensible conclusions in section 4. Note that each protocol may have unique features that would not allow us to compile a global comparison table. For instance, in some cases we shall present results based on complexity of protocols as no actual data are available to use. It should also be noted that some comparisons cannot be made, as an equivalent of [12]'s procedures might not be available in the target protocols.

Note: Although a lot of effort has been put into making this project's results as accurate as possible, our comparisons may not be entirely accurate as all obtained results are host-platform dependant. Please note that the benchmarks given in the next parts are only theoretical; however we present an actual benchmark whenever possible[25]. Clearly the same host dependency rules apply to the schemes below.

3.1 Okamoto's Second Scheme

As described in 2.2.3, there have been other protocols designed based on "Universal Electronic Cash" one of which is Dr. Okamoto's second scheme based on his earlier work and Brands' scheme. We mentioned in 2.2.3 that there have been attacks on the both schemes; therefore they would not be suitable for implementation. As a result, the comparisons we make here are theoretical efficiency of [22] against the efficiency of our implementation[26].

Parameters: Bank security parameter of $|n| = 514$ bits (we need two of these). Customer key of (Williams product) $|N|$=512 bits, where $N = PQ$, $|P| = |Q| = 256$ and $(1/4)n_1^{1/2} < P, Q < (1/2)n_1^{1/2}$).

Assumptions: Wallet size = \$1000. Smallest amount possible = \$0.01.

[25] This is because implementations might not exist for some schemes.

[26] Data obtained from the results of the paper and manual calculations.

Theoretical efficiency: User data (licence and parameters) = 192bytes, Wallet size = 200bytes.

Account opening: This stage requires about 4000 multi exponentiations mod n (Even including any exponentiations, the account opening procedure should take less time than that of our implementation).

Results: Clearly this scheme is more efficient as far as memory needs are concerned. As for time, the payment protocol of this scheme is more efficient as a result of single term coin usage.

3.2 Project CAFE

CAFE[27] [24] (stands for "Conditional Access for Europe") is a product of joint efforts between many cryptographers in Europe. It is open source and prototypes of pocket sized electronic wallets are available. Its security can be verified (as it is open source). It is very interesting to know that even this project uses some ideas introduced in "Universal Electronic Cash" such as the HSTs. Results were obtained from [25, 26].

Total pre-computation time: With 1024bit DSA keys (equivalent to our pre-computations, but in DSA) pre-computations take about 80 seconds.

Account opening: Using a security parameter of 1024 bits, 14 Seconds.

Issuing E-Cash: About 0.5 seconds regardless of amount.

Paying by E-Cash: 0.07 seconds regardless of amount.

Wallet size: Total wallet size regardless of amount takes less than 3 Kbytes.

3.3 Brands' Scheme

The efficiency of Brands' scheme is principally the same as early versions of CAFE in 1994-1995. No known implementations of this protocol exist[28]. Early versions of CAFE used Okamoto's second protocol and Brands' scheme; hence the similarity in their efficiencies.

3.4 Compact E-Cash

This is the most recent invention in E-Cash which was presented in Euro-Crypt'05. So far no known implementations have been presented, therefore we shall consider its efficiency theoretically. The protocol is divided into two sub-protocols of which the latter is an extension to the former with added coin traceability.

[27] http://www.semper.org/sirene/projects/cafe/index.html

[28] I have researched many sources including the author's homepage for this paper which promises an implementation to date.

Note: Each wallet in Compact E-Cash can have different number of coins in $o(\log k)$ (where k is the security parameter of the issuer, say 1024 bits).

Preliminary computations: The signatures used in this protocol are an interesting modification of strong RSA signatures called CL signatures [27]. The bank has a CL signature key pair for each coin value of size $O(l \log q)$ (q is a prime, say 512 bits long) and does around ql modular exponentiations to obtain this. Customers on the other hand have one unique key pair of large a modulus k.

Wallet size: In protocol one, the wallet size of a user who has 1024 coins is $O(l + k)$ bits. In the second protocol, the user's wallet has size $O(l \times k)$ as traceability comes at a storage cost.

Result: "Compact E-Cash" is much more space efficient compared to "Universal Electronic Cash" in both cases. This is apparent as in the latter we have wallet sizes of $O(2^l)$. The efficiency of the preliminary computations cannot be directly compared as the setup procedures are different. We are also unable to obtain actual timings for "Compact E-Cash" as there are no realisations to date, perhaps due to the implementation scale.

4 Concluding Remarks

We shall finish this document by drawing conclusions on the usability of E-Cash and comment on its practicality.

4.1 Potential Uses for Our E-Cash Implementation

We briefly covered what E-Cash could be used for, but no particular potential uses of our implementation have been discussed. Although a secure and reliable scheme, [12] would not be an efficient scheme for use in large payments (refer to 2.2.2 for benchmarks). However this scheme could be easily implemented in snack machines, print credit systems, selling daily newspapers and many more. This conclusion is drawn based on our earlier results (i.e. a slightly smaller security parameter results in a large decrease in computational resources needed).

4.2 Future of E-Cash

As mentioned before, E-Cash has already been implemented by various companies. It is however interesting to see that many have not even heard of this phenomenon. Perhaps one of the reasons for this is that the need for E-Cash has not been created within societies. Besides, the tremendous cost of changeover from credit card systems to E-Cash reduces its popularity. Futurologists have

mixed views on the future of E-Cash; some predict the complete replacement of physical cash by 2020[29] and some predict a strong government presence (and/or credit card monopolists) holding back the spread of it[30]. One hurdle that certainly needs to be overcome is the issue of trust in E-Cash. No one would like to lose all of their money by losing their E-Cash devices (which does not happen with physical cash). The current trend of E-Cash is moving towards universally acceptable schemes where merchants would be able to deposit their received payments at any bank (i.e. all E-Cash banks could accept other banks' clients). This has certainly opened doors to more exciting possibilities and a starting point for further work.

4.3 Future Work

We have now covered an important basic E-Cash scheme and hopefully have a meaningful understanding of the ideas behind it. This realisation would be an ideal starting point for research, particularly the study of more advanced ideas inline with current E-Cash trends (such as the one mentioned in 4.2).

4.4 Final Words

E-Cash has come a long way since its conception. In this paper we concentrated on the good part of the last 15 years of research that has been put into it. It can be seen that E-Cash has become more and more efficient in terms of transaction execution, security and size, at the expense of more complex implementations. We also saw that almost all major protocols have been using Okamoto's "Universal Electronic Cash" in some or most parts of their schemes (and somehow no formal implementations existed for it yet[31]).

We covered our original aims by implementing the scheme (completely offline, divisible and transferable E-Cash with an easy-to-use interface) and by analysing its theoretical and practical efficiencies. Having considered alternative approaches to our implementation has provided a good starting point for future realisations of this protocol.

Our brief comparisons against other major schemes were particularly difficult to perform as many of them are yet to be implemented; therefore our comparisons remain open for corrections, should any realisations arise. We hope that our implementation would be a good starting platform for making comparisons against the efficiency of other protocols. We also hope that our work will perhaps be used as a teaching tool in cryptography lectures, as many teach their

[29] http://www.btinternet.com/∼ian.pearson/web/future/financialservices.rtf
[30] http://personal.law.miami.edu/∼froomkin/articles/cfp97.htm
[31] To the best of my knowledge none exist. I am open to be proven otherwise.

students about basic E-Cash without any working demonstrations[32] (this is due to the lack of availability of implementations).

[32] A simple search on the internet would demonstrate this.

5 References

1. D.Chaum. Blind Signatures for untraceable payments. In David Chaum, Ronald L Rivest, Alan T Sherman, editors, *Advances in Cryptology – Crypto '82*, Pages 199-203. Plenum Press, 1982.

2. David Chaum. Blind Signature Systems. In David Chaum, editor, *Advances in Cryptology – Crypto '83*, page 153. Plenum Press, 1983.

3. *Applied Cryptography* (1996). Bruce Schneier. Published by John Wiley & Sons. ISBN: 0-471-11709-9. pages 139 to 150.

4. Jan Camenisch, Susan Hohenberger, Anna Lysyanskaya. Compact E-Cash. *Lecture note series in Computer Science. Eurocrypt 2005*, pages 302-321. Springer Verlag, 2005.

5. S. Brands. Electronic cash systems based on the representation problem in groups of prime order. In *Preproceedings of CRYPTO '93*, pp. 26.1–26.15, 1993.

6. D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, Oct. 1985.

7. D. Chaum, A. Fiat, and M. Naor. "Untraceable electronic cash". In *CRYPTO '90*, vol. 403 of *LNCS*, pp. 319–327, 1990.

8. T. Okamoto and K. Ohta. Disposable zero-knowledge authentications and their applications to untraceable electronic cash. In *CRYPTO*, vol. 435, pp. 481–496, 1990.

9. Y. S. Tsiounis. Efficient Electonic Cash: New Notions and Techniques. PhD thesis, Northeastern University, Boston, Massachusetts, 1997.

10. J. L. Camenisch. Group Signature Schemes and Payment Systems Based on the Discrete Logarithm Problem. PhD thesis, ETH Z¨urich, 1998.

11. A. Chan, Y. Frankel, and Y. Tsiounis. Easy come – easy go divisible cash. In *EUROCRYPT '98*, vol. 1403 of *LNCS*, pp. 561–575, 1998.

12. T. Okamoto, K. Ohta. NTT Labs. Universal Electronic Cash. Advances in Cryptology – Crypto '92, page 324-325. Springer-Verlag, 1998.

13. Mårten Trolin. A Universally Composable Scheme for Electronic Cash. *Progress in Cryptology - INDOCRYPT 2005: 6th International Conference on Cryptology in India*, Bangalore, India, December 10-12, 2005., Springer-Verlag, 2005.

14. S. Brands. Untraceable offline cash in wallets with observers. In *Advances in Cryptology – Crypto '93, volume 773 of Lecture Notes in Computer Science*, Pages 302—318. Springer Verlag, 1994.

15. N.T. Ferguson, "Single term offline coins", in *Advances in Cryptology – EuroCrypt '93, volume 765 of Lecture Notes in Computer Science*, pages 318 – 328. Springer Verlag, 1993.

16. Schnorr, C.P.,"Efficient Signature Generation by Smart Cards", *Journal of Cryptology*, vol.4 no.3, 1991, Pages 161-174.

17. Chang Yu Cheng, Jasmy Yunus, Kamaruzzaman Seman. "Estimations on the Security Aspect of Brand's Electronic Cash Scheme," *aina*, pp. 131-134*, 19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 2 (INA,, USW,, WAMIS,, and IPv6 papers)*, 2005.

18. L. Carter & M. Wegman, "Universal classes of hash functions", *Journal of computer and system sciences*, 1979, pp. 143–154.

19. D. Chaum and T. Pedersen, "Transferred cash grows in size", in *Proc. EUROCRYPT'92. Lecture Notes in Computer Science 58* (1993), 390-407.

20. E. Bach and J. Shallit. "Algorithmic Number Theory: Efficient Algorithms", *volume 1. MIT Press, Cambridge MA*, 1996.

21. O. Watanabe & O. Yamashita, "An Improvement of the Digital Cash Protocol of Okamoto and Ohta", *Lecture Notes In Computer Science; Vol. 1178. Proceedings of the* 7^{th} *International Symposium on Algorithms and Computation*. Springer-Verlag, 1996, Pages: 436–445.

22. T. Okamoto, "An Efficient Divisible Electronic Cash Scheme$''$", *Advances in Cryptology - Crypto '95, LNCS, Vol. 963,* Springer-Verlag, pp438-451, 1995.

23. A. Chan, Y. Frankel, P. MacKenzie and Y. Tsiounis. "Mis-representation of identities in ecash schemes and how to prevent It$''$. *In Advances in Cryptology – ASIACRYPT '96 Proceedings, Lecture Notes in Computer Science vol. 1163*. Springer-Verlag, *pp.* 276-85, Berlin, 1996.

24. J.P. Boly, A. Bosselaers, R. Cramer, R. Michelsen, S. Mjølsnes, F. Muller, T. Pedersen, B. Pfitzmann, P. de Rooij, B. Schoenmakers, M. Schunter, L. Vallée, and M. Waidner, "The ESPRIT project CAFE - High security digital payment systems", *Proceedings ESORICS'94, LNCS 875*, D. Gollmann, Ed., Springer-Verlag, 1994, pp. 217-230

25. BCMM_1995 A. Bosselaers, R. Cramer, R. Michelsen, S. Mjølsnes, F. Muller, T. Pedersen, B. Pfitzmann, C. Radu, P. de Rooij, B. Schoenmakers, M. Schunter: "Functionality of the Basic Protocols", *CAFE Public Report IHS8341, CWI Amsterdam*, October 7, 1995.

26. Arnd Weber, Bob Carter, Birgit Pfitzmann, Matthias Schunter, Chris Stanford, and Michael Waidner. "Secure internation payment and information transfer - towards a multi-currency electronic wallet. Principles, Results from Initial Surveys, Scenarios.". *Conditional Access For Europe CAFE, 1995.*

27. J. Camenisch and A. Lysyanskaya. "A signature scheme with efficient protocols". *In SCN 2002, vol. 2576 of LNCS,* pp. 268-289, 2003.

6 Appendix A: Selected Code Segments

In the following pages, we have included programming code that is essential to subsection y2.2.2 and our results.

Notes: The reader should note that we have not included all parts of our implementation in this document as we are bounded by space restrictions. The reader should also note that we have removed any extra function calls and parts that are irrelevant to our analysis in this document as again we are bounded by the same restrictions. The full programming code has been submitted with this document and it can be viewed for more details and comments. Often comments have been put at the beginning of each method to make them compatible with JavaDoc.

The code might look a little confusing here as extra line formatting and space have had to be removed to save paper. However our submitted code is well formatted for easy understanding.

Segment 1: Bank.java → new Bank(…)

```
/**Creates a new bank with the details send by input args*/
public Bank(String name, String address, int secParam, int   maxCashIssued,
int cashDenominations, Vdu vdu) {

        try {

            long execTime;
            vdu.println("Building bank. Please wait....");
            int fileNameInc = 1;
            File bankFile = new File(ECash_Bank.bankFilePath + fileNameInc +
             ECash_Bank.bankFileExt);
            File transactionFile;
            ObjectOutputStream oos;
            BigInteger[][] ecashKeys = new
             BigInteger[(maxCashIssued/cashDenominations)+1][3];
            BigInteger pa, qa;
            SecParam = secParam;
            PQLength = SecParam/2;
            int totalCashKeys = (maxCashIssued/cashDenominations);
            float percent = (float)((float)100/(float)totalCashKeys);
            vdu.prepareProgressBar(true);
            execTime = System.currentTimeMillis();
            //Extra lines removed. See full source.
            for (int i = 0; i <= totalCashKeys; i++){

                pa = new BigInteger(PQLength, 30,new Random());
                qa = new BigInteger(PQLength, 30 ,new Random());
                modulus = pa.multiply(qa);
                BigInteger Phi = pa.subtract(BigInteger.ONE);
                Phi = Phi.multiply(qa.subtract(BigInteger.ONE));

                do{

                    pubExponent = new BigInteger(SecParam,
                                        new Random());

                }while ((pubExponent.compareTo(Phi) != -1) ||
                (pubExponent.gcd(Phi).compareTo(BigInteger.ONE) != 0));

                secKey = pubExponent.modInverse(Phi);
                ecashKeys[i][0] = pubExponent;
                ecashKeys[i][1] = modulus;
                ecashKeys[i][2] = secKey;
                vdu.percentPrint((int) (i*percent));

            }
            execTime = System.currentTimeMillis() - execTime;
            vdu.println("Done.");

            while (bankFile.exists()){

                fileNameInc++;
                 bankFile = new File(ECash_Bank.bankFilePath +
                        fileNameInc + ECash_Bank.bankFileExt);

            }

            bankFilePath =  ECash_Bank.bankFilePath + fileNameInc +
                                              ECash_Bank.bankFileExt;
            transactionFilePath = "Transactions for " +bankFilePath;
            transactionFile = new File(transactionFilePath);
            transactionFile.createNewFile();
            oos = new ObjectOutputStream(new
                  FileOutputStream(bankFile));
            oos.writeObject(name);
            oos.writeObject(address);
            oos.writeInt(secParam);
            oos.writeInt(maxCashIssued);
            oos.writeInt(cashDenominations);
            oos.writeObject(ecashKeys);
            oos.close();
            vdu.println("Bank created successfully.\nFile name:
                                        "+bankFile.getName());
            line(vdu);
            vdu.println("Operation took "+execTime+"ms");
            vdu.println("The generated data have
               "+bankFile.length()+" bytes size.");
            line(vdu);


        } catch (Exception e) {

            vdu.println("An I/O error occured.\nDetails: "+e);
            return;

        }

        System.gc();

    }
```

Segment 2: Customer.java → openAccount(…)

```
(Note extra spaces have been removed to save printing space)
protected Boolean openAccount(BankDetails bankDetails, ECash_Customer
ecustomer){

        Boolean success  = false; // Has account opening been successful?
        vdu.prepareProgressBar(true);

        try {

            File logFile = new File(customerLogFileName + "AccountDetails");
            FileOutputStream os = new FileOutputStream(logFile);
            ObjectOutputStream ps = new ObjectOutputStream(os);
            long execTime, tempExecTime;
            SecParam = bankDetails.securityParameter;
            CryptoLib cryptoLib = new CryptoLib();
            Random random = new Random();
            execTime = System.currentTimeMillis();//Start timing the execution
            BigInteger[] a = new BigInteger[SecParam];
            p = new BigInteger[SecParam];
            q = new BigInteger[SecParam];
            N = new BigInteger[SecParam];
            BigInteger[] blindCandidate = new BigInteger[SecParam];
            BigInteger temp, temp2;
            BigInteger[] r = new BigInteger[SecParam];
            Socket connection;
           //Step 1: Generate random numbers for later use in the account
       opening procedure.
            for(int i = 0; i < SecParam; i++){
                a[i] = new BigInteger(IDlength, random);
                p[i] = BigInteger.probablePrime(PQLength, new Random());
                q[i] = BigInteger.probablePrime(PQLength, new Random());
                r[i] = new BigInteger(SecParam, random);
                while(!p[i].mod(CryptoLib.WilliamsMod).equals(CryptoLib.three)
                ){
                    p[i] = BigInteger.probablePrime(PQLength, new Random());
                }

                while(!q[i].mod(CryptoLib.WilliamsMod).equals(CryptoLib.seven)
                ){
                    q[i] = BigInteger.probablePrime(PQLength, new Random());
                }
                N[i] = p[i].multiply(q[i]);
            }
            tempExecTime = System.currentTimeMillis() - execTime;
            ps.writeObject(N);
            ps.writeObject(p);
            ps.writeObject(q);
            ps.writeObject(a);
            ps.writeObject(r);
            vdu.percentPrint(17);
            System.gc();
            execTime = System.currentTimeMillis();
            //Step 2: The blind candidates are calculated and sent to the
             bank.
            blindCandidate = cryptoLib.OkaFormBlindCandidates(bankDetails,
             this, a, customerSecKey, N, r);
            I = cryptoLib.I;
            tempExecTime += System.currentTimeMillis() - execTime;
            ps.writeObject(blindCandidate);
            ps.writeObject(I);
            try {
                connection = new Socket(ecustomer.BANK_NET_ADDRESS,
                 ecustomer.BANK_PORT);
                 ObjectOutputStream sendData = new
                 ObjectOutputStream(connection.getOutputStream());
                 ObjectInputStream serverResponse = new
                 ObjectInputStream(connection.getInputStream());
                 sendData.writeObject("_OpenAccountRequest_");
                 vdu.println("Request to open an account sent to bank. Waiting
                 for response...");
                 if(serverResponse.readObject().equals("_Ready_")){
                     vdu.println("Response received." +
                     " Please wait while your accout opening request is being
                     processed...");
                     sendData.writeObject(blindCandidate);//send blind
                     candidates to bank.
                     vdu.percentPrint(34);
                     execTime = System.currentTimeMillis();
              //Step 3: Getting a list of Blind candidates indices from bank.
                     blindIndices = (int[]) serverResponse.readObject(); //We
                     will need this for Step 6
                     vdu.percentPrint(51);
              //Step 4: Sending the Required blind candidate components to the
               bank.
                     sendData.writeObject(ID);
                     int j;
                     BigInteger[][] blindComponent = new
                     BigInteger[SecParam/2][5];
                     for(int i = 0; i < SecParam/2; i++){
                         j = blindIndices[i];
                         blindComponent[i][0] = a[j];
                         blindComponent[i][1] = p[j];
                         blindComponent[i][2] = q[j];
                         blindComponent[i][3] = cryptoLib.hashedValues[j];
                         blindComponent[i][4] = r[j];

                     }
```

```
    sendData.writeObject(blindComponent);
    if(serverResponse.readObject().equals("_Invalid_")){
        vdu.println("Verification of customer faild." +
         " Data tampered by user or network
         conditions.\nExiting.");
        success = false;
        vdu.prepareProgressBar(false);
        return success;
    }
    vdu.percentPrint(68);
    System.gc();
  //Step 5: Receive data containing the licence from the bank.
    BigInteger licenceData = (BigInteger)
    serverResponse.readObject();
    vdu.percentPrint(85);
    //Step 6: Extract customer's licence from the licence data
    received in Step 5.
    String randomIndices =(String)serverResponse.readObject();
    int indicesCounter = 0;
    licence = BigInteger.ONE;
    for(int i = 0; i < SecParam; i++){
        if(!randomIndices.contains("," + i + ",")){
            blindIndices[indicesCounter] = i;
            indicesCounter++;
            licenceData =licenceData.multiply(r[i].modInverse
            (bankDetails.modulus)).mod(bankDetails.modulus);
        }
    }
    licence = licenceData;
    tempExecTime += System.currentTimeMillis() - execTime;
    ps.writeObject(licence);
    line(vdu);
    vdu.println("Licence acquired.\nLicence =
   "+licence+"\nClosing connection to Bank...");
    vdu.println("Operation took "+tempExecTime+"ms");
    vdu.println("The generated data have "+logFile.length()+"
    bytes size.");
    line(vdu);
    ps.close();
    os.close();
    success = true;
    connection.close();
    serverResponse.close();
    sendData.close();
}else{
    vdu.println("Unknown response received from
    Bank.\nExiting");
    success = false;
    vdu.prepareProgressBar(false);
                return success;
            }
        } catch (Exception e) {
            vdu.println("Error occured during account opening procedure.
            Details:\n"+e);
            success = false;
            vdu.prepareProgressBar(false);
            return success;
        }
        System.gc(); // Clean any held up memory
    } catch (Exception ex) {
        ex.printStackTrace();
        vdu.prepareProgressBar(false);
        success = false;
        return success;
    }
    vdu.percentPrint(100);
    return success;
}
```

Segment 3: Customer.java→buyECash(…)

```
/**The E-Cash buying protocol as in part II of Okamoto.*/
protected void buyECash(BankDetails bankDetails, String serverAddress, int
serverPort, int value){
    /* This is the preliminary stage of Part 2. Customer reguests
       public key pairs of the bank corresponding to the amount of
       E-Cash that he/she wants to buy.
    */
    BigInteger blindNote, hashedLicence;
    BigInteger blindingFactor = new BigInteger(bankDetails.securityParameter,
    new Random());
    BigInteger hashPadding = new BigInteger(IDlength, new Random());
    Socket connection;
    ObjectOutputStream sendData;
    ObjectInputStream receivedData;
    BigInteger[] quadRes = new BigInteger[bankDetails.securityParameter/2];
    /* Stage 1 of Part 2 of the protocol. Here the user forms a candidate for
       a bank note of face value $value to be digitally signed by the bank.
    */
    try {
        connection = new Socket(serverAddress, serverPort);
        sendData = new ObjectOutputStream(connection.getOutputStream());
        sendData.writeObject("_RequestBuyECash_");
        vdu.println("Request to buy $" + value +
                " of E-Cash was sent to bank. Please wait...");
        receivedData = new ObjectInputStream(connection.getInputStream());
        if(receivedData.readObject().equals("_Ready_")){
            sendData.writeObject(value);
            String serverResponse = (String) receivedData.readObject();
            if(serverResponse.equals("_OK_")){
                CryptoLib cryptoLib = new CryptoLib();
                hashedLicence = cryptoLib.hash(licence, hashPadding);
```

```
                BigInteger bankNoteExponent = (BigInteger)
                receivedData.readObject();
                BigInteger bankNoteModulus = (BigInteger)
                receivedData.readObject();
                blindNote = blindingFactor.modPow(bankNoteExponent,
                bankNoteModulus);
                blindNote = blindNote.multiply(hashedLicence.mod(
                bankNoteModulus));
                blindNote = blindNote.mod(bankNoteModulus);
                vdu.println("Bank supports this value. Proccessing...");
                sendData.writeObject(blindNote);
               /*Stage 2 of Part 2: Now the customer receives the digitally
                *signed blind-note.
                */
                signedNote = (BigInteger) receivedData.readObject();
                cashDenominations = (Integer) receivedData.readObject();
                /* Stage 3 of Part 2: The customer extracts the signed note.*/
                signedNote = signedNote.multiply(blindingFactor.modInverse(
                bankNoteModulus));
                signedNote = signedNote.mod(bankNoteModulus);
                vdu.println("Bank note received. E-Cash serial no.
                "+signedNote+"\nIt must be spent in denominations of $" +
                cashDenominations);
                totalCash = value;
                quadRes = new BigInteger[bankDetails.securityParameter/2];
                hst = hst.populate(hst, value, "0", cashDenominations);
            }else{

                vdu.println("There was something wrong. The bank responded:
                "+serverResponse);
                return;
            }
        }else{
            vdu.println("Bank refused to issue E-Cash.");
        }
        connection.close();
        sendData.close();
        receivedData.close();
    } catch (Exception e) {
        vdu.println("Error occured while trying to obtain electronic
         bill.\nDetails: ");
        e.printStackTrace();
        System.exit(0);
    }
}
```

Segment 4: Customer.java→spendMoney(…)& getRTable(…)&getOTable(…)

```
/**This procedure interacts with merchants in order to pay them. It also
handles user's
 *wallet by calling HST. The performance of this method is logged.
 */
protected void spendMoney(BankDetails bankDetails, int value){
    try {
        File logFile = new File(customerLogFileName + "MoineySpending");
        File hstLog  = new File(customerLogFileName + "UserWallet");
        PrintStream hps = new PrintStream(new FileOutputStream(hstLog));
        FileOutputStream os = new FileOutputStream(logFile);
        ObjectOutputStream ps = new ObjectOutputStream(os);
        long execTime;
        /*
           This is the preliminay stage of Part 3 of the protocol.
           The customer computes values of ECash tree's root node.
        */
        execTime = System.currentTimeMillis();
        CryptoLib cryptoLib = new CryptoLib();
        int valueLeft = value;
        int[] randomBits;
        String[] paymentArray;
        String paymentNodes;
        HST tempHST;
        Socket socket;
        ObjectInputStream ois;
        ObjectOutputStream oos;
        int pos, pos2; //Holds the positions of the k=1, k=SecParam/2 (They
         are randomly chosen)
        BigInteger[] Y;
        BigInteger[] A;
        if(totalCash < value){
            JOptionPane.showMessageDialog(null,"You cannot spend more than
             your account balance!",
            "Error:",JOptionPane.ERROR_MESSAGE);
            return;
        }else{
            if(value <= 0){
                JOptionPane.showMessageDialog(null,"Invalid payment
                amount","Invalid entry",JOptionPane.ERROR_MESSAGE);
            }else{
                if((value % cashDenominations)!= 0){
                    JOptionPane.showMessageDialog(null,"You cannot pay with
                    this amount!\n" +
                    "You do not have the right coin combination to
                    pay.","Error:",JOptionPane.ERROR_MESSAGE);
                    return;
                }
            }
        }

/*Step 2 of Part 3: We work out what nodes need to be used for payments and
compute the X values for them*/
        Y = new BigInteger[bankDetails.securityParameter];
        A = new BigInteger[bankDetails.securityParameter];
```

```
socket = new Socket(ECash_Customer.MERCHANT_NET_ADDRESS,
 ECash_Customer.MERCHANT_PORT);
oos = new ObjectOutputStream(socket.getOutputStream());
ois = new ObjectInputStream(socket.getInputStream());
oos.writeObject("_Payment_");
oos.writeInt(value);
//to send I[i]

    for (int i = 0; i < bankDetails.securityParameter/2; i++){
        oos.writeObject(I[blindIndices[i]]);
        oos.writeObject(N[blindIndices[i]]);
    }

    oos.writeObject(blindIndices);
    oos.writeObject(licence);
    oos.writeObject(signedNote);

    if(!ois.readObject().equals("_licenceOk_")){

        vdu.println("Error: Licence test failed.");
        vdu.println("Exititng!");
        return;
    }

    tempHST = hst;
    paymentNodes = tempHST.getPaymentNodes(value, tempHST, vdu);
    paymentArray = paymentNodes.split(":");
    x = new BigInteger[paymentArray.length]
     [bankDetails.securityParameter/2];
    r = new BigInteger[paymentArray.length]
     [bankDetails.securityParameter/2];
    o = new BigInteger[paymentArray.length]
     [bankDetails.securityParameter/2];

   /*Here we precompute the 1st step of part III of the protocol*/
    for(int i = 0; i < bankDetails.securityParameter/2; i++){

       r[0][i] =  cryptoLib.getQuadRes(signedNote, N[blindIndices[i]],
       p[blindIndices[i]], q[blindIndices[i]],"0");

    }


    for(int k = 0 ; k < paymentArray.length; k++){

        //Alg for tree: Initiazlize. Add used boolean. search from
        left right for free slot.
        //read everything from keyboard. do the entry based on
         keyboard.(read levels)
        //test = new BufferedReader(new InputStreamReader(System.in));
```

```
        oos.writeObject("continue");

        for (int i= 0; i < bankDetails.securityParameter/2;i++){

            pos = blindIndices[i];
            x[k][i] CryptoLib.sqrModNegQR
            (getRTable(paymentArray[k], i),p[pos],q[pos],N[pos]);
            //follow the protocol with the bank
            //get random bits and calculate Yi(Step 4 of Part 3)

        }

        oos.writeObject(x);
        randomBits = (int[]) ois.readObject();

        for(int j = 0; j < bankDetails.securityParameter/2; j++){

            pos2 = blindIndices[j];
            A[j] = cryptoLib.getMultipleQuadRes(signedNote,
             N[pos2], p[pos2], q[pos2], paymentArray[k], "10");

            if(randomBits[j] == 0){

                Y[j] = cryptoLib.sqrModPosQR(A[j], p[pos2],
                  q[pos2], N[pos2]);

            }else{

                Y[j] = cryptoLib.sqrModNegQR(A[j], p[pos2],
                  q[pos2], N[pos2]);

            }

        }

        oos.writeObject(Y);

    }

oos.writeObject("_finished_");
execTime = System.currentTimeMillis() - execTime;
ps.writeObject(r);
ps.writeObject(x);
hst.writeHST(hst, hps);
ps.close();
os.close();
hps.close();
socket.close();
oos.close();
```

```
            ois.close();
            totalCash -= value;
            line(vdu);
            vdu.println("Operation took "+execTime+"ms.");
            vdu.println("The generated data have "+logFile.length()+" bytes
            size.");
            vdu.println("The user's compressed wallet size is "+
             hstLog.length()+" bytes size.");
            line(vdu);
            JOptionPane.showMessageDialog(null,"Payment successful.\nYou have
               $" + totalCash +" left in your account.",
               "Done",JOptionPane.PLAIN_MESSAGE);

        }catch(Exception e){vdu.println("Error in spend.
                              ");e.printStackTrace();}

}

/**This method deals with the Omega user privacy protecting variables.
 *Omega variables are essentsial in protecting the user's privacy when making
 *a payment.
 *Omega is part of Part III of the protocol
 */
private BigInteger getOTable(String kbEntry, int i){

    return (new CryptoLib().getPositiveRes(signedNote, kbEntry,
       N[i],p[i],q[i], "1"));

}

/**Lambda tables are representatives of blinding variables
 *They also help enforce a structre to HST payment paths.
 *A recursive use strategy has been devised for this method as
 *it reduced the code size by many lines.
 */
private BigInteger getRTable(String kbEntry, int i) {

    if(kbEntry.equals("0")){

        return r[0][i];

    }

    String newString = kbEntry.substring(0,kbEntry.length()-1);
    int power = (kbEntry.charAt(kbEntry.length()-1)-48)*2;
    return CryptoLib.sqrModQR(getOTable(newString, i).pow(power)
        .multiply(getRTable(newString, i)), p[i], q[i], N[i]);
}
```