An Accountable Decryption System Using Blockchain Technology and Intel Software Guard Extensions

Dominic Fraise

A thesis presented for an MSc in
Computer Science

School of Computer Science
University of Birmingham
September 11^{th} 2017

Contents

1 Abstract

This paper outlines a system that makes decryption accountable; this proof of concept implementation is based on how companies handle a users location data. A decryption request for location data must be generated to perform any decryptions. This request is added to a blockchain which generates cryptographic proofs that the blockchain contains evidence that the decryption has been requested. These proofs are sent alongside the request to a secure device running Intel SGX which verifies them. After the proofs are verified the ciphertext is decrypted and sent back to the requester. The owner of the file can then access the system to see who has decrypted their files and why, therefore making the party who requested the decryption of these files accountable.

2 Introduction

In today's digital age everything is recorded, stored and can be accessed upon request. This is invaluable in many aspects of life from academic research to around the workplace. Modern technology has replaced paper whereby information can simply be stored in the cloud or on any network connected machine and access files from anywhere. Unfortunately, with this sort of convenience there are risks involved - for example, there is a possibility that a criminal could attempt to access and steal online data. This is a reason why cryptography has become an important aspect of the internet. Cryptography is essential for online security, however when trusting a third party such as an employer or a mobile phone provider to handle all encrypted files, it cannot guaranteed that these institutions will not decrypt your files without necessary cause to do so.

Where important data is concerned, there is a balance to be struck between security and privacy. The government need to be able to observe patterns of criminal activity so they can prevent crime, however, in being able to do so, the privacy of the public becomes limited. With increased terror threat levels worldwide, governments are tightening surveillance of their nations, including the introduction of the Investigatory Powers Bill by the U.K. government [1]. This bill allows for 6 months of data on every citizen to be stored by internet service providers without a warrant or justification. This data can then be accessed by any government controlled institution, such as the counter terror police or fire services, with zero accountability. Many security researchers and human rights activists are concerned that access to this data could be abused with no way for the public to know if their information has been accessed. Further, there has been evidence to suggest that this bill may not be as effective as initially planned [2], as such, if institutions were held accountable for looking at these files, it could lessen the invasion of privacy of innocent citizens.

When considering current available technology, there is no personal or institutional liability for decrypting a file. This means that any party in control of the encrypted data could be accessing all that information and could then deny ever doing so, with no evidence that any decryptions were performed. The

system that has been created in this project is based on the concept outlined by Mark Ryan [3]. It ensures that any attempts to decrypt files must be requested, logged and verified before the actual decryption is carried out by a separate device. It uses blockchain technology to store decryption requests in a log which can be queried by a user to ascertain if their files have been decrypted and uses a device running Intel Software Guard Extensions (SGX) to perform the decryptions.

3 Background

The following section is designed for readers who have limited knowledge of the topics that are mentioned below. Each of these concepts have been used as a part of this project and during the description of the implementation knowledge of these topics is assumed.

3.1 Public Key Cryptography

Public key cryptography is one of the most widely used forms of cryptography on the internet due to its ability to allow parties to communicate without previously deciding on a key. Symmetric key cryptography uses the same key for encryption and decryption, this is usually an efficient method, however it is much less so if someone else will be decrypting the files you have encrypted. One of the most popular public key cryptographic algorithms is RSA developed by Ron Rivest, Adi Shamir, and Leonard Adleman in 1978 [4]. This cryptosystem is based on the concept of generating two keys, a public and a private key that are mathematically linked, so that the encrypting party does not need to know the decryption key.

The key generation process is as follows; two large primes p and q are selected randomly and multiplied together to calculate the value of the modulus N. Next, Euler's totient function [5] is used to calculate ϕ where $\phi = (p-1)(q-1)$. An exponent e is then chosen at random provided that $1 < e < \phi$ and that e and ϕ are co-prime. Finally, the exponent d is calculated by finding the multiplicative inverse of $e \; mod \; \phi$. The public key is the exponent e and the modulus N and the private key is d. Encryption of a message m is done by calculating $c = m^e \; mod \; N$ and the decryption of the cipher text c is given by $m = c^d \; mod \; N$.

This is what is known as plain RSA as it uses no additional features beyond what is stated above to enhance its security. Since Rivest, Shamir and Adleman published their paper outlining this cryptosystem, many vulnerabilities have been found RSA in this form such as susceptibility to dictionary attacks [6]. Plain RSA is still useful for demonstrative purposes due to its elegant nature and there are also many adaptations that can be made which can enhance the security of RSA such as using OAEP padding [7].

3.2 Hash Functions

A hash function is a cryptographic tool that can be used to create a pseudo-random fixed length output for any given input that is unique for that input. Provided the hash function is an effective one, any slight change to the input file such as flipping one bit, will drastically change the output file. They are widely used for verifying the integrity of a file that has been downloaded or received from an untrusted source. This is done by taking the hash of the file in question before transmission, sending the file, then taking the hash of the file upon receiving it and ensuring that the initial hash matches the final hash. If anyone has tampered with the file, the hashes will not match.

There are many different hash functions that are in use but most are built using the same Merkle–Damgård construction. This is a method of splitting up a file into sections, consuming one block at a time until only one fixed length block is left as the final digest. Each block has its bits permuted and combined with the result from the previous section giving a unique final output for a given selection of input blocks. The permutations in this process are defined by the hashing function that is used in that construction. SHA-1 has been widely used as a hash function for many years despite it having been removed from the official guidelines. This hash function has recently been broken by finding a matching output for two different inputs [8]. If two different inputs can be shown to give the same output for a given hash function, that function can no longer be considered secure as it cannot be relied upon to verify a files integrity. SHA-256 is a successor of SHA-1 with a 256-bit output therefore making collisions harder to find. There is also SHA-3 which is set to use a method other than the Merkle–Damgård construction and is promising to be more future proof than SHA-256 [9].

3.3 Blockchain

Blockchain is the underlying technology of the hugely successful crypto-currency Bitcoin [10]. This technology refers to any system that uses a cryptographic chain of entries where an entry cannot be edited or removed from the chain undetected. This feature makes it applicable in many different scenarios, from financial transfers to distributed file storage systems [11]. Bitcoin is one of most notable and successful uses of this technology due to its popularity for performing online financial transactions anonymously. The rationale for creating a crypto-currency is to take control away from any central controlling bank or institution [12]. Instead, all the data relating to each transaction is stored in a distributed blockchain, this means that there are many different copies of the same log of transactions kept in various locations. If a blockchain was not used and two of the logs did not match, there would be no way of knowing which was genuine. However, with the logs stored as a blockchain, if an entry gets edited that chain will no longer be valid. This is due to each block in the chain containing the cryptographic hash of the previous block. This allows for easy verification of a blockchain by computing the hashes along the chain and

ensuring that the value stored at the head remains unchanged.

These blocks are stored in what is known as a hash chain, this is where each block contains the hash of the previous block. This is a simple implementation and is ideal for situations where you only want to ensure none of the entries have been edited but does have efficiency drawbacks in some cases [13]. However, there is another data structure, a Merkle Tree [14] that allows for the efficient generation of various proofs such as audit and consistency proofs [15]. A Merkle tree is a like a hash chain but implemented as a tree instead of a list. The entries in the tree are the leaf nodes and each parent is the hash of its two children concatenated together. The Root Tree Hash (RTH) is a unique value that is defined by the given set of leaves. If any leaf is changed, the RTH will change as well.

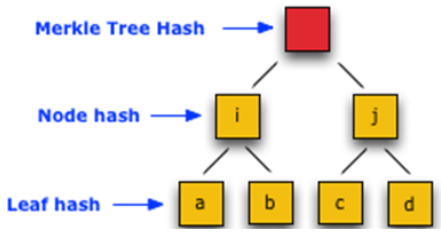

Figure 1: A Merkle Tree - The entry nodes are nodes a,b,c and d. Node i is given by $i = h(a||b)$ and the root is given by $RTH = h(i||j)$ where $h(x)$ is a hash function and $x||y$ is y appended to x

3.4 Intel SGX

Intel's Software Guard Extensions (SGX) is a relatively new product which contains a secure enclave that is separated from the rest of the memory. SGX provides software attestation and access control features that are provided by the processor. Inside this secure enclave, sensitive data and code can be stored without the rest of the system being able to access it directly. This section can only be accessed via remote procedure calls defined by the code in the enclave. A signed hash is received of the process running on the remote host, and this process is protected using access control and encryption. This makes it possible to establish a secure channel to the process, that even privileged software cannot access. SGX is useful in any situation where you have sensitive information on an untrusted system. A malicious user, or malicious software cannot access the data in the enclave. These properties make it ideal for storing cryptographic keys securely.

3.5 The Accountable Decryption Protocol

The main protocol, initially introduced by Mark Ryan [3] and expanded as a proof of concept in this project, is a method of combining blockchain and a secure device to create a system that provides accountability for decryption. This protocol can be applied to any system where the decryptions are made by someone other than the owner of the files and the owner of these files would want to be able to check if their data has been accessed.

User U generates ciphertexts by encrypting their files with a public encryption key ek. An administrative user Y decrypts ciphertexts by generating evidence e. This evidence e can be viewed by U to learn more information about the decryption that has been made. To ensure that e is created unavoidably every time, decryptions are performed by a separate secure device D. This device stores the most recent Root Tree Hash (RTH) stored in the log and a decryption key pk. Decryptions are performed by sending a request with e to this device D, thus ensuring that the decrypting party Y must generate e to retrieve the plaintext. The secure device examines e before it performs the decryption, if the device is satisfied by e it will perform the decryption and send back the plaintext, otherwise D will ignore the request.

For every decryption request, there is some corresponding e and this evidence must be stored for later examination by user U. This evidence is stored in a log L implemented as an append only Merkle tree outlined in section 3.3, allowing one root tree hash (RTH) to be published to define all evidence in the log at a given time. The log maintainer can provide on request 3 things: The current root tree hash, proof of presence of a leaf π and proof of extension ρ. Details on these proofs can be seen in section 5 of this paper. Evidence e is made up of a request R and proofs π and ρ. Decryptions are performed as follows:

- Obtain most recent RTH H from the device D
- Generate a request for a file R and add it to the log L
- Obtain new RTH H' from the log
- Obtain proof of presence π that R in the log with RTH H'
- Obtain proof of extension ρ that log with RTH H' is an append only extension of the log with RTH H
- Submit R, H', ρ, π to device D
- D verifies proofs π and ρ
- If the proofs are verified, D updates its RTH to H', decrypts record contained in R and sends the plaintext back to requester Y
- The log L can then be inspected by user U to see their file is contained in it

The generation and transmission of these proofs means that the log maintainer does not need to be a trusted party. The log maintainer cannot edit the log and generate proofs that will be verified by the secure device if the log has been tampered with. This protocol has been developed and expanded in this project. The implementation outlined in section 4 is based around the use of encrypted location data from your phone, where organisations such as Google or GCHQ would want to view what a users location data was at a given time, and a user can see if the data has been decrypted and, if so, by whom and why.

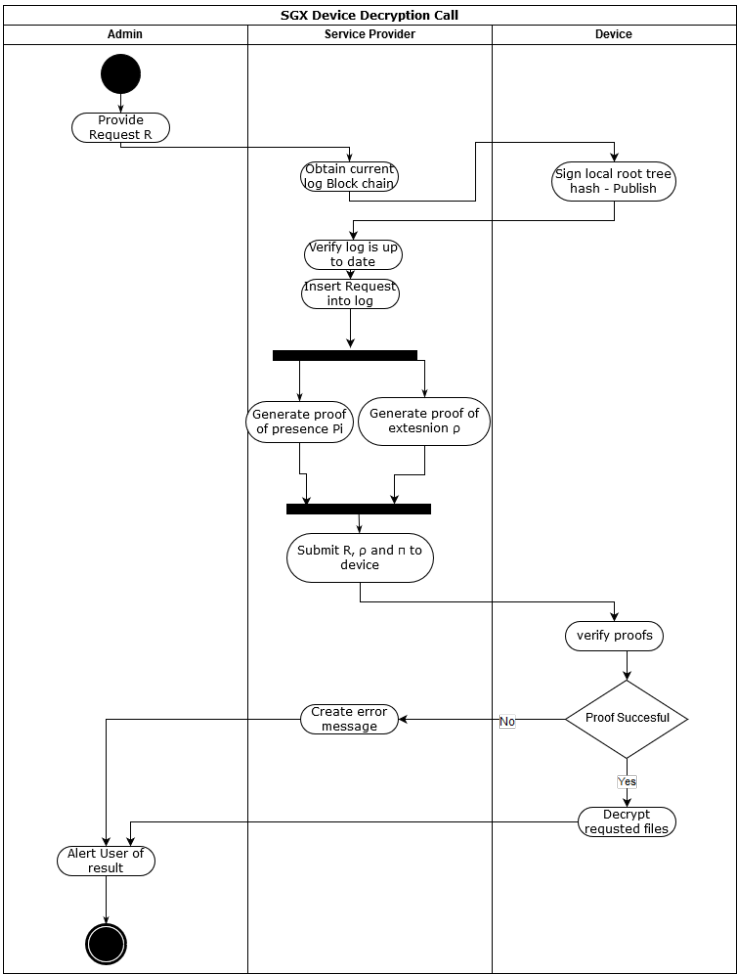

Figure 2: Activity diagram showing message flow during a remote procedure call *DecryptRecord*

4 System Overview

This system overview provides details on each component and the role it has within the system. A brief outline of the design and implementation of each component is detailed in its corresponding section below, along with details on how information travels around the system.

The system has 4 main constituent parts: the service provider, the log maintainer, the secure device and the front end. The service provider is the general name that will be used for the institution that has direct access to the encrypted data. It is this service provider that is central in the architecture as all requests and data pass through this server between the end user, log maintainer and the secure device. For a higher level view of the system as a whole, see section 4.5 for details on how the components communicate.

4.1 Log Maintainer

The log maintainer, referred to as the log L in section 3.5, could be a separate institution from the service provider, however in this project it has been implemented as a PostgreSQL database attached to the web server detailed in section 4.2. The log maintainer keeps a log of every decryption request that is made, maintains this log as a blockchain and generates cryptographic proofs to be passed to the secure device. The blockchain is stored as a Merkle tree in the database with each node in the tree as a record. The Merkle tree is implemented using SHA-256 hashes, each file that has its been requested has two entries inserted into the tree. One entry is the SHA-256 hash of the file that has been requested. This is so the owner of the file can query the log with only information on the file its self, without knowledge of the decryption request. The other entry inserted into the tree is all the information in the decryption request hashed together as another node. This stops any party who has made a decryption request from changing their justification or any other data about the request after the fact. The database also stores the log containing all leaf nodes that have been inserted, a record of the history of Root Tree Hashes and the full decryption request containing data such as the requester, when the request was made, what company they are affiliated to etc.

4.2 Service Provider

The service provider is implemented in this project as a web server that controls all communications in the system. The web server is a Java Servlet using the servlet engine Jetty. This web server has almost all the computational responsibility of this system as it also generates the proofs that would usually be done by the log provider. The server has 4 main responsibilities; to host the front-end website and handle requests from it, communicate with the log providers database to fetch data such as the encrypted files or requests, to generate audit and consistency proofs, and to perform remote procedure calls for decryption to the secure device. The web hosting section of this server is straightforward, it

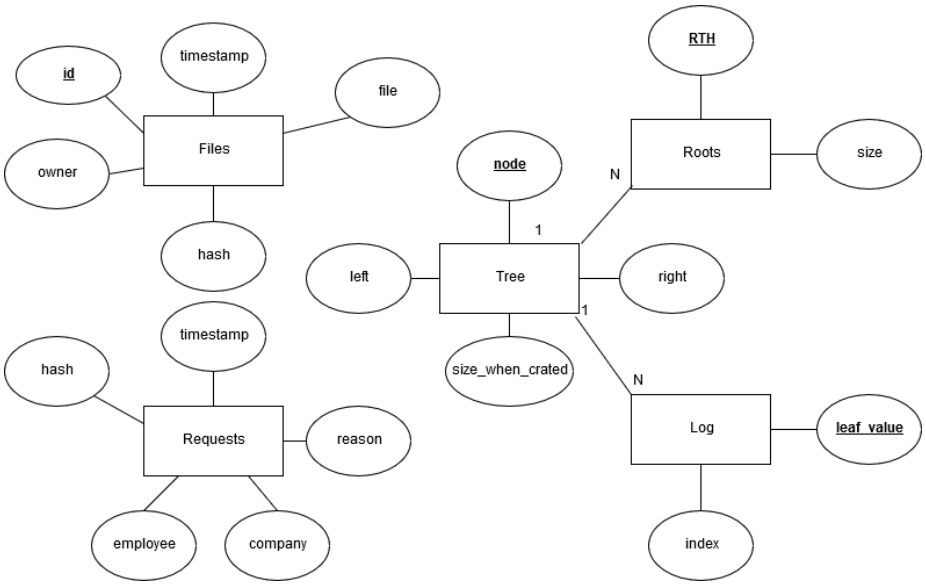

Figure 3: Entity relationship diagram - tables Log, Roots and Tree belong to the log maintainer and tables Files and Requests belong to the service provider

accepts "get" requests for various pages which is covered in more detail in section 4.4, it also accepts "post" requests where a user has submitted information to the system. Most "post" requests return results by inserting HTML snippets into the page the user is viewing giving the impression of a dynamic webpage. The server also generates all SQL requests to the log maintainer mentioned in section 4.1. The Remote procedure calls to the secure device is one of the more complicated responsibilities of the server covered in more detail in the following section.

4.3 Secure Device

The secure device is implemented here as an emulation of an SGX enclave described in section 3.4. The code for this enclave was written in Go created by Kristoffer Severinsen [16]. This device stores a value of the most recent root tree hash (RTH) stored in the log, a signing key to produce a signature on this RTH, and a RSA decryption key for decrypting records. The device acts as a server listening for remote procedure calls from a client which in this project is running on the web server outlined in section 4.2. This connection uses Google's remote procedure call protocol (gRPC) which uses pre-defined Protocol Buffer files to define the communication protocol across different languages and platforms [17]. There are 3 types of calls that can be made to the device; $GetRootTreeHash$ which returns the RTH and a signature by the device on the RTH value, $GetPublicKey$ which returns the public encryption key and the

verification key that can be used to verify the signature and *DecryptRecord* which will decrypt a record and return the plain text. Both $GetRootTreeHash$ and $GetPublicKey$ calls do not require any verification by the device as the information provided is publicly available, however *DecryptRecord* requires a different approach. The advantage of having a separate device performing the decryptions in this system is that the log maintainer does not need to be trusted. The log maintainer does not have access to the decryption key and must make requests via the device. When a user requests a decryption, the log maintainer must generate proof that the given request has been correctly added to the log (audit proof π), and that the new log is an append only version of the previous log (consistency proof ρ). These proofs are sent with the decryption request to the device and are verified. If the proofs are successfully verified the file is decrypted by the device and sent back to the requester. See section 3.5 for more detail on this protocol and section 5 for details on the proofs.

4.4 Front End

The front end to this system is implemented as a website. This site will have two types of users, administrative users who can request decryptions, and normal users who can only inspect the log to see if any of their files have been decrypted, referred to in section 3.5 as users Y and U respectively. Both types of users will also have access to the Proofs section of the site where can access the proof generation algorithms that are used in the system. The admin section includes three pages, a Search page, the Decryption Basket and the View Files page. An administrative user would use the search page to find data they may want to decrypt. In the case of this project, the system has been designed for accessing encrypted location data about users. Therefore, an administrative Y user can enter a username of a user they wish to know the location of, as well as a start date and time along with an end date and time. The results from this search will show the files from that user created in the given time window. The admin user must then select which files they wish to decrypt and provide justification of why they require the file. These requests can then be added to the decryption basket which is stored in memory on the web server. The user Y can observe the basket before the requests are made to ensure they are the correct files, if not they can be removed from this basket. After the basket has been examined the user Y can request the decryptions of the files by inserting their username and company they work for into the corresponding fields. The decrypted files can be found in the View Files page and can be downloaded as .txt files.

The other section of the website is intended for a normal user U who is only interested in inspecting the log. From this section, the user U can insert their username and query the log to see if any of their files have been decrypted. The results from this query are displayed in a table showing all of the requests that have been made for the user's files, along with other useful information such as who made the request, when it was made and the justification that was given.

4.5 Message Flow

The first interaction in the system must be message 1 shown in Figure 4 where a user uploads their encrypted files to the system. In this implementation, these files were uploaded manually, however in future iterations of this system they could be uploaded via a smartphone app. Once these files are on the system, administrative users can search for these files and generate a request, shown as message 2 in Figure 4. This message containing information on the requester and the requested file, is sent from the website to the service provider. The service provider then forwards this information to the log maintainer who inserts the request into the log. Cryptographic proofs are then generated by the log maintainer and sent back to the service provider. These proofs are then sent to the secure device along with the bytes of the ciphertext. If the device successfully verifies these proofs, the record is decrypted and sent back to the requester via the service provider. Details on these proofs can be seen in section 5.

There are other two interactions in Figure 4 which are not numbered due to them not fitting in the same message flow as they can be performed at any time during that process. Inspection is the act of a user viewing the log to see if their files are present and the requesting of the RTH is for verification purposes and can be accessed via the Secure Device tab in the Proofs section of the website. The RTH can be requested at any point by submitting a randomised nonce V and receiving back the RTH and V with a signature by the device on both items. This is a way to ensure the information received is up to date as the signature must have been created after V was submitted.

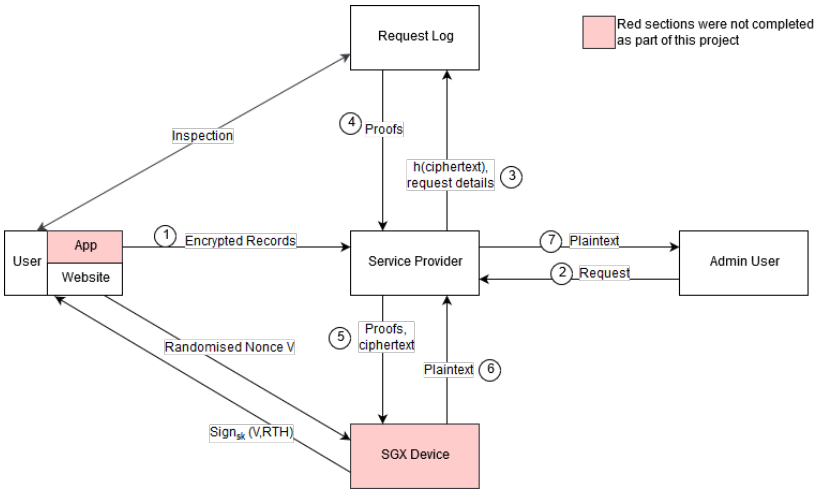

Figure 4: Architecture Diagram - Messages that are numbered outline the order of execution of the message flow. Unnumbered Interactions can be preformed at any time.

5 Proofs

The reason a tree structure was used for the blockchain instead of a hash chain, as mentioned in section 3.3, was that a Merkle trees allow for efficient generation and verification of various proofs [15]. Two of these proofs are audit proofs π, proof that a leaf is present in a tree, and consistency proofs ρ, proof that a new tree is an append only extension of a previous tree. In this section of the paper details of these proofs and their form are given in detail. Example outputs of these proof generation algorithms can be seen in appendix A

5.1 Audit Proofs π

Audit proof, also known as proof of presence, is required by the device to ensure that a decryption request has been added to the Merkle tree before the device proceeds with the decryption of the file. The proof generation algorithm requires access to the database and takes two parameters, a root tree hash (RTH) and the value of a leaf. The algorithm finds the tree in the database with the given RTH and begins to generate a semi-complete proof tree containing various intermediate nodes. This tree can be used to verify the leaf with the given value is present in the tree with the given RTH. The proof nodes are provided in such a way that anyone wishing to verify the proof should be able to calculate a RTH from the proof tree and this hash should match the one provided as a parameter. Figures 5 and 6 represent a simplified example of a Merkle tree and its corresponding proof tree respectively, for the audit proof for the third leaf. The leaves in the example do not contain SHA-256 hash values as they would in the actual system, single letter strings are used instead for simplicity.

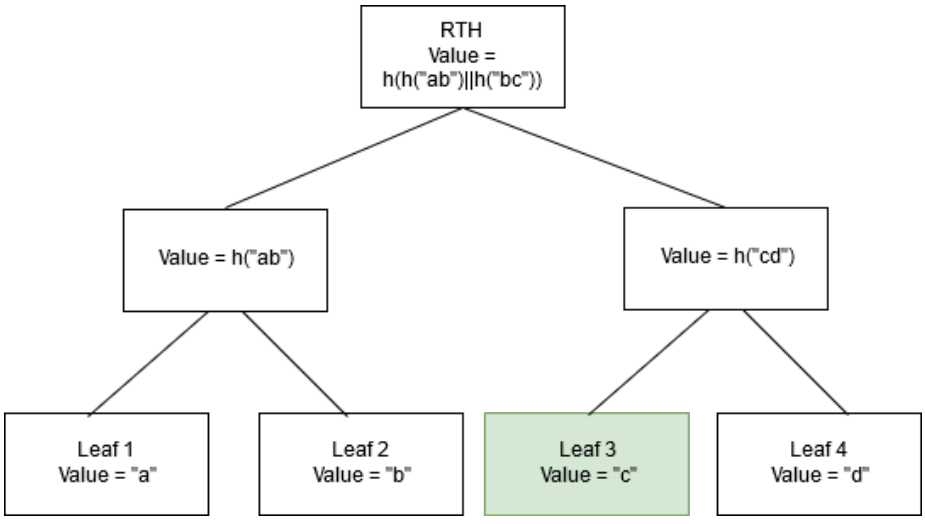

Figure 5: A representation of a Merkle tree with leaves a, b, c and d respectively. $h(x)$ represents the hash of x. $x||y$ represents y appended to x

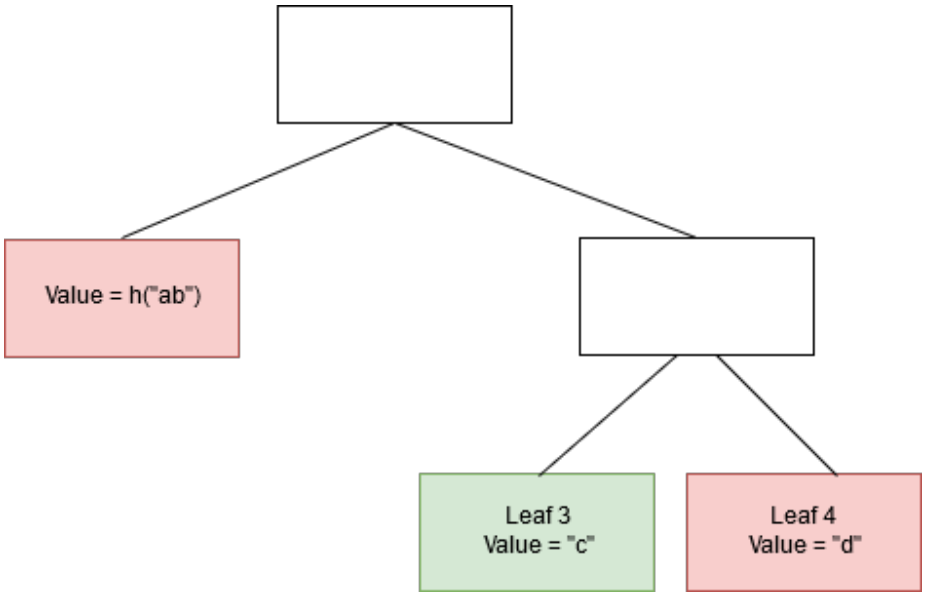

Figure 6: A representation of a proof tree for the presence of the leaf with the value "c" shown in Figure 5

The red nodes in the proof tree in Figure 6 are the nodes that would be provided in object returned from the proof generation algorithm and the green node is the leaf to be proven to be present in the tree. The empty nodes are to be filled in during the verification process by calculating the hash of its two children i.e. the lower right box would be calculated as $h(c||d)$. This process is continued until all empty nodes in the proof tree have been filled in up to the root. This root is then compared to the root provided in the Merkle tree in figure 5 and if the values match the proof has been verified. The proof tree is in the format of a JSON object, representing the nodes in the proof tree shown in Figure 6. At the top level of this JSON object there are 4 objects:

- RTH – A string representing the root tree hash in hexadecimal form of the Merkle tree containing the leaf in question
- $Value$ – The hexadecimal string stored in the leaf that is present in the tree
- $Index$ – The integer index of the leaf along the tree – counting from zero, left to right
- $Proof$ – A JSON object containing the proof tree

Each node in the proof tree is either another tree node or a terminating node. If It is a terminating node this means the node is either the leaf that is being proven to be present therefore containing the object:

– $Leaf$ – The hexadecimal string stored in the leaf that is present in the tree

Or it is a proof node containing an intermediate hash represented as red nodes in Figure 6:

– $Hash$ – The hexadecimal string containing the value stored in an intermediate proof node

If it is another node i.e. non-terminating, it contains two objects:

– $Left$ – a JSON tree containing any proof nodes in the left subtree

– $Right$ – A JSON tree containing any proof nodes in the right subtree

An example file output can be seen in appendix A.1.

5.2 Consistency Proofs ρ

Consistency proof, also known as proof of extension, is required to show that the new Merkle tree that has been created contains all the previous tree's elements in the same order. This is required by the device so it can ensure that no entries have been deleted from the log whilst making a new decryption request. The algorithm for the generation of this proof takes two parameters, the initial RTH i.e. the RTH before a new entry was inserted, and the new RTH i.e. the RTH after the new entry has been inserted. This proof is also in a JSON tree format, providing various intermediate nodes allowing a verifying party to recreate the RTH. Unlike Audit proofs however consistency proofs require two proof trees, one to recreate the initial RTH and one to recreate the new RTH.

In the top level of the JSON object there are 4 objects:

– $OldRTH$ – A SHA-256 hexidecimal hash showing the value of the RTH before some entry was inserted

– $OldProof$ – A JSON Tree containing the proof nodes needed to recreate $OldRTH$

– $NewRTH$ – A SHA-256 hexidecimal hash showing the value of the RTH after some entry has been added

– $NewProof$ – A JSON Tree containing the proof nodes needed to recreate $NewRTH$

The proof trees are organised in a very similar format to the trees produced from audit proofs. Each node in the proof tree is either another tree node or a terminating node. If it is a terminating node, i.e. has no children, the node is the following object:

– $Hash$ – The hexadecimal string containing the value stored in an intermediate proof node

If it is another node i.e. non-terminating, it contains two objects:

- $Left$ – a JSON tree containing any proof nodes in the left subtree
- $Right$ – A JSON tree containing any proof nodes in the right subtree

An example file output can be seen in appendix A.2.

5.3 Proofs of Absence

Unlike proofs of presence and extension, there is no efficient way to prove the absence of leaf in a tree. This proof is not required by the internal mechanisms of the system however if a user does not trust the results given from an inspection they can generate this proof. The only way to assert that a user does not have any results in the log is to download the entire log as a list of leaves and recreate a Merkle tree out of them, then assert that the RTH of this tree matches the one kept on the device. Once that has been verified the user must compare the list of leaves with a list of that user's files to see if the list contains any of their files. To generate this proof only a username is required.

As with the other two proofs this proof is provided as a JSON object and has 4 objects within it:

- $RTHFromDevice$ – A string containing the RTH currently stored in the device
- $DeviceSig$ – A signature of the device on RTHFromDevice
- $Leaves$ – An array containing all entries in the log
- $FileHashes$ – An array containing SHA-256 hashes of all the user's files

This is enough information to verify if a user's files are absent from the log. An example output can be seen in appendix A.3.

6 Software Engineering Process

During the creation of the software for this project, due to the systems unique nature, many of the design decisions had to be made as the system developed. Several of these decisions were made to either increase the usability or security of the system, however some decisions had to be made due to various difficulties that presented themselves. The main design decisions and some of the major problems that altered how the system was created are detailed below.

6.1 Creating the Merkle Tree

When deciding how to implement the decryption protocol outlined in section 3.5, the first thing that had to be considered was the form that the proofs ρ and π would take. However, to generate these proofs, a Merkle tree data structure was

needed to test and generate them. After considering various implementations that can be used for ensuring file integrity or developing efficient data processing systems, it was decided that a bespoke lightweight Merkle tree would be an effective and efficient starting point. Due to there being no real benefit in generating proofs in any language versus another, it was decided that this code would be written in Java due to its widespread usage globally. The Merkle tree started as a simple recursive binary tree with left and right nodes. Each node contained a value, with that value encoded as a string, and the height of that node above a leaf. A static method was written to insert an entry to the tree - this method inserts a value as a leaf and updates all the nodes up the tree to keep the tree valid.

As the algorithm for generating these proofs evolved, it became apparent that a full node, whereby all its children had exactly 2 children apart from the leaf nodes that have zero children, needs to be treated differently to an asymmetric tree. This meant that further information was needed in this data structure, including the number of nodes and the number of leaves. This allowed for more efficient calculation of whether a node was full or not, without having to check recursively. It was also decided that the way any data and proofs were to be transmitted through the system would be using Java Script Object Notation (JSON), as it is a clean and readable format that can be processed with libraries in all major programming languages.

It then became apparent that the log could potentially grow to millions or even billions of entries, meaning that the entire log could not be stored in memory, resulting in the current method for storing this tree being inadequate. As such, the tree was then implemented in a database with the following schema: $Tree\ (node, left, right, size\_when\_created, path\_length)$. This complicated the process of adding entries and generating proofs due to the manipulations of the tree needing to be done via SQL statements.

6.2 Creating the Web Server

After the database was created and proofs could be generated, the system could not function on its own without additional components such as, a simple Java server and client using a Swing UI, or an android app to upload and receive data from a server. The app was a strong contender but due to a lack of experience in android development it was decided it would be overly-ambitious in the time-frame that remained. In the end, a website was created with an HTTP server due to the freedom it gives users to be able to access the data from anywhere. An app would be a good addition in order to upload encrypted data files directly from your phone as a future iteration of this system.

At this point it was decided that the centre of the system connecting all the components would be done using a web server. However, this produced another problem; most web servers do their back-end processing using PHP, but almost all the code already developed and tested was written in Java. This prompted research into how Java code can be used in a web server. The most promising finding was to use Java Server Pages (JSP) to embed Java code into HTML files

in a similar way to how JavaScript is embedded. Not all web servers support JSP so an Apache Tomcat server was created on a local host as the school of Computer Science recently removed their Tomcat server. JSP with Tomcat produced a wide variety of problems such as installing and running the server and getting arbitrary Java imports to work. Progress was taking longer than anticipated, so an alternative was investigated: Jetty. Jetty is a web server like Tomcat, however with Tomcat, the server is run which then processes JSP files to run Java code. By comparison, Jetty is a web server that is embedded into your Java program. This meant that JSP was no longer be needed, as when using Jetty, the server can run Java methods in response to Get and Post requests as PHP would in a normal server.

6.3 Developing RPC For the Device

Once the database was created and the web server was up and running, the secure device needed to be integrated into the system. As mentioned in section 4.3 the secure Intel SGX device was developed by Kristoffer Severinsen in Google's programming language Go. Due to Go being a newer programming language, it was difficult to interface with the code for several reasons. Firstly, Go had to be installed in order to test the SGX code that would be used in this project and due to a lack of experience in Go programming, it was difficult to debug or to make minor changes to the code. Next, it was difficult to interface Severinsen's Go code with the Java web server in any meaningful way, so it was decided that Google's remote procedure call gRPC would be used for the calls into the enclave. Details on the different calls and information on gRPC itself can be found in section 4.3. Further, gRPC is designed so that a server written in one language can easily transmit data to and from a client written in a different language. However, it was difficult to compile the correct Java files from the Protocol Buffer file created by Severinsen, due to some compiler issues which were eventually rectified by recompiling Google's example gRPC files whilst including Severinsen's protocol buffer files. When the correct files were created, the Java client was set up and the data was finally being transmitted, it was unfortunately discovered that the actual decryption process was not working. This was rectified when it was found that Java and Go treat OAEP padding of RSA slightly differently resulting in compatibility issues meaning that we would have to settle with plain RSA with no padding for the final product.

7 Strengths and Limitations

The main strength is that the core functionality needed by the system for a minimal viable product is present as a user can request the decryption of a file and cryptographic proofs are generated resulting in the files being successfully decrypted. The log can also be inspected by any user at any time which are the key features that were in the initial plan. However, it should be noted that this project is a proof of concept rather than a fully developed system, meaning

that there are some features that could be added in the future including the completion of the secure device. Some of these features are mentioned below followed by an analysis of the security of the system and steps that could be taken to improve it.

7.1 Features

On the webpage when an administrative user requests the decryption of a file, they are asked to provide some justification for this request, however an answer is not mandatory - if the field is empty, the justification will simply be sent as an empty string. The idea behind the justification for a request is so that a normal user who is inspecting the log can see why their files have been decrypted. However, if this field is left blank, it offers no information to the owner of the file. It is easy to make the justification field a requirement, however it is difficult to enforce obtaining a meaningful response; the only way this could be achieved would be to provide strict guidelines for decryption requests.

The Proofs page, mentioned in section 4.4, is not required for the core functionality of the system due to the log maintainer generating proofs automatically. However, this section was added as a way of verifying that the proof generation algorithms are working as expected. The proofs are generated so that the log maintainer does not need to be trusted (mentioned in sections 3.5 and 4.3). As such, if the user does not trust the service provider and without access to these algorithms, they would have no way of verifying that these proofs are being generated and forwarded to the device. In this case, access to the proof generation algorithms allows for a user to ensure the service provider is giving information that is consistent with the proofs that are to be generated.

The Inspection section of the front-end site is limited to displaying a list of the decryptions that have been requested. Whilst this is enough for a user to examine the log, it is not user-friendly, particularly if many decryptions of a user's files have been requested. Future implementations of this system could incorporate a graphical display of decryption requests, however due to time restrictions, this could not be applied within this system. This could be displayed as monthly or yearly histograms, showing how often files had been decrypted. This could include the ability to hover over each histogram bin to display information on an individual request, such as who made the request and why. This feature would not add any real functionality to the system, but it would make the system easier to use.

7.2 Security Analysis

Severinsen developed a prototype SGX server to perform the decryptions that were necessary to demonstrate this system. However, due to time limitations, there are still no mechanisms that can verify the proofs sent to the secure device in the current prototype. This means that anyone could send a request to the device without valid proof and the device would still decrypt the message, allowing someone to access a decrypted file without adding their request correctly

into the log. In this instance, the lack of verification undermines the core goal of the system, however this could be implemented with more time.

Currently, the secure device is not running on an Intel SGX chip, however it does emulate most of its properties. Without this chip, the device cannot be considered "secure" as the information on the device can be accessed by other processes running on the same machine (see section 3.4 for more details). An insecure device should be run on a trusted machine, however even a trusted machine can be exploited by malicious users; these users can take advantage of the fact that decryption keys are not in a secure location, and can attempt to extract them with various methods (e.g. implanting viruses).

The initial cryptosystem intended was RSA with OAEP padding to make the system more secure, however compatibility issues between the device and the web server made this impractical (see section 6.3). However, with the focus of this project being the process of making decryption accountable rather than the decryption itself, a plain RSA has been used in the final system despite not being considered secure as stated in section 3.1. For future iterations of the system, plain RSA should be avoided in favour of a more secure cryptosystem, such as one using OAEP padding.

For certain pages in the front-end website (e.g. Inspection page, Decryption Basket page), a user must manually enter their username before they can inspect the log, request a decryption or view decrypted files. Manually entering a username means that a single user could input incorrect details to avoid accountability. With regards to administrative users, if they can input incorrect details, their decryptions requests, their name and company details would not be added to the log correctly, thus escaping accountability. If an administrative user can decrypt files without their name and information being stored in the log, this makes the decryption process unaccountable, meaning that the system has been functionality undermined. This could be avoided by creating company-issued accounts whereby all users, (i.e. normal vs administrative) must log in with their verified allotted credentials before they can access the system, meaning that no external user can have an account without being company-verified. Furthermore, the current site encompasses both normal and administrative users, meaning that a normal user could attempt to decrypt a file (which should be limited to administrative users only). A company assigned login would designate a user into either a 'normal' or 'administrative' account, meaning that the functionality of the site would be restricted to the respective sections allowed by the account type.

The secure device, the service provider and the log could theoretically be in separate geographical locations, meaning that data transmission within the system would be via the internet. The communication of non-sensitive information does not pose a security threat, however for sensitive information such as plaintext transmission, there is a security issue. For example, after a decryption request has been made from the website to the secure device via the web server, the device must send the plaintext record through the network. At present, the transmission of this information is clear text and can be read by an eavesdropper on the network. As this project is closer to a prototype than a

readily deployable system, this issue was not a primary concern, however if this was to be deployed, encryption methods such as TSL or SSL would be needed to ensure secure communication.

8 Conclusion

The aim of this project was to investigate if it was possible and effective to build a system that makes decryption accountable using blockchain and Intel Software Guard Extensions. While there are still features missing from the system that would make it deployable in the real would, it has been an effective proof that the concept can be developed into a viable product. The features that create the minimum viable product have been created as well as extra features, including an attractive user interface and proof generation access which have been robustly implemented. The log maintainer successfully generates cryptographic proofs from a request, which are sent to the secure device which then decrypts the files and sends back the plaintext. The log can also be inspected by anyone, therefore ensuring accountability for the decryption requests. Whilst the system in its current state has several limitations mentioned in section 7, it has the potential to be an effective cryptographic protocol that could be implemented in various situations from employee email accounts to government data collection.

A Proof Output Files

Below are the output from 3 proof genertion algorithms used in this project. Many of the SHA-256 hashes in the proofs have been shortened for display purposes.

A.1 Audit Proof Output


```
{
    "RTH":"e9cd3285af60af3230947e8b...f78a837b65f8c18413",
    "Value":"3",
    "Proof":{
        "Left":{
            "Left":{
                "Hash":"6b51d431df5d7f141cbec...61a3eefacbba918"
            },
            "Right":{
                "Left":{
                    "Leaf":"3"
                },
                "Right":{
                    "Hash":"4"
                }
```

```
            }
        },
        "Right":{
            "Hash":"5"
        }
    },
    "Index":2
}
```


A.2 Consistency Proof Output


```
{
    "OldRTH":"375c39afce841a66a1ee088f4...376a2a67fda0e5e7799cf5b",
    "OldProof":{
        "Left":{
            "Hash":"6b51d431df5d7f1eccf79e...f0b11661a3eefacbba918"
        },
        "Right":{
            "Hash":"3"
        }
    },
    "NewRTH":"5cd190530acea64c3277d3ab...dfb34cb2c5aa7f3d89dca7c5",
    "NewProof":{
        "Left":{
            "Left":{
                "Hash":"6b51d431dfececcf7...069f0b11661a3eefacbba918"
            },
            "Right":{
                "Left":{
                    "Hash":"3"
                },
                "Right":{
                    "Hash":"4"
                }
            }
        },
        "Right":{
            "Hash":"fdefe3e799c1f867e99b7...0fa162ed69d93ac74ffb"
        }
    }
}
```


A.3 Absence Proof Output


```
{
    "Leaves":[
```

```
        "e3b0c44298fc1c149afbf4c8996fb9...b934ca495991b7852b855",
        "f9c589bcc4925c39855568afe7fa42...d9003f3ea0a1f3ba4a62a",
        "7315bc51dc1378687ab12dbb31aaaf...04aa8e19099aef2523deb",
        "c8c1f83272a2711aef62f327c8142c...be7c4def23c5dd9698552",
        "ce92b36cee7ac99d9ee1d3f1e487b0...2cfd97fea2b60b3386b94",
        "e5bca4a9b4487e399d2fa9ae151573...041bb6032dc66087b0e07",
        "d9338e5895763d5073438a519ab107...a540cbb960d5ad1c6cadd",
        "602074ea49dc8b0aacc54063cef728...b179495063cf77fca8a0e",
        "77b30cf2f484bf74bb086e550a2101...d348bcf749ae6f7127aa7",
        "94a413ec03461f836473cadbf63244...7fe70c6711af214b0f9e6",
        "5bb92d7717dcba77f55c56963ad7be...1fb2a507a73e7b6ea59e7",
        "93fdd608a37744f4d4a8ac0bfce3e1...9c4282548c0d891d0c767",
        "55b7da7b5a09c64f3039eca5390658...4a527cda0b55405f1574e"
    ],
    "DeviceSig": "7bdb30c61c51b3241131f317196a54844636be519cff37c80
    3eb5c23c76b93a70c85b7925c162f9aac36428f2e6e1f7efe12a78ee6fd5ffb
    ...
    a83214facc7bf0eb25aea01493613f68c247fd3b89f3e82ab79acbe198 ",
    "RTHFromDevice":"8414efaddaf3e3553...1c13a1bac3604cfc484",
    "FileHashes":[
        "c8c1f83272a2711aef62f327c...d969855251e75460",
        "f9c589bcc4925c39855568afe...d9003f3eaa4afe2a",
        "e5bca4a9b4487e399d2fa9ae1...041bb60087b0e02d",
        "9c93fede6d22d9d6ae0ea6d74...793a90d3df53325f",
        "93fdd608a37744f4d4a8ac0bf...279c42841d0c7677",
        "94a413ec03461f836473cadbf...37711af2b09e623b",
        "056f5e9743b3fcc88e0a3b098...3319b87274593ffa",
        "602074ea49dc8b0aacc54063c...61f0b179c8a0e847"
    ]
}
```


B Running Instructions

- Set up the SGX secure device server
  - Install Go from https://golang.org/
  - Ensure environment variables GOROOT and GOPATH are set up correctly
  - Ensure sgx-decryption-service is located in the directory $\$GOPATH/src/github.com/sewelol/sgx-decryption-service$
  - Run the SGX server $\$go\ run\ server/main.go$
- Set up the web server
  - Ensure you can access the University of Birmingham's School of Computer Science database server $mod-fund-databases.cs.bham.ac.uk$.

This can be achieved by either connecting to the schools VPN or being on the wired network at the school

  - Check the Go server has loaded and presented you with its Initial RTH in the terminal
  - Run the server file located at *AccountableDecrpytion/src/main/java/JavaServer/JavaServer.java*

- Access the website
  - Open a web browser and go to the address *http* : *//localhost* : 8080
  - From here you can access all the systems functionality

C Key Terms

This appendix outlines some terms that may have not been explained for non-technical readers.

- RTH - Root Tree Hash, the value stored at the top of a Merkle Tree
- Nonce - A randomly generated "nonsense" string
- RPC - Remote Procedure Call, the process of executing some code or procedure on a remote machine
- JSON - Java Script Object Notation, a way of formatting data into nested objects. Is used in a similar way to XML
- SQL - Structured Query Language, used for accessing relational databases

D An Important Aside About GIT

During the development process of this project, login credentials to the schools GIT repository located at https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2016/dxf695.git were not working. For this reason, I used my own personal version control until mid-August. You will therefore be unable to see a full commit history on this repository. Supervisor of this project Mark Ryan can verify this and can confirm that weekly deadlines were being met throughout the summer.

References

[1] The Home Office. Draft investigatory powers bill, 2015.

[2] Ian Brown and Brian Gladman. The regulation of investigatory powers bill–technically inept: ineffective against criminals while undermining the privacy, safety and security of honest citizens and businesses. *undated. URL: http://www. fipr. org/rip/RIPcountermeasures. htm.*

[3] Mark D Ryan. Making decryption accountable. *Security Principles and Trust Hotspot 2017*, 2017.

[4] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[5] Burt Kaliski. Euler's totient function. In *Encyclopedia of Cryptography and Security*, pages 430–430. Springer, 2011.

[6] Dan Boneh, Antoine Joux, and Phong Q Nguyen. Why textbook elgamal and rsa encryption are insecure. In *ASIACRYPT*, volume 1976, pages 30–43. Springer, 2000.

[7] Dan Boneh. Simplified oaep for the rsa and rabin functions. In *Advances in Cryptology—CRYPTO 2001*, pages 275–291. Springer, 2001.

[8] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1.

[9] Brian Baldwin, Andrew Byrne, Liang Lu, Mark Hamilton, Neil Hanley, Maire O'Neill, and William P Marnane. Fpga implementations of the round two sha-3 candidates. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 400–407. IEEE, 2010.

[10] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[11] Shawn Wilkinson, Jim Lowry, and Tome Boshevski. Metadisk a blockchain-based decentralized file storage application. Technical report, Technical Report, Available: http://metadisk. org/metadisk. pdf, 2014.

[12] Guy Zyskind, Oz Nathan, et al. Decentralizing privacy: Using blockchain to protect personal data. In *Security and Privacy Workshops (SPW), 2015 IEEE*, pages 180–184. IEEE, 2015.

[13] Yaron Sella. On the computation-storage trade-offs of hash chain traversal. In *Computer Aided Verification*, pages 270–285. Springer, 2003.

[14] Michael Szydlo. Merkle tree traversal in log space and time. In *Eurocrypt*, volume 3027, pages 541–554. Springer, 2004.

[15] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. Technical report, 2013.

[16] Kristoffer Severinsen. Sgx decrpytion service. https://github.com/sewelol/sgx-decryption-service.

[17] Google. grpc open-source universal rpc framework. https://grpc.io/.