Cryptographic Cloud Storage Framework

by

Matthew D. Falk

S.B., Computer Science, M.I.T., 2012

S.B., Mathematics, M.I.T., 2012

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

June 2013

Author: ______________________________
Department of Electrical Engineering and Computer Science
May 24, 2013

Certified by: ______________________________
Prof. Nickolai Zeldovich, Associate Professor of EECS
Thesis Supervisor

Certified by: ______________________________
Dr. Gene Itkis, Technical Staff, Lincoln Laboratory
Thesis Co-Supervisor

Accepted by: ______________________________
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Cryptographic Cloud Storage Framework

by

Matthew D. Falk

Submitted to the Department of Electrical Engineering
and Computer Science

May 24, 2013

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer
Science

Abstract

The cloud prevents cheap and convenient ways to create shared remote repositories. One concern when creating systems that provide security is if the system will be able to remain secure when new attacks are developed. As tools and techniques for breaking security systems advance, new ideas are required to provide the security guarantees that may have been exploited. This project presents a framework which can handle the ever growing need for new security defenses. This thesis describes the Key Derivation Module that I have constructed, including many new Key Derivation Functions, that is used in our system.

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor of Electrical Engineering and Computer Science

Joint Work

This work is in joint collaboration with the Department of Defense Lincoln Laboratory Division 5 Group 8 team on Secure Cloud Computing consisting of: Project Manager Dr. Gene Itkis, Ninoshka Singh, John Darby Mitchell, Tim Meunier, Walter Bastow, Emily Shen, and Kendra Kratkiewicz.

Contents

List of Figures

1 Introduction

Engineers are always looking for faster, cheaper, and more reliable ways of getting things done. Processors get faster, chips get smaller, storage becomes redundant, etc. The emergence of the cloud in modern computing has had many benefits. One of the most promising features being that information can be stored remotely, ubiquitously accessible from any cloud connected device. One doesn't even need to be concerned with where or how that information is stored. It also presents a more cost effective storage medium and easier ways to share information. However, this also presents new concerns for security. Defenders need to ensure that all of the original security properties are maintained when using the cloud. While many security ideas and tools are portable to the cloud, some need to be reconfigured in order to fit the new medium. Further, information owners must trust cloud providers for all of their security.

We have built a system that will allow users to utilize the benefits provided by the cloud without having to worry about the security of their data. The trust required is minimized and users can select settings to safeguard their data to satisfy whatever level of paranoia they possess. As new attacks arise, the system will be able to incorporate new ideas that will protect against these attacks.

If you want to entrust information into the cloud, you need to protect it, which almost certainly includes confidentiality. Being one of the simplest types of security to achieve, required for practically all applications, and able to help with further types of security, confidentiality was a natural first focus. One of the primary goals of this project was to incorporate confidentiality protocols ensuring that any data placed on the cloud would be confidential. This needed to be done in a way that allowed other defenses to be easily implemented on top of the initial design. The thesis focuses on the derivation of keys used in the system.

Derived keys took most of the focus for this thesis as they are typically created from weak entropy sources. These are keys that need to be repeatedly creatable. That is, the proper user should have the ability to recreate these keys, without error, every time they attempt to create the key. They cannot be based on randomness as that will not allow the key to properly rederived. These keys must come from a source of entropy: what you know, what you are, what you have, etc. Entropy can come from passwords, biometric data, smart cards or tokens, etc. Obtaining and combining these entropies into a useful key must be difficult for an attacker to do. The goal of these key derivations is to produce a reliable and repeatable key from potentially noisy sources of entropy that is difficult for an attacker to brute force. Thus, the system employs new ways of obtaining these entropies, but this thesis focuses mostly on new Password Based Key Derivation Functions (PBKDFs) that increase the amount of effort needed to brute force the password compared to that needed to generate the correct password.

While keys are great for encrypting data, if those keys are stolen, then decryption becomes trivial. In order for data to remain confidential, it is equally important that the keys used to perform the encryption remain secure as well. The base system includes a module for key management which stores keys in encrypted form. It uses generated keys to access the stored keys.

This thesis begins with an overall description of the system in Section 2 followed by a description of the Threat Model in Section 3. Next, I provide the design criteria for Password Based Key Derivation, describe relevant implementations of PBKDFS, describe the designs I have built, and analyze them in comparison to current standards in Sections 4, 5, 6, and 7, respectively. Finally, I discuss the performance of the system in Section 8, provide a small discussion with future work in Section 9, and give some concluding remarks in Section 10. Acknowledgements are given in Section 11.

2 System Overview

Our system is intended to be very similar to DropBox with the possibility of more security features. The extensibility of the system will allow new security mechanisms to be implemented and become a part of the system very easily. The system achieves this by using objects, that we call *boxes*, which are passed around the system. Boxes consist of a list of *tags*, which are instructions for dealing with a particular box, and a *capsule*, which is the actual data contents of the object. Many different boxes can be constructed, each with a unique set of tags, which define the unique way of dealing with that box in particular.

Our system uses three types of keys, a *master key*, a *derived key*, and a *random key*. The master key provides the initial entry into the system, identifying a particular user. Derived keys, are based off of the master key and an additional piece of metadata, to be used for performing cryptographic functions for single users. As shared objects cannot be derived consistently with different entropy sources, we have random keys for shared operations that are all accessible via personally created derived keys.

As the master and derived keys, described above, define all of the entry accesses and identity of a user in our system, it is necessary that these processes of creating the keys be secure. Thus, we focus largely on the derivation of the keys from various entropy sources, mainly passwords.

3 Threat Model

In order for the Department of Defense (DoD) to use a commercial cloud service to store information, it is necessary that the security of all data be guaranteed. It is necessary that information not be accessible, in plaintext, to any party that does not have proper permission; no parties should be able to read or write any files that they were not granted access to. No unauthorized party should be able to tamper with the privileges of any data, unless they have permission to do so. Deleting data should be limited to those who own that data. Tampering with data that resides on the cloud should be detectable in the very least and avoidable if possible. Further, users should be protected against traffic analysis and other methods of obtaining information about a specific user from their access patterns.

The goals of the adversary can be summarized into three types: gaining access to a particular private data, deleting private data, and recovering a key. Each of these has a different associated risk. If an adversary is able to access a particular private data then that particular information is no longer secure. It becomes an untrusted file and must be assumed stolen. This does not lead to other files being compromised, but will cause unrest. Deleting private data does not reveal any sensitive information to the enemy, but it can be crippling to the DoD if large quantities of important data go missing. Finally, the most threatening goal is stealing a key. If a valid key is compromised, then any information protected by the key whether directly or indirectly becomes compromised along with it. This will be exponentially harder to contain as it does not correlate to a closed pool of data. The consequence of a stolen key is a direct access into the system in whole or in part, depending on the severity of the key.

We break the threats up into which assets they compromise. We go through the list of assets and describe briefly the threats that exist.

3.1 Human Assets

Specifically, these adversaries may be foreign intelligence agents. If placed within the network, their goal is to leak information, which they do or do not have access to, to their respective government. Foreign intelligence agents working with the Cloud Provider may opt to transfer all physical data overseas and allow their governments unlimited cipher text data to attempt to crack. If instead, they are malicious users without any special connections their main goal would be to uncover National Defense secrets that could be used to cripple our defense. Lastly, their aim could be to delete large amounts of data that would cause a major catastrophe for the government. Further, aside for foreign intelligence agents, or similarly terrorists, adversaries can be curious Americans. The goal of these individuals or organizations would not necessarily be to leak secrets out of the country and cause threats to our national security, but rather to find out secrets that the government for pure academic interest or because they feel the government is keeping things private that should be public knowledge. However, all of these parties pose the risk of exposing information that could harm the DoD if released and, thus, need to be protected against. Once any of these adversaries succeed, the jobs of the others are trivialized.

There are a few different types of adversary to the system. Any adversary is assumed to have access to the source code and is able to access the internet. Each type of adversary can be either passive or aggressive; that is curious or malicious. Curious adversaries simply wish to learn information that they did not know previously to satisfy their own interest, while malicious adversaries attempt to cripple the system in any way and actively seek private information for their own gains.

3.1.1 Attacker

As with any system, we can define the threat of a malicious party that does not have any special access (DoD contractor or Cloud Provider employee). This type of adversary falls into two categories: those that are also using the system and those not using the system. As the latter category is strictly more general than the former (and anyone can become a 'user' of the system), we will refer to any party without special privileges in this group and assume that they are a commercial cloud user.

This adversary is completely focused on gaining access to or disrupting information that they otherwise could not affect. This party will try to modify or delete contents that have been stored on the cloud. They will also attempt to obtain keys or data pertaining to any object. Here, the slightest leak of any information is considered a success.

Also, note, that this adversary has the least amount of tools at his fingertips, but may be a user of the commercial cloud. They may have access to stolen credentials and they could have a secure communication channel set up with the cloud provider.

A curious outsider is interested in accessing data that has not been released to the public. Their sole intent is to read and learn about things that they were not given access to. Typically, this type of adversary encompasses normal citizens and conspiracy theorists. Information leaks to this end reveal sensitive data to the public sector and make it, unintentionally, available to outside parties, which could be dire, if successful.

A malicious outsider has the goal of disrupting the state of the DoD through invalid data accesses or data corruptions. At their expense are possibly specialized hardware, stolen credentials, and false identifications. They will try anything from man in the middle attacks to stealing identities, etc. in order to achieve their goals. From a software perspective, these adversaries will attempt to perform things like phishing attacks to steal credentials. If successful, this adversary accesses private data that they can sell to the highest bidder or deletes and corrupts sensitive data, which increase the security alert of the DoD. Clearly, if this type of adversary yields successes, then any adversary can succeed and the data is not secure.

While these types of threats are very justified and pose large concerns, they are not the threats we aim to protect against through hardware and software. However, they are the individuals that will run the succeeding types of attacks, so they are still of much concern.

Typically, these individuals will be the type of adversary that we are protecting against. They will also be the ones to invoke the attacks described in the following sections. Next we present the security assets of the system and which threats we do aim to safeguard against.

3.2 Security Assets

Many of the assets belonging to the system pertain to specific Personal Identifying Information (PII), which provide the entry points into the system. Thus, if a malicious party is able to gain knowledge of one or more of these PIIs, they will be able to compromise some part of the system. Most of these assets are also subject to traffic analysis. While we do not specifically protect against these attackers, we do incorporate security measures to ensure that data is always protected when traveling among servers and machines.

3.2.1 Keys

Stealing the master key allows an attacker access to objects once a connection to the cloud has been established. This results in lost or corrupted data because an adversary can establish a secure connection to the cloud with valid credentials that they gained on their own and then use the stolen master key to access objects that they do not own or do not have permissions to. While stealing is a major threat of the system, it is not one that we particular focus upon. Keys are also vulnerable to a software attack; they can be cracked given enough computation. That is, through the use of *Brute Force Attacks*, *Dictionary Attacks*, *Rainbow Table attacks*, etc. the keys used in our system become potentially vulnerable. There are a few different types of keys used in our system: the master key, keys generated from the master key, and keys that are completely (pseudo) random. While no compromises are ideal, these each lead to different problems.

As mentioned above, cracking someone's master key, would allow you full access into there system. A systematic way of reproducing the master key allows you to unlock or derive all other keys associated to a particular user and grants you full access to everything they have permissions for. Leaking a generated key gives the attacker access to whatever that key can open. The key can not be used to access any higher level keys and therefore no higher level data, but everything that is generated or unlocked as a result of that key is compromised. Finally, cracking a random key simply allows the attacker to see a particular file or data object. This type of key is not used to generate any other keys, which makes them the least problematic, but consequently the least sought after. However, leaking any keys due to stealing or cracking leads directly to leaked private data.

We aim to protect against software attacks that lead to compromised keys through a key management module and new key derivation functions. This is one of the main threats that is addressed in this thesis.

3.2.2 Credentials

An adversary that steals a valid user's credentials would be able to instantiate a secure connection to the cloud provider and establish a communication; however, they would not be able to generate the corresponding user's master key. Therefore, although this adversary would have a connection to the cloud and (possibly) be able to view which data objects reside in the cloud, they will be unable to decrypt or view these objects in plaintext without further attacks at their disposal.

3.2.3 Entropy

This threat is the result of some party (adversary) copying or correctly mimicking the identity of a valid party. There are a few different pieces of identifying data that protect a user. In order, to successful gain access to the system as a different user, one would need to steal all of the entropy sources used to generate the user's master key.

Stealing one's entropy is equivalent to stealing one's master key because this is all of the input that goes into the function, which is public. Thus, this attack is just as deadly as the stolen master key. We do not aim to protect against social engineering attacks that lead to stolen entropy, but, like the keys, we do aim to avoid software attacks that lead to compromised entropies.

3.3 Software Assets

This next section of assets refers to those pieces of the system that must be built and public. These assets are subject to exposure, code injection, malware, viruses, etc. This is where the majority of the software attacks can come into play and we aim to minimize the chances of any of these being successful.

3.3.1 Framework Modules

The framework modules are pluggable pieces that provide different securities. As these can be written by outside parties, there needs to be some way of stopping malicious developers from integrating malware, malfunctioning code, and viruses into the system. We will have safeguards the validate the code that becomes part of a user's system to try and mitigate these attacks, but the majority of this will fall onto the user, and what sources they trust to get their security modules from.

These attacks occur when an adversarial developer writes code that is meant to do things other than those described. It may involve back doors that leak information or channels that reveal secret or protected data. If a user incorporates modules of code that are illegitimate, the adversary can completely control what the user does and doesn't see on top of what information does or does not get leaked.

Finally, we will write the code in a way that does not allow malicious modules to corrupt or steal data from other legitimate modules.

3.3.2 Source Code

An attacker could use the extensibility of the framework as grounds for an attack. An adversary could write code that performs harmfully and get users to inject this code into their program. If successful, this could grant the adversary access to all of that user's private files or anything that the user has access to. This attack would come from a developer who intends to be malicious. While code reviews and having multiple developers will hope to avoid this issues, this is left largely to background investigators to protect against this threat.

3.3.3 Files

Files are the most important asset of the system. Revealing private files is the worst type of breach that can occur. The files that are put into the system are intended to be private and the leaking of any of these could be catastrophic. Much of our system is thus focused on making these types of attacks impossible. Many security modules will be written to protect the data in different ways. Further, as attacks become stronger, new security modules can be written to protect against them.

3.4 Hardware Assets

These assets are the physical objects that make up our system. Much of this is the cloud servers that are actually hosting all of the encrypted files. Further, this includes any machine that is connected to the system with valid credentials.

3.4.1 Storage Servers

The storage servers, as per the cloud, are not stored in DoD facility, but instead are pushed to some outside provider. If we accept that the files are secure in software (a large assumption, but necssary to isolate particular threats). These servers are not protected by DoD level background checked employees. Thus, these employees could be malicious as described above. The employees could attempt to delete or alter the encrypted data or leak the encrypted data. We do not have any way of protecting against this threat; however, there will be redundancies built into the system in order to protect data that is subject to these attacks.

3.4.2 Local Machine

While slightly out of our control in physical structure, we can still employ securities that protect against compromised machines. If an attacker stole a machine and was able to extract PII or keys from it, that would be a huge security leak and minimize the breaking of the safety of our system to simply stealing a machine.

Thus, we have mechanisms in place that ensure that sensitive information is securely erased from any device connected to are system as soon as possible (such that it will not be frustrating to the user). This way we can help protect against the different attacks that might come into play on local devices.

4 Key Derivation Design Criteria

In order to ensure that users can repeatedly and consistently access the same objects when using the system, there needs to be a hard-to-generate string that will connect a user to their objects. This string also needs to be reasonably difficult for an attacker to guess. *Key Deriver*, consequently, aims at providing a repeatable and reliable mechanism that users can use to access their objects in a way that is not easily achievable for an attacker.

Entropy needs to be obtained, cleaned, and transformed in order to produce a key. While this project focused on multiples ways of achieving this: Biometric Data (what you are), Hardware Tokens or Keys (what you have), and Passwords (what you know), this thesis focuses largely on what you know, as password based keys are a large part of my contribution to this system.

A major concern with deriving keys from what you know, is that humans can "know" or remember relatively small amounts of data, or, more precisely, humans choose to. Remembering more than a few bytes proves difficult to convince the average human to do. When told to pick a password, users will choose the simpliest password they can come up with that fits any criteria placed in front of them, thus, deriving keys from what you know suffers from the possibility of an attack. A computer can be programmed to enumerate all possible *what you knows* and try each one or try and randomly guess the *what you know.* With this in mind, Password Based Key Derivation Functions aim to make all sorts of attacks difficult to execute, without increasing the work actual users need to do by proportionally as much.

Key Derivation Functions suffer from different types of attacks, the most popular noted below:

- **Brute Force Attack** - An attacker enumerates all of the possible passwords, acts with the KDF on each one, and compares the result to the one they are looking for.
- **Dictionary Attack** - An attacker only enumerates passwords that contain strings from a list of common words (i.e. more likely to be chosen as passwords). Then they perform a Brute Force Attack on this sub-list of passwords.
- **Rainbow Attack** - An attacker precomputes a *chain* that permutes the password space. This is acheived through a function, $\mathcal{R}$, that maps hashes back to strings in the password space (not to be confused with inverting). This function defines a random permutation through the password space.

$$\mathcal{P}_0 \rightarrow_{\mathcal{H}} \mathcal{H}_0 \rightarrow_{\mathcal{R}} \mathcal{P}_1 \rightarrow_{\mathcal{H}} \mathcal{H}_1 \rightarrow_{\mathcal{R}} \mathcal{P}_2 \rightarrow_{\mathcal{H}} \mathcal{H}_2 \rightarrow_{\mathcal{P}} \ldots \rightarrow \mathcal{P}_{n-1} \rightarrow_{\mathcal{H}} \mathcal{H}_{n-1} \rightarrow_{\mathcal{R}} P_n$$

This is done for a chosen n for a large number of random passwords. All of the $\{\mathcal{P}_0\}$ and $\{\mathcal{P}_n\}$ are stored as pairs in a *Rainbow Table.* When the attacker tries to crack a given hash value $\mathcal{H}_S$, they go through this process repeatedly reversing and hashing, until they get a $\mathcal{P}_n$ that is in the table. They then know that the correct password lies somewhere along the chain. Starting with the respective $\mathcal{P}_0$, they can navigate the chain until they determine the correct password.

Each of these attacks exploits some part of the system. Brute Force attacks exploit the fact that KDFs operate very quickly and can be done in rapid succession. Dictionary Attacks exploit the fact that humans typically choose weak passwords with low entropy. Rainbow Attacks exploit the fact that a naive KDF is 1 to 1 and always outputs the same key for the same input (of course, this has been combated with salts, but in some cases like will we see in Section 7.1 with LinkedIn [7], this would have been an effective attack). All of these methods allow an attacker to operate in parallel and try crack multiple passwords at once due to their 1 to 1 nature. Further, faster machines and CPUs, GPUs, cloud sourcing, and multiple machines allow attackers to perform these attacks at an accelerated pace, which is only improving with time. As space requirements decrease, rainbow tables and dictionary attacks can introduce more entries allowing the attacks again to become faster.

The battle is not lost; there is an equally long list of defenses that can be used to thwart these attacks. Most notably:

- **Slowness (T)** - KDFs can be made to run purposefully slow in order to increase the time it would take to launch a Brute Force Attack.
- **Password Space (Q)** - Having a larger password space increases the number of queries that an attacker needs to try; therefore, making brute force attacks much more difficult.

- **Password Complexity** - Although requiring passwords to conform to a specific criteria, makes Dictionary Attacks useless, complex passwords are hard to remember and most users tend not to use them. Thus, although this a valid defense, it becomes infeasible in practice.

- **Salts** - Salts are public index values that choose a particular hash function from a family of hash functions. Each of the different hash functions would require a unique rainbow table in order to launch a Rainbow Attack.

- **Network Interactions (N)** - Network interactions can be monitored to detect multiple access attempts. Including a network call in a KDF limits the attacker's ability to test multiple passwords.

- **Memory Usage (M)** - KDFs can be made to use large amounts of RAM in order to make attacking multiple passwords at once infeasible. Further, if a state needs to be remembered, large amounts of space will be needed to do it for large numbers of passwords.

- **Additional User Strings** - Requiring additional user strings effectively increases the password space, making all attacks less feasible, but also increases the difficulty for the user, and thus is not looked at in this thesis.

- **User Selection (U)** - KDFs can be made to provide users with options to choose from. This also, effectively, increases the password space, but at a reduced cost to the user. The user can use a mechanism or rule to select options, rather than have to memorize exact entropy.

Krawczyk provides the following definition of KDF security [4]; however, it does not take into account many of the resources described above.

Definition 4.1. *A key derivation function KDF is said to be (t, q, ϵ)-secure with respect to a source of key material Σ if no attacker $\mathcal{A}$ running in time t and making at most q queries can win the following distinguishing game with probability larger than $1/2 + \epsilon$:*

1. *The algorithm Σ is invoked to produce a pair σ, α.*

2. *A salt value r is chosen at random from the set of possible salt values defined by KDF (r may be set to a constant or a null value if so defined by the KDF).*

3. *The attacker $\mathcal{A}$ is provided with α and r.*

4. *For $i = 1, ..., q' \leq q$: $\mathcal{A}$ chooses arbitrary values c_i, l_i and receives the value of KDF(σ, r, c_i, l_i) (queries by $\mathcal{A}$ are adaptive, i.e., each query may depend on the responses to previous ones).*

5. *$\mathcal{A}$ chooses values c and l such that $c \notin \{c_1, ..., c_{q'}\}$.*

6. *A bit $b \in \{0, 1\}$ is chosen at random. If $b = 0$, $\mathcal{A}$ is provided with the output of KDF(σ, r, c, l), else $\mathcal{A}$ is given a random string of l bits.*

7. *Step 4 is repeated for up to $q - q'$ queries (subject to the restriction $c_i \neq c$).*

8. *$\mathcal{A}$ outputs a bit $b' \in \{0, 1\}$. It wins if $b' = b$.*

We can use a new definition of security for a similarly described game:

Definition 4.2. *A key derivation function KDF is said to be ($\mathcal{T}$, $\mathcal{M}$, $\mathcal{Q}$, $\mathcal{U}$, $\mathcal{N}$, ϵ)-secure with respect to a source of key material Σ if no attacker $\mathcal{A}$ running in time $\mathcal{T}$ and memory $\mathcal{M}$, making at most $\mathcal{Q} * \mathcal{U}$ queries and $\mathcal{N}$ network calls can win the following distinguishing game with probability larger than $1/2 + \epsilon$:*

In the following section, we will give each of the PBKDFs variable metric values that we will use to compare all of our functions to, where '-' denotes an irrelevant factor. The KDFs presented here aim to introduce the defenses in new ways to provide additional difficulty for an attacker without effecting the user to a similar degree. Next, I present current accepted password based key derivation functions and describe which defenses they employ. I then describe the new PBKDFs that were implemented in this project and then analyze each of them in accordance with the above metric.

5 Key Derivation Background

This section describes current standards and new implementations in Password Based Key Derivation Functions (PBKDFs). These deal with converting "what the user knows" (i.e. passwords) into a random string of variable length. The goal of the key derivation attempts is to separate the space between difficulty for the user and difficult for someone trying to brute force a user's key. It is preferable that the cost for the brute force attacker be driven up, while the cost for the user is not increased by the same amount. While, this is an inherent limitation in KDFs, the main effort of a KDF is to maximize the efforts of the attacker within the confines of this limitation. Below are some PBKDFs, the first being a major component for the functions described later, as most of them are an adaptation of PBKDF2.

5.1 PBKDF2

This is the current accepted standard in cryptography for key derivation. The function declaration appears below. It is useful to note that the only private parameter is the Password (PW); the Salt (S) and the number of Rounds (R) are public.

```
MK = PBKDF2(PW, S, R)
```

where the function is defined as:

$$MK = \mathop{\Big|\Big|}_{k=1}^{L} \bigoplus_{i=1}^{R} U_i(k)$$

$$U_i(k) = PRF(PW, U_{i-1}(k)) \text{ with } U_0(k) = S||k$$

This is a more condensed version of the algorithm that appears in [3]. PBKDF2 uses a salt so it can defend against rainbow attacks. Further, the variable number of rounds allows the user to choose their level of security. That is, the user can satisfy their own paranoia. Choosing a very large round number will effectively divert brute force attacks, but make the function more frustrating for the user, having to sit through all of those iterations whenever they derive their key.

According to the metric above, PBKDF2 is ($\mathcal{T}_P$, -, $\mathcal{Q}_P$, -, -, ϵ_P)-secure, where the subscript p denotes PBKDF2.

5.2 Scrypt

This is a PBKDF that aims to increase the amount of space required to compute a key from a password [5]. It relies on certain parts of a pseudorandom string that must be stored in core memory while the function computes. Thus, the goal being that it is infeasible to perform many attacks in parallel. However, the string can also be computed on the fly, but takes considerable more time as it is supposed to be computationally expensive.

This PBKDF introduces a *time vs. memory* tradeoff. An attacker can choose to try multiple derivations in parallel, but they will take considerable more time. It introduces the memory factor as another resource, which makes all of the attacks mentioned above perform more slowly, due to computations on larger data sizes. However, it also helps against GPUs because it uses up the RAM and does not allow for GPUs to perform as effectively as they do on other PBKDFs.

According to the metric above, Scrypt is ($\mathcal{T}_S$, $\mathcal{M}_S$, $\mathcal{Q}_S$, -, -, ϵ_S)-secure, where the subscript S denotes Scrypt.

5.3 Bcrypt

Bcrypt is another PBKDF that employs salts to protect against rainbow attacks [8]. Like PBKDF2, it also allows for a variable number of rounds so that as technology advances, the number of rounds can be increased to provide continued protection against brute force attacks. Bcrypt has an expensive start cost when it creates the initial state (an internally required value), but then simply performs Blowfish encryption in electronic code book mode over and over for each round. Using salts and increasing the number of rounds of Blowfish in ECB allows the PBKDF to remain resilient against rainbow attacks and brute force attacks, even as computation power increases.

According to the metric above, Bcrypt is ($\mathcal{T}_B$, -, $\mathcal{Q}_B$, -, -, ϵ_B)-secure, where the subscript B denotes Bcrypt.

6 Key Derivation Design

Here is presented the technical description of a few different Password Based Key Derivation Functions that have been implemented into our system. All of the following Key Derivation Functions require a Pseudorandom Function (PRF) and have a hard coded output length (L). We use a hard coded output length in our system, but the KDFs are not actually subject to this limitation and can work for variable size. We use length, to mean the multiple of PRF-Output-Length. That is, if the chosen PRF has an output length of 64 bits, and we choose $L = 8$, then we are actually coding a KDF that outputs a Master Key (MK) of length 512 bits. The definition of $U_i(k)$ in Section 5.1 will be used in subsequent explanations of key derivation functions, as they are adaptations of PBKDF2.

6.1 Terminate Full Match Key Derivation Function

This is a slight variation of PBKDF2; the number of rounds, (R), is not passed into the function after the initial generation process has occurred. As a result, R is also not stored publicly. The difference is due to a different stored value. In addition to the Salt (S), a positive integer (I), (strictly less than R) and Match String (MS) are stored publicly. In this function, the first time it is used, R needs to be supplied, but it can be forgotten afterwards as it is never needed again. The very first time the function is called is uses the declaration:

```
MK = TFMKDF(PW, S, I, R)
```

where the function is defined as:

$$MK = \mathop{\Big|\Big|}_{k=1}^{L} \bigoplus_{i=1}^{R-I} U_i(k)$$

with MS defined as follows, and stored for future function calls.

$$MS = \mathop{\Big|\Big|}_{k=1}^{L} \bigoplus_{i=1}^{R} U_i(k)$$

Once the initial call has occurred, the subsequent declaration appears as (Note: I and MS are stored publicly).

```
MK = TFMKDF(PW, S, I, MS)
```

In order to determine MK, the function begins looping until MS is reached, then the function rolls back I rounds and produces MK. The roll back is simulated by keeping the last I values in memory in a FIFO stack.

The advantage of this function is that an attacker will not know how many rounds to run the function for even if they guess the correct password. That is, if an incorrect password is provided the looping will never cease because MS will never be reached (in a feasible amount of time). The attacker will have to make a decision to stop the attack if they think they have been running long enough. In essence, this PBKDF's advantage is that it never provides verification that a password is incorrect, only that a password is correct.

6.2 Terminate Partial Match Key Derivation Function

This KDF, like its name, is very similar to the previous one. Firstly, MS and R are no longer needed. Secondly, the new addition is a public Match Function, f. f is a function that determines if a given string passes a certain (user-defined) condition. The main caveat of this function is that there are multiple possible strings that will pass (i.e. the string must end in the bits 001001). The function's declaration is as follows:

```
MK = TPMKDF(PW, S, f, I)
```

where MK is defined:

$$MK = \mathop{\Big|\Big|}_{k=1}^{L} \bigoplus_{i=1}^{j-I} U_i(k)$$

with j defined as follows:

$$X_j = \mathop{\Big|\Big|}_{k=1}^{L} \bigoplus_{i=1}^{j} U_i(k) \text{ s.t. } \min_j f(X_j) = True$$

In order to determine MK, the function begins looping until a string X_j passes the match function f. Then, the system rolls back I rounds, as before using a FIFO stack to get MK.

The benefit of the KDF over the previous implementation is that there is no longer a verification of correct passwords; however, there is now a terminating condition of incorrect passwords. Every password eventually leads to a string that passes f, whether it is the correct password or not. Therefore, the attacker will generate *a* master key every time, but it may not be correct. Thus, they will actually have to test the master key on some cipher text and see that the decryption is garbage before they discount the password as incorrect. This is a severe time hit for the attacker. It is useful to note that the user (with the correct password) is unaffected by this change.

6.3 User Selection Key Derivation Function

This is another variant of PBKDF2; this function introduces another source of user input in addition to the password. The initial part of the declaration is as follows, with n being a large random number and stored publicly for future uses:

```
MK = USKDF(PW, S, n)
```

The function begins by looping and creating $\{X_0, X_1, ...X_n\}$:

$$X_j = \Big\|_{k=1}^{L} \bigoplus_{i=1}^{j} U_i(k)$$

The $\{X_j\}$ are displayed to the user, who selects a number, m, of them. We label the chosen strings as $\{Y_0, Y_1, ...Y_m\}$. Then the master key can be defined:

$$MK = PBKDF2(\bigoplus_{i=0}^{m} Y_i, S, n)$$

While this may seem very complicated for the user to have to remember all of the different $\{Y_i\}$, the user can choose simpler ways of remembering their selection. Let us consider how many bits of entropy result from this method (aside from choosing a password). There are n total strings to choose from and the user selects m of them. Thus, there are a total of:

$$\binom{n}{m}$$

combinations, which results in:

$$\log_2 \binom{n}{m}$$

bits of entropy. If we choose appropriate numbers of $n = 1024$ and $m = 8$ we obtain:

$$\log_2 \binom{1042}{8} = 64.86 \approx 65$$

bits of entropy. Naturally, this would require the user to remember 65 bits of entropy; however, due to the nature of the strings, there is a way for the user to repeatedly choose the same strings without having to remember them individually. The strings are hexadecimal strings of length 64 characters. For example the following is a sample output of $\{X_i\}$.

```
1788609810BC9B342C2D2D59F8DB7635BFBB6384359EF242AC1A8F5BBD633A0E

3CEB77A83114649547A495C5680F275F212318E94A1EA02651571B202D54ECB5

1837D6D678E4E8F1ACA7902ADE4CDCF5AB47772A56722E4297EE4574A072DFAA

56495E99653441140A7FC95EB710D120447618E5E412F8A89155C281B1CA1944

3F85C337FCE39DA5D9FBF41125D74C970FE40665A5C2C0E245296813082CA6BC

...
```

As there are only 16 choices for any particular character, there will be a lot of repetition among the given strings. Thus, the user can choose a simple mechanism for remember a given selection of strings, such that the entropy required to remember that string is minimal, or rather much less than 65 bits. For example, the user can choose the following mechanisms:

- The first 5 hashes beginning with the letter E.
- Every 5^{th} hash from the start, for 10 hashes.
- The last 15 hashes.

- Every other hash that ends in C.
- Hashes containing the hexadecimal sequence "BAD".
- The first 7 hashes that begin with a 7 after number 777.

Each of these mechanisms require the user to remember information, but not a random string of bytes. The typical mechanism would require memorizing roughly 10-20 bits. Each value in hexadecimal has 4 bits of entropy, plus the entropy needed for the rest of the mechanism, typically a 4 bit number and some other information. This is much less than is actually provided by the key derivation function.

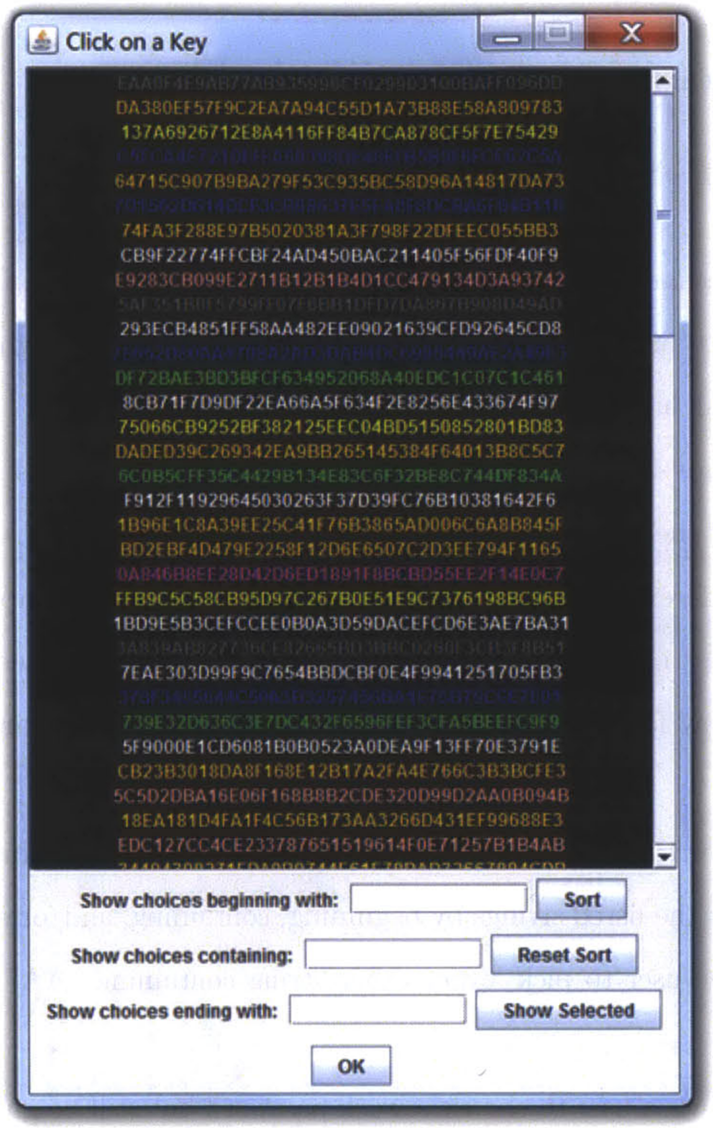
There are two different versions of this Key Deriver that we have implemented. The first color codes different strings based on a matching condition, while the second does not. This simply serves as a way for the user to make more mechanisms. Both are featured below in Figure 1 and Figure 2.

The buttons below the scroll menu provides ways for the user to reduce the search space. They are able to sort the listed strings by beginning, containing, and/or ending strings. This makes it easy for the user to pick 'every other string containing "AA"' for instance, as is started in Figure 3.

This function allows a user to choose a week password but still be guaranteed of security by the rule that they are required to choose.

6.4 Trap Door Key Derivation Function

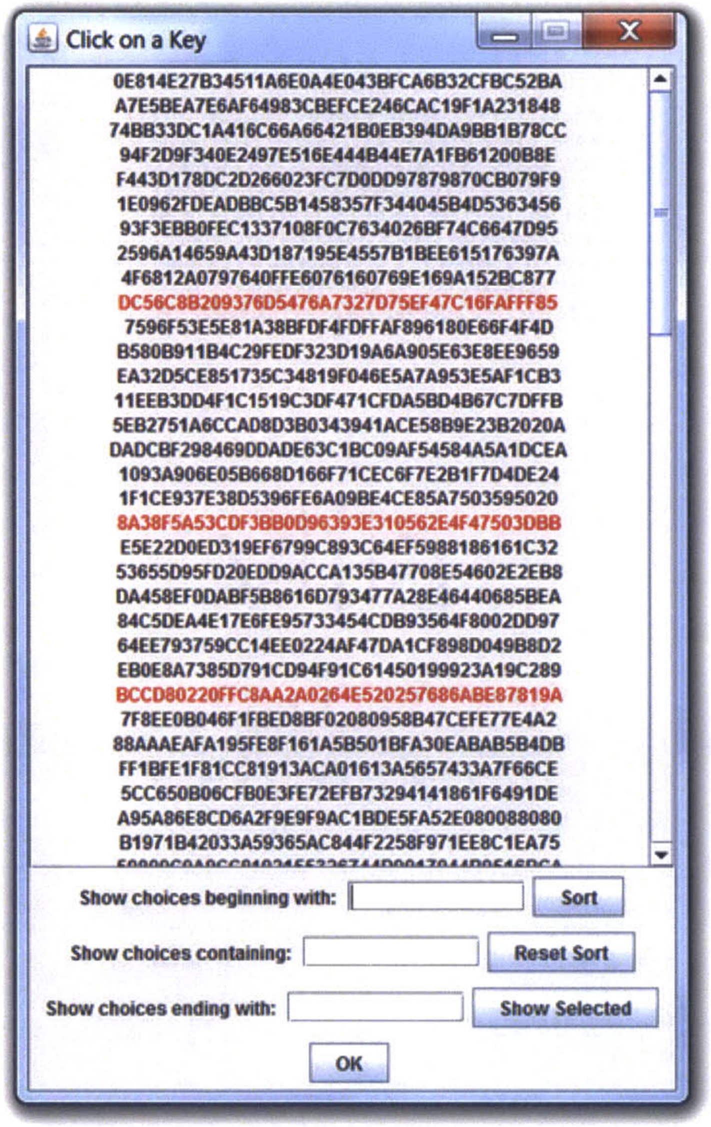
The key derivation function is used to alert a pre-decided party of all distressed access attempts. Similar to a field agent, every user has multiple login passwords. That is, all of the passwords can generate the same key; however, certain instances will trigger a red flag and alert the desired party of the distressed login.



```
EAA0F4F9AB77AB935996EF0299D3100BAFF096DD
DA380EF57F9C2EA7A94C55D1A73B88E58A809783
137A6926712E8A4116FF84B7CA878CF5F7E75429
[illegible]
64715C907B9BA279F53C935BC58D96A14817DA73
[illegible]
74FA3F288E97B5020381A3F798F22DFEEC055BB3
CB9F22774FFCBF24AD450BAC211405F56FDF40F9
E9283CB099E2711B12B1B4D1CC479134D3A93742
5AF351B8E5799FE07E6BB1DFD7DA897B908D49AD
293ECB4851FF58AA482EE09021639CFD92645CD8
[illegible]
DF72BAE3BD3BFCF634952068A40EDC1C07C1C461
8CB71F7D9DF22EA66A5F634F2E8256E433674F97
75066CB9252BF382125EEC04BD5150852801BD83
DADED39C269342EA9BB265145384F64013B8C5C7
6C0B5CFF35C4429B134E83C6F32BE8C744DF834A
F912F11929645030263F37D39FC76B10381642F6
1B96E1C8A39EE25C41F76B3865AD006C6A8B845F
BD2EBF4D479E2258F12D6E6507C2D3EE794F1165
0A846B8EE28D42D6ED1891F8BCBD55EE2F14E0C7
FFB9C5C58CB95D97C267B0E51E9C7376198BC96B
1BD9E5B3CEFCCEE0B0A3D59DACEFCD6E3AE7BA31
[illegible]
7EAE303D99F9C7654BBDCBF0E4F9941251705FB3
[illegible]
739E32D636C3E7DC432F6596FEF3CFA5BEEFC9F9
5F9000E1CD6081B0B0523A0DEA9F13FF70E3791E
CB23B3018DA8F168E12B17A2FA4E766C3B3BCFE3
5C5D2DBA16E06F168B8B2CDE320D99D2AA0B094B
18EA181D4FA1F4C56B173AA3266D431EF99688E3
EDC127CC4CE233787651519614F0E71257B1B4AB
```

Figure 1: This version of the User Selection Key Deriver features color coded strings to allow more mechanism choices for the user.

This method of key derivation allows the user to set up good and bad passwords. That is, all passwords will lead to the correct key being generated, but some of the passwords will cause traps to be triggered. A user can be notified when someone tries to break into their system or further a user can log into a system while at the same time revealing information about their situation (whether they are under duress or not). This system requires a trusted

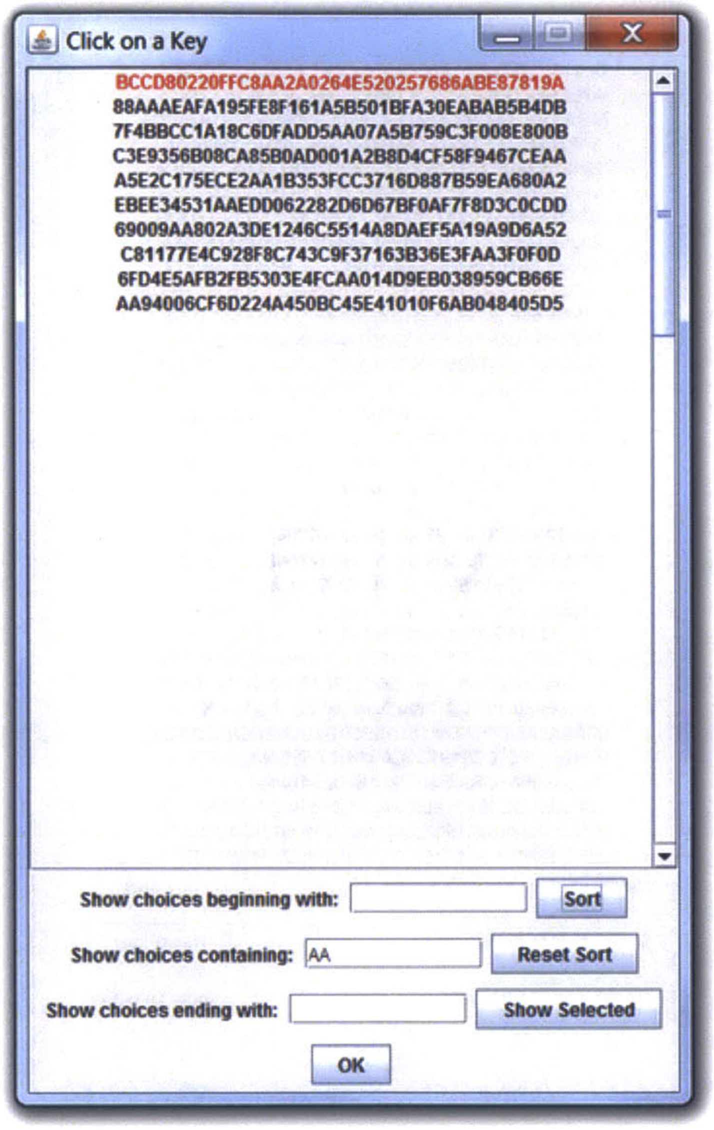


```
0E814E27B34511A6E0A4E043BFCA6B32CFBC52BA
A7E5BEA7E6AF64983CBEFCE246CAC19F1A231848
74BB33DC1A416C66A66421B0EB394DA9BB1B78CC
94F2D9F340E2497E516E444B44E7A1FB61200B8E
F443D178DC2D266023FC7D0DD97879870CB079F9
1E0962FDEADBBC5B1458357F344045B4D5363456
93F3EBB0FEC1337108F0C7634026BF74C6647D95
2596A14659A43D187195E4557B1BEE615176397A
4F6812A0797640FFE6076160769E169A152BC877
DC56C8B209376D5476A7327D75EF47C16FAFFF85
7596F53E5E81A38BFDF4FDFFAF896180E66F4F4D
B580B911B4C29FEDF323D19A6A905E63E8EE9659
EA32D5CE851735C34819F046E5A7A953E5AF1CB3
11EEB3DD4F1C1519C3DF471CFDA5BD4B67C7DFFB
5EB2751A6CCAD8D3B0343941ACE58B9E23B2020A
DADCBF298469DDADE63C1BC09AF54584A5A1DCEA
1093A906E05B668D166F71CEC6F7E2B1F7D4DE24
1F1CE937E38D5396FE6A09BE4CE85A7503595020
8A38F5A53CDF3BB0D96393E310562E4F47503DBB
E5E22D0ED319EF6799C893C64EF5988186161C32
53655D95FD20EDD9ACCA135B47708E54602E2EB8
DA458EF0DABF5B8616D793477A28E46440685BEA
84C5DEA4E17E6FE95733454CDB93564F8002DD97
64EE793759CC14EE0224AF47DA1CF898D049B8D2
EB0E8A7385D791CD94F91C61450199923A19C289
BCCD80220FFC8AA2A0264E520257686ABE87819A
7F8EE0B046F1FBED8BF02080958B47CEFE77E4A2
88AAAEAFA195FE8F161A5B501BFA30EABAB5B4DB
FF1BFE1F81CC81913ACA01613A5657433A7F66CE
5CC650B06CFB0E3FE72EFB73294141861F6491DE
A95A86E8CD6A2F9E9F9AC1BDE5FA52E080088080
B1971B42033A59365AC844F2258F971EE8C1EA75
```

Figure 2: This version of the User Selection Key Deriver features no colors, but gives off a cleaner and easier to read style. As you click on specific keys, they turn to red. This indicates a selection that you have made. This action is easily undone, by simple reclicking the string.

server to compute the final key. The user's system makes requests to the trusted server.

The system again requires the user to enter a password, P, and a round count, R, but the user must remember the round count this time. Similar to the previous method, the



```
BCCD80220FFC8AA2A0264E520257686ABE87819A
88AAAEAFA195FE8F161A5B501BFA30EABAB5B4DB
7F4BBCC1A18C6DFADD5AA07A5B759C3F008E800B
C3E9356B08CA85B0AD001A2B8D4CF58F9467CEAA
A5E2C175ECE2AA1B353FCC3716D887B59EA680A2
EBEE34531AAEDD062282D6D67BF0AF7F8D3C0CDD
69009AA802A3DE1246C5514A8DAEF5A19A9D6A52
C81177E4C928F8C743C9F37163B36E3FAA3F0F0D
6FD4E5AFB2FB5303E4FCAA014D9EB038959CB66E
AA94006CF6D224A450BC45E41010F6AB048405D5
```

Show choices containing: AA

Figure 3: This state of the application represents after a user has sorted the results based on certain criteria In this case, the criteria is "contains the sequence of character 'AA'". Only those strings that match are presented, and the user is free to choose whichever of them they like, according to their own mechanism.

system generates the SK using $H_{R,R-1}$, but this time it also generates a trap key, TK, using $H_{R-100,R-100}$. However, the process is a little different.

$$H_{R,0} = Hash(P||R)$$

$$H_{R,i} = Hash(H_{i-1}||R)$$

$$SK = H_{R,R-1}$$

$$PK = Hash(H_{R,R-1}||R)$$

$$TK = H_{R-100,R-101}$$

The generation of SK, PK, and TK takes only a constant factor times that of the previous system. However, decryption *can* be made to be much slower.

When the user supplies a P^* and R^*, the system begins by generating $H_{R^*,R^*}, H_{R^*+1,R^*+1}, \dots$ where $H_{x,0} = Hash(P^*||x)$. The system continues until it reaches the SK. That is, the system halts when $H_{R^*+n,R^*+n} = SK$. Again, if $P^* \neq P$, this will never happen. The user sends each of the $H_{i,i}$'s to the trusted server to check for verification. Whenever a string is verified, the previous string is the key. This is to ensure that the actual key is never passed in plaintext. However, if the Trap Key is hit a red flag will be triggered. The major downside of this function is that it requires to an external sever, which not only hinders performance, but leads to another entity that can be compromised.

The correct user can supply the correct P and R and generate the key very quickly. However, if they forget R, then set $R = 0$ which will generate the right key, eventually. Then they can just ignore the red flag.

This method can be extended to make round count a character at the end of the password, so password and round count are just a combined value. Further, if the server ends up compromised, this reduces to the previous method, with a longer generation time.

A user can choose to have as many trap keys as they desire with this system and they can be as close or as far from the actual key as desired.

6.5 History Key Derivation Function

This variant of PBKDF2 desires to increase the amount of space an attacker would need in order to launch attack on the system. The correct user would be able to run this KDF without any problem, as they only need to use the resources once; however a malicious user, attempting a parallel attack would need to have many times the resources as a single user, making it infeasible. The function declaration is the same as PBKDF2:

```
MK = HKDF(PW, S, R)
```

$$MK = \mathop{\Big|\Big|}_{k=1}^{L} \bigoplus_{i=1}^{R} U_i(k)$$

$$U_i(k) = PRF(PW, U_{i-1}(k)^{\otimes i-1} \oplus \mathop{\Big|\Big|}_{j=0}^{i-2} U_j(k))$$

with $U_0(k) = S||k$ as before. Note this is the same loop as in the original KDF but the value passed into the PRF has been expanded to include the entire history of the function. This has the advantage that the attacker cannot begin attacking a particular password at any given round without storing the concatenation of all previous $U_i(k)$, which will be computationally to expensive to mount a parallel attack. Notice that $U_i(k)$ needs to remembered for all i in this scheme, however, combining this scheme with indeterminate halt (TFMKDF) means that you need to remember $U_i(k)$ for all i for all k

The full function uses a combination of the details described above and terminate full or partial match. That is, the attacker still has to loop until a match is hit, but no longer can the attacker mount a parallel attack. The combined effort forces the attacker to perform a lot of repeated operations as they cannot "store the current state" of any single password attempt and return to it later. They must abandon all effort made on a single password (whether it was enough or not) and possibly return to mount the same attack again, but with a larger round count and starting from scratch.

6.6 Network Key Deriver

This PBKDF involves a network call that acts as a 1 to 1 function (i.e. a hash function with a private salt). It introduces an extra step at the beginning of the KDF. Instead of passing the password straight to the KDF, the password is sent to the network which computes the salted hash and returns the new passphrase. This is then used as the input to the hash family function for derivation. An attacker can continue to brute force this, but they will have to make many network calls, which is easily detectable and therefore defendable (just block a particular user trying multiple passwords). It is slower for a user because it has to contact the network each time a key is derived. An attacker can bypass the network call; however, they will no longer know the password space and have to brute force a much larger space to be sure that they contained the transformed space.

The network function acts a transformation function that takes all passwords to a pseudo-random string of fixed (longer) length. It is injective and the transformed space in unknown. It is known to be a proper subset of all strings of that fixed length (whatever the length may be), but it will be a very small subset. Thus, brute forcing at this step will require substantially more work for the attacker.

7 Key Derivation Analysis

The Key Derivation module creates and generates all of the keys used in the system. Again, this system does not hold onto any of the keys that it generates. When keys are needed, there is a request form KM to get the respective key from KD. All of the details needed to generate a particular key are passed in and the generated key is passed out. Occasionally, user input is needed to generate a certain key; this is all handled within KD.

This module contains the function for deriving the user's *masterKey*, which is the main key the user needs in order to access their data. This project allows for the choice among a variety of different KDF functions.

This section is devoted to discussing the complexity of the Key Derivation sub-module. This section discusses the complexity of KDFs and the level of defense they provide. This aspect of the project does not provide further defense strategies against things like password sniffers or key loggers, but rather focuses on defeating particular attack strategies that are typically employed by an **attacker** such as dictionary attacks, brute force attacks, and parallelized attacks.

All of our key derivation functions (KDFs) are based off of PBKDF2 [3] and thus the complexity will be derived from this. At the heart of these functions is the core algorithm of PBKDF2, no analysis of the security of the functions is provided (i.e. we do not analyze attempts to break PBKDF2 directly), but note that one can obtain the security guaranteed by PBKDF2, provided that they use an accepted pseudorandom function (PRF) and enough rounds.

Instead, this section will look at these KDFs in terms of the separation of efforts between **user** and **attacker**, which are defined below.

Definition 7.1. *A **user** is any party who holds the correct password and attempts to gain the key derived from the appropriate KDF, when the KDF is provided with the correct password.*

Definition 7.2. *An **attacker** is any party that does not hold any non-public information about the correct password, who is trying to obtain the key derived from the appropriate KDF, when the KDF is provided with the correct password.*

By analyzing the KDFs in terms of separation of the complexity for a **user** and the complexity for an **attacker** one can see how much harder the job of the **attacker** becomes, while the job of the **user** does not get any more difficult. Thus, things that increase the complexity of all parties equally will not become a part of our analysis. Things such as round count (R), which PRF is used, length of the password $|P|$, etc. which, in general, increase the complexity of the function for the **user** and the **attacker** by the same factor, are not considered. Rather, some of the functions will attempt to derail known attacks of KDFs such as those described above. Further, things that alter the use of parameters such as a varying round count, or indeterminate halting conditions will be used to increase the space between the two types of parties.

Typically, an **attacker** will attempt to brute force the password used to derive the key. They will enumerate the possible passwords and run the algorithm on each of them in turn to see if the correct key is generated. In order to speed up the process, this will likely be parallelized to test multiple passwords at once, using up more of the system resources.

Finally, we will discuss how each of the following uses the features described above and analyze them with the metric presented.

7.1 Existing Solutions

The recent invent of GPUs has spiked performance when attacking KDFs, but specifically hash functions. GPUs take advantage of the highly parallelizable nature of hash functions and KDFs providing speeds an order of magnitude faster [9]. Using a cluster of 25 GPUs optimized for these attacks, Gosney was able to try 63 billion ($\approx 2^{36}$) iterations of SHA1 per second. Similarly, he was able to make 71,000 ($\approx 2^{16}$) guesses against Bcrypt and 364,000 ($\approx 2^{18}$) guesses against SHA512crypt (another PBKDF) [1]. The largest bottleneck of PBKDF2 is the reliance on a PRF, if we assume SHA1 with 1 million rounds, then this machine will be able to make approximately 2^{16} guesses at PBKDF2 per second, the same as Bcrypt. (The round number was chosen conveniently to compare the two schemes together).

According to *Errata Security*, about a third of all passwords contain about 6 characters or less, which is about 36 bits of entropy (if we consider alphanumerics) [2]. This means that the average password can be cracked in less than 2^{36} guesses. Considering PBKDF2 and Bcrypt, this means that we can crack a password using an exhaustive search in $2^{36}/2^{16} = 2^{20}$ seconds or about 12 days.

As was stated by Gosney, all Windows passwords were able to be cracked in less than 6 hours [1] and according to [7], over 6.5 million LinkedIn passwords were cracked recently. Most of these passwords were very short or low on entropy. Thus, any KDF which adds entropy to the password will be helping these passwords that are easily crackable due to low entropy.

Further, Scrypt uses PBKDF2 as a subroutine to account for slowness and then uses sequential memory-hard 'mixing' functions to account for memory accesses [6]. This use of RAM means that GPUs are ineffective for speed ups because they cannot use the parallelization when doing random cache lookups. The amount of memory used in a parameter of the function and can be tuned to fit one's paranoia.

Asking users to choose stronger or longer passwords is a lost cause, thus we need to make stronger (more complex and slower) PBKDFs to help protect passwords. At the end of each analysis, I provide, if applicable, an analysis similar to that of the above function.

7.2 Terminate Full Match KDF

Terminate Full Match KDF (TFMKDF) removes the R from being public information. No one knows how many rounds the derivation process will take a priori. This means that if the incorrect password is provided the function may never stop running, unless killed by the party that ran the function, while the correct password will generate the key after $R_{correct}$ rounds. When an **attacker** attempts to brute force a key derived from this KDF, their task will be hardened because they do not know how long they are supposed to run each attempt for before moving on.

The usually parallel brute force attack may not work straightforwardly. The **attacker** will have to make choose a strategy to optimize his performance. The new attack will most likely attack all passwords by running the KDF for a set number of rounds, R_i, and then increasing that amount if the correct key has not been found. The optimal way to choose values of $\{R_i\}$ is to repeatedly double the previous round count. With this strategy, the **attacker** will never go twice as far as he needs to, but the optimal defense would be to pick an actual round number R that is equal to $2^k + 1$ for some k.

This means that the **attacker** will have to attack all passwords in the space, instead of half of them on average, and the **attacker** may need to go twice as far $2R$ rounds on each of the passwords. This leads to an overall increase of 4 times as much work for the attacker, without making the job of the **user** any more difficult or lengthy.

Proposition 7.3. *Any attack strategy $\mathcal{A}$ has an optimal defense strategy $\mathcal{D}$ that leads to a minimum effort increase factor of 4, where the increase is recognized in comparison to brute forcing PBKDF2 with a set number of rounds. If the* ***user*** *chooses a defensive strategy from the family $\{\mathcal{D}_i\}$ then the* ***attacker*** *will have to choose randomly from the attack strategies $\{\mathcal{A}_j\}$ to run the attack, leading to a possibly larger effort increase.*

It is useful to note that the change to this KDF means that whenever the **attacker** tries the KDF with the correct password, they are notified of it being correct, which is not desirable. This leads to the removal of the verification part of the derivation. If all parties are notified of a correct derivation then they do not have to perform the added step of confirming the derived key is correct or not. While this does not reduce the space between different parties' efforts, it does reduce the overall complexity of the function, which needs to be rectified.

By increasing the complexity of the *match* function, one can simulate the verification stage of the process and instead of reducing the overall complexity of the function, actually increase it, as parties would need to perform the verification step (test the *match* case) at every round. TFMKDF achieves a minimum of a factor of 4 in increasing the complexity space of good and bad users, making it ($\mathcal{T}_P$, -, $4\mathcal{Q}_P$, -, -, ϵ_P)-secure.

As described previously, the best attack method takes at least 4 times as long as PBKDF2 (with a set number of rounds). Therefore, if we use PBKDF2 with roughly a million rounds, as before, an attacker can only perform 2^{14} guesses per second and can finish an exhaustive search in $2^{36}/2^{14} = 2^{22}$ seconds or about 48 days.

7.3 Terminate Partial Match KDF

Much like the previous solution, Terminate Partial Match KDF (TPMKDF) does not have a set round number. In contrast, it does have a finite halting condition. The KDF will stop when a particular condition has been met, but that condition takes an inconsistent amount of time to run for different passwords. Instead of having a unique solution to the halting condition, this KDF employs a *match* case that accepts multiple answers. An attack strategy for this KDF would involve running each password until the halting condition is met, or using the attack described in the previous discussion for TFMKDF.

This KDF will have the same benefit as the previous example due to the inconsistency in round number, but it also has the added benefit that when the program does halt, it is not guaranteed to be correct. Much like the previous KDF, this allows the individual user to choose their level of paranoia. They can choose an incredibly sparse matching function.

Proposition 7.4. *If a* ***user*** *chooses a matching condition of strings that end with '0b01010101', then there will be many strings that pass this test (*≈ 1 *in every* $2^8 = 256$ *strings). However, if instead, the* ***user*** *chooses a matching condition of: strings that end with '0xDEADBEEF',* *then there will be much fewer strings that pass the test (≈ 1 in every* $2^{32} = 4,294,967,296$ *strings), which would take considerably longer to compute.*

As the matching condition is stored publicly, the **user** is not required to remember it, this does not lead to any extra effort by either party, comparably. However, it does increase the overall time of the KDF to a level that the **user** is comfortable with. With the goal of increasing the complexity between parties in mind, we can remove the match condition from being public information.

If it is required that the user supply the matching condition, the **attacker** will not attempt to guess the matching string, but follow the attack outlined in TFMKDF and check the key generated at the end of every round. While this would be a great increase in complexity for the **attacker**, the **user** would be required to remember a lot of new information, which increases the complexity of the attack for the **user**.

This KDF offers the trade-off of increasing the complexity of the **attacker**'s efforts severely at the expense of the **user** remembering more information. It also allows users to select their own level of paranoia (based on choice of matching condition) even if they choose to make it public. At the very least, this KDF is ($\mathcal{T}_P$, -, $\mathcal{Q}_P$, -, -, ϵ_P)-secure, with the potential of being ($2\mathcal{T}_P$, -, $\mathcal{Q}_P$, -, -, ϵ_P)-secure.

This KDF has attacks that perform comparably to those of PBKDF2, so an attacker can perform 2^{16} guesses per second and can finish an exhaustive search in $2^{36}/2^{16} = 2^{20}$ seconds or about 12 days.

7.4 User Selection KDF

User selection KDF (USKDF) requires the user to choose multiple selections from a list of pseudorandom strings. The benefit here comes from the fact that people can remember rules or mechanisms for selecting things, as opposed to exact strings, much more easily. While the

user has to remember a rule, which could take as little as a single byte of entropy, but will not take more than a few bytes, the **attacker** would have to attempt every possible combination of selections, which provides much more entropy. There will typically be ≈ 1000 strings to choose from and the user can select any number of them. Even with reductions to only select a few strings with an obvious pattern, the number of possibilities grows exponentially.

$$\log \sum_{k=1}^{10} \binom{1000}{k} = 78 \text{ bits of entropy}$$

This is roughly 10 bytes of entropy, which is more than the average password! Our application UI offers ways to search through the presented strings and narrow the presentation based on similar properties.

Proposition 7.5. *A mechanism for remembering which strings to choose (such as all strings containing "AA") require very little effort for a human to remember. This choice could lead to anywhere from 1 to 50+ strings matching. While the* ***user*** *must remember the simple mechanism, the* ***attacker*** *must enumerate all possible mechanisms, which will take considerable more effort.*

USKDF has increased the complexity of the efforts by the **attacker** by much more than it has for the **user**. This is very beneficial because it shows that by increasing the amount of work the **user** is required to by a small amount, it can increase the amount of work that the **attacker** has to do by $\mathcal{U}$, where this refers to the available options of choice by the user. Thus, this KDF is ($\mathcal{T}_P$, -, $\mathcal{Q}_P$, -, $\mathcal{U}$, ϵ_P)-secure.

The extra step required to combine the user's selections into a single result is a small cost when automated, so it does not introduce any speed change for the attacker, thus, the attacker can perform iterations of this as fast as PBKDF2; however, after the iterations have been performed, the attacker needs to enumerate and try every combination of choices. This effectively adds up to 78 bits of entropy to the user's password at the cost of them remembering a rule or mechanism that should not exceed a few bits of entropy.

This PBKDF is a huge improvement over existing algorithms because it effectively adds entropy that is more than double the typical password for a small user cost [2]. For instance, even if a user selects a rule leading to 25 bits of entropy, which would be very low, this still increases their total entropy to 50+ bits, when the password is incorporated. Using the number of guesses per second from before (2^{16}), at the very worst this leads to $2^{50}g/2^{16}gps = 2^{34}s$ = 500+ years to crack a weak password-rule combination. This is much more secure than the existing PBKDFs, especially for weak passwords.

7.5 History KDF

The History KDF (HKDF) attacks one of the resources of the **attacker** making it impossible to launch a parallelized attack. The idea being that if each run of the HKDF takes up a large portion of RAM then the user will not be able to do many of these at the same time because they will run out of RAM to use. Combining this with the strategy of TFMKDF, it ensures that whenever an **attacker** returns to an attempt that had previously been started, they will have to start from scratch. That is, they cannot save the state of an attempt to begin from later on, because storing them will take up too much space.

The **user** only ever has to perform a single attempt of HKDF at any given time, which means that, provided the memory cost is not too great, this action will go unnoticed by the **user**. Of course, this simply refers to space and not time. As the memory required in computing the function grows, the relative time to compute the function also grows. However, as this grows the same way for both the attacker and the user, it is of little consequence to us.

Proposition 7.6. *If a single run of the KDF uses memory* $= O(available\_memory)$ *there will not be an option of running a parallel attack. The best attack strategy would have to look at each password individually.*

Proposition 7.7. *Combining HKDF with TFMKDF eliminates the possibility of a parallel attack and it introduces a new trade off for the* **attacker**. *Any attack strategy must account for the non-halting of the KDF. A premature halt means potentially wasting computation that was already done and will need to be repeated if the key was not found, but not halting early means that you will be wasting time doing useless computation.*

This KDF succeeds in using up resources, other than time, that the **attacker** has to worry about, again meeting the goal of the function. This function offers the largest space between **user** and **attacker** efforts as the **user** will not have to worry about using the memory resources a single time and they will never go past the actual round count. This function runs just as PBKDF2 would from the perspective of the **user**, but the efforts of the attacker are made much worse due to the absence of a parallel attack strategy and the trade-off between throwing away computation and doing unnecessary computation.

This KDF is based off of PBKDF2 but it uses the memory like Scrypt, thus it is ($\mathcal{T}_P$, $\mathcal{R}$, $\mathcal{Q}_P$, -, -, ϵ_P)-secure. Here $\mathcal{R}$ represents the round number that was used, as the amount of memory, $\mathcal{M}$, required to derive the key using this method grows proportionally with the round count. That is, as the round increases, the amount of memory required also increases. Our scheme introduces a linear build up of memory, that is, the amount of memory depends linearly on the round count and the constant is dependent on the PRF used. Although an **attacker** could choose to store all of the information not in RAM, this would create a performance hit when trying to access that memory on every round.

An advancement to this function would be to make the ordering of the pieces random for each round. That is, have the ordering of the pieces be deterministic but not "in order" and unique for each seperate round. This will make the RAM accesses hard to parallelize, which will deter GPUs as they rely on parallelization. The function, as is, does not allow for an arbitrary amount of extra memory to be created, as in Scrypt [5]. However, it will be a great future direction to determine how to alter the function in order to insert a more variable amount of memory.

7.6 Network KDF

The Network KDF (NKDF) uses a different resource. It relies on network calls to gain information necessary to completing a function call. This information is unique to each input and is not computable, but a mapping that needs to be known. While this version of a KDF must also take into consideration the security of the outside server providing the information, it will be no less secure than any other KDF.

The final key is generated by computing the XOR (exclusive or) of the key derived by the KDF and the information retrieved from the server. Thus, if the **attacker** was able to get a hold of the information contained on the server, NKDF reduces to PBKDF2. However, assuming this were not the case, the network lookup allows for extra security measures to be accounted for. It can monitor access and defeat brute force attacks that are attempting to request many values from the server.

Also, the cost of a network call will be much slower than computing the function, but as this is again the same for the **user** and **attacker** it is ignored. This KDF does produce a successful separation in the amount of computation required by defeating brute force attacks, provided the server remains uncompromised. As stated previously, a compromise of the server leads directly to PBKDF2.

Proposition 7.8. *Including a network call in the KDF, while almost unnoticeable for the* ***user****, will deter any attacker that doesn't compromise the remote server, because access patterns will detect brute force attacks.*

This KDF doesn't strictly increase the efforts between the two parties, but it does offer a way to detect an attack strategy that involves brute force making it ($\mathcal{T}_\mathcal{P}$, -, $\mathcal{Q}_\mathcal{P}$, $\mathcal{U}$, -,

ϵ_P)-secure.

While this PBKDF is harder to analyze for a concrete time increase, it is clear that having the network call severely slows down the attacker's efforts making this much stronger than any existing solutions, but at the cost of requiring a network connection and the user to wait for a response. The server is allowed to do make restrictions on access attempts. Thus, if we limit the number of accesses by a particular IP address to 128 (2^7) guesses per hour (gph), then an attacker with a single machine would require $2^{36}g/2^7gph = 2^{29}h$ or 60,000 years to exhaustive search a 36 bit password compared to the 12 days of PBKDF2. Of course, an attacker could reduce this time, as with all PBKDFs, by using more machines. Allowing a KDF access to a network as part of the computation, removes the possibility of brute force a key due to restrictions the server can place on queries; however, it opens the door to new attacks, which keep it from being the perfect solution (traffic analysis and server compromise).

8 Performance

As the system must communicate with the cloud in order to perform classic read and write operations, the performance of the system is largely limited by the connection speed of each individual user. However, we can look at the overhead provided by this system in terms of the security operations. Here we see the various overhead created for each type of operation of various file sizes.

These tests were performed with files of varying sizes (10kB, 100kB, and 1MB) containing random input. There were over a hundred different files for each size. For the tests, we can consider a functional cost $N_A(l)$ for the usage of the network, where l represents the length of the message being sent to or from the cloud and A refers to a specific action (read, write, create, etc.). We can assume that $N_A()$ scales linearly with l and that it includes all of the effort performed by the cloud provider for a given action type. The basis for comparison is how long it would take to perform actions without any encryption at all. Thus, for any action, the time we are comparing is $N_A(l)$ vs. $T_A(l) = C_A(l) + N_A(l)$, where $C_A(l)$ is the cost to perform the various securities (i.e. the overhead induced by the system).

Figures 4, 5, and 6 depict the comparison of time to perform the given action without any security vs. those actions with the functionality of this system included.

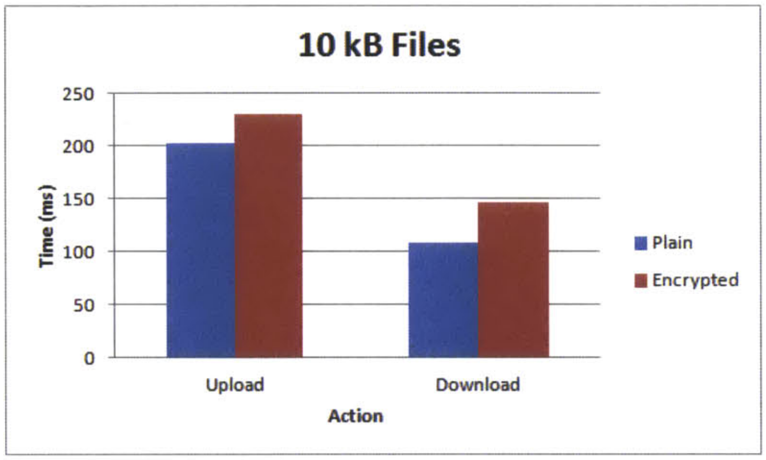


Figure 4: This graph shows the average over 100 files performing the designated actions with and without the system's encryption.

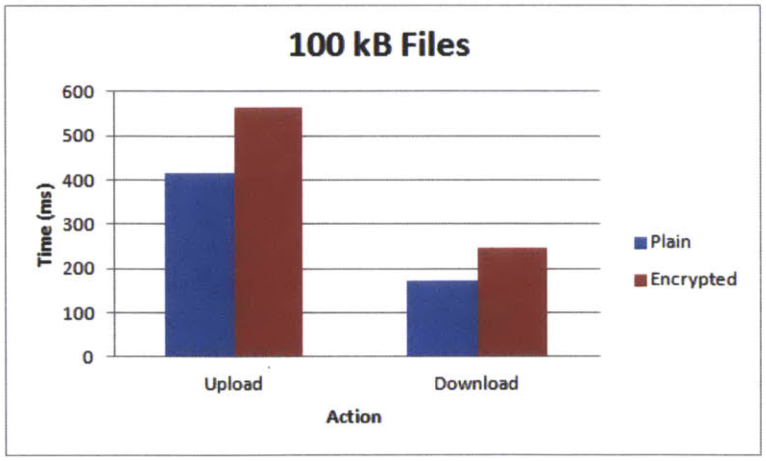


Figure 5: This graph shows the average over 100 files performing the designated actions with and without the system's encryption.

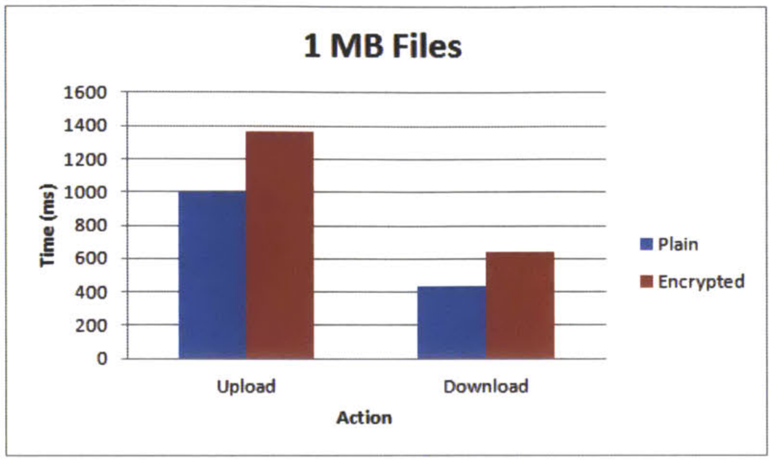


Figure 6: This graph shows the average over 100 files performing the designated actions with and with the system's encryption.

These times includes everything from reading in the file or writing out the file, performing the necessary encryption/decryption, and communication data-objects to/from the cloud. These results showed a 13.86%, 35.49%, and 36.41% overhead for uploading when including the security features for the 10kB, 100kB, and 1MB files respectively, culminating in a just below 30% speed reduction for uploads. Download results showed speed reductions of 35.19%, 43.02%, and 45.89% for the three file sizes, providing an overall performance reduction of slightly above 40%.

With different network access speeds these results may vary slightly, but the security features provided by the system do not prove to be detrimental when compared to the network speeds. All in all, for the interacting with the cloud using this system, there is less than a 50% overhead for any operation.

It is useful to note; however, that as the framework becomes more complex and more tools are included to provide more security, these overheads will be increased to accomplish the tasks.

9 Discussion

Although there are plans for the next stages of the system, currently the system does not provide protection against attacks like traffic analysis and denial of service. We feel that these are more to be combated at the level of the cloud provider and thus we make no effort in protecting against that. Also, although we can detect if the boxes stored on the cloud have been manipulated, there is no way for us to protect against an attacker who deletes boxes from all of the cloud servers that it is stored on. However, that does not mean they can access the contents of the box.

This project has a lot of room to grow. Things like version authentication, traffic analysis, etc. can still be explored for the system. In fact, the hope is that new boxes will be constructed that have protection against these types of attacks. One could construct a box that is equipped with a log, thus performing its own last version authentication. One major area of research is the recovery of a lost *masterKey*. As there is never anything stored pertaining to this key, on or off of the cloud, there is no way for a user to recover the *masterKey*. This is less than ideal, as a user would lose everything they have stored in our system if they lose any one of the entropy sources used to derive the *masterKey*.

We aimed at producing a system that was secure enough for the private sector. We have, thus far, created a system that is reliable and will not leak data to outside sources. Until we are able to guarantee version consistency, reliability, and *masterKey* recovery, the system will not be of use. This is not due to a lack of security in the areas of privacy and authentication, but rather in the areas of consistency and reliability. Our team requires a system that has all of the things we included and did not. The project does span for another two years, minimum, so these things will become a part of the system. We have succeeded in producing the base construct for our overall goal. The system uses new key derivation techniques that are even more difficult for an attacker to break, key management techniques that are fast and provide UNIX like permissions for the users of the system, and included a new framework for a file system that guarantees the security of individual data-objects in terms of secrecy and authentication.

10 Conclusion

The system overall provides an adaptive level of security, but key management and separation of control have been accounted for. The securities achieved are reliable unless there is an attack to break the underlying cryptographic algorithms. The project has provided numerous ways of combating the main attacks used against key derivation functions by allocating more resources and hiding certain parameters of the function. Assuming a connection to the cloud has been established, the system operates with a reasonable overhead.

Although the iris biometrics project is still in progress, the system is equipped to handle this integration so that users can increase the number of entropy inputs that go into creating their *masterKey*. The *masterKey* is never stored anywhere, but is securely and reliably generated on session connects. It is based on a number of entropy sources are relies on new Key Derivation Functions for generation.

New Key Derivation Functions that provide additional factors of security for our users have been defined and implemented. These KDFs are also configurable and extendable. Using the TFMKDF in conjuction with the HKDF is strong combination that requires the attacker to waste time redoing operations that they already did in the past, but were unable to store do to memory overflows.

As was seen in Section 7, Terminate Full Match KDF and User Selection KDF provide improvements over existing PBKDFs by increasing the entropy of the passwords provided to the function through manipulation of the function itself and additional entropy from the user that is easy to remember. USKDF increases the work for weak passwords so that all users can be protected despite their lack of concern for security. Network KDF uses a new resource to provide a huge performance hit for the attacker, while not impacting an honest user severely, and also allows for the possibility of placing alarms in the hash function that can detect attackers. Thus, we have succeeded in creating new PBKDFs that are stronger/more complex and will stand up against more powerful machines than those currently being used. It is useful to note that, as PBKDF2 increases in strength and complexity, so do the above PBKDFs, without any modification, because they effectively use PBKDF2 as a black box.

11 Acknowledgments

Special thanks go out to the members of my team at MIT Lincoln Laboratory who have helped guide me in this project, contributed to the details and implementations described here, and provided feedback, comments, and help in writing this, especially Ninoshka Singh and Darby Mitchell. I would like to thank our Project Leader and my Thesis Co-Advisor, Gene Itkis. His continued feedback brought clarity and purpose to the write up of this project! Further thanks go out to my Thesis Co-Advisor, Nickolai Zeldovich, who has not only guided and helped the production of this thesis, but has also provided relentless feedback to ensure the best possible results, which I truly do appreciate (much more in hindsight), as does my team at the lab!

References

[1] Dan Goodin. 25-gpu cluster cracks every standard windows password in less than 6 hours, 2012. [Online; accessed 24-April-2013].

[2] Robert David Graham. Linkedin vs. password cracking, 2012. [Online; accessed 24-April-2013].

[3] S. Josefsson. Pkcs #5: Password-based key derivation function 2 (pbkdf2) test vectors, 2011.

[4] Hugo Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. *Lecture Notes in Computer Science*, 6223:631–648, 2010.

[5] C. Percival, Tarsnap, and S. Josefsson. The scrypt password-based key derivation function. *Network Working Group*, 2012.

[6] Colin Percival. Stronger key derivation via sequential memory-hard functions. *Tarsnap*, 2010.

[7] Rapid7. Linkedin passwords lifted. [Online; accessed 24-April-2013].

[8] Johnny Shelley. Bcrypt, 2002. [Online; accessed 16-April-2013].

[9] UserAnswer.com. How fast can you brute force pbkdf2. [Online; accessed 24-April-2013].