Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

FADE: Self Destructing Data using Search Engine Results

Author:
Sam Wood

Supervisor:
Dr. Naranker Dulay

Second Marker:
Dr. Anandha Gopalan

June 18, 2018

Abstract

We present a new self destructing data scheme called FADE that uses news search engine results to create self destructing data objects. In particular, we show that the scheme:

- produces data objects with well-defined, predictable lifetimes.
- is much faster than existing scheme designs in the space.
- can easily be deployed as a user-facing browser extension.

FADE places a degree of trust in search engines as third parties, trusting them not to proactively detect usage of FADE and store search result queries. We deem this attack sufficiently unlikely and propose an extension for mitigating it.

Acknowledgements

Dr. Naranker Dulay for supervising this project and providing valuable support and guidance.

My family for supporting me throughout my academic career.

My colleagues and friends for the pleasure of getting to know them throughout my time at Imperial.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Our use of technology is ever increasing - and as it does, so does our digital footprint; the digital traces we leave in our wake when we interact with technology. As we register to more apps and services, communicate more over social networks and chat with our friends over email and instant messaging, we are constantly adding to an ever-growing digital footprint. Our digital footprints are made up of all of the data that lies in the physical storage for all the technology we interact with. This leads to the question - where will our data be in 5, 10, 50 years time? Will it still exist and who will have access to it? Once our data is handed over to an external party, we know very little about how it will be handled in the years to come.

With the complexity of modern technology stacks, our data is (necessarily) replicated and backed up in many different locations by the myriad of third party services that we trust our data with. This means it's hard to know if and when (if ever) our data has been deleted and even harder for us to *ensure* that it has been deleted. Most importantly, as it stands, most of the things we post online are *by default*, permanent. Adding to this, we must trust companies to both handle our data appropriately and to ensure that it doesn't get into the hands of malicious actors. In recent years, we have seen a spate of privacy breaches such as those at Equifax[2], Uber[3], NSA[4], Yahoo[5], Adobe[6] and Facebook/Cambridge Analytica[7]. As users of the web, we implicitly *trust* third parties to do everything securely and in our best interests. Naturally, it leads to the questions - does it have to be this way?

1.2 Retrospective Privacy

The user-facing privacy aspect of data deletion is termed *retrospective privacy*. Retrospective privacy simply means that users have a guarantee that at some point in the future, it will no longer be possible to read their data. The research field of Self Destructing Data attempts to this particular aspect of privacy. This space of research defines ways we transfer arbitrary information (such as emails, Facebook status updates, tweets, etc) such that after a certain amount of time, the data will "destruct", regardless of who possesses it and where it is stored. Self Destructing Data designs attempt to achieve this by encrypting data in a way such that at some point in the future, it will be impossible to decrypt the data. In this sense, the data is "destructed" in that it is useless without the destroyed encryption key. One important property to consider of such systems is the degree to which they rely on trusted third parties.

1.3 Main Challenges & Goals

We identify three main challenges in developing a "Self Destructing Data" system:

- Well Defined Lifetimes - when creating data that self destructs, users would like some control over when the data will expire along with strong guarantees over the latest time of destruction. Even better is if they can later choose to extend or revoke the lifetime of an object as and when circumstances change.

- Feasible to Deploy in the Real World - is the system practical to implement? Would it work well for end users and is it reliable?
- Limited Reliance on Trusted Third Party - does the system rely on a trusted third party? How much so?

We thus define our goal to develop a Self Destructing Data system that has well defined lifetimes, is deployable in the real world and does not rely on any trusted third party. In particular, we wish to develop a system that can be applied to everyday digital interactions such as emails, Facebook posts, Tweets and Instant Messaging.

In our security analysis, we consider an attacker to be a malicious adversary who desires to violate retrospective privacy by attempting to view a *specific* piece of (self destructing) data *after* its desired expiry time.

1.4 Contributions

Our main contribution to the field of Self Destructing Data is a new scheme, Fade which uses search engine results to create self destructing data objects. In particular, we show that Fade:

- produces data objects with well-defined, predictable lifetimes.
- is much faster than existing scheme designs in the space.
- can easily be deployed as a user-facing browser extension.

Fade places a degree of trust in search engines as third parties, trusting them not to proactively detect usage of Fade and store search result queries. However, it is worth noting that in section 4.1.4 we show how the scheme can be extended to mitigate this.

Chapter 2

Background

2.1 Snapchat & other mobile-centric social apps

Snapchat is a social media messaging app released in 2011. Snapchat is unlike most chat apps as all messages on the platform are automatically deleted by default after a short time window. In addition to this, Snapchat is primarily used to send annotated photos, although it can also be used for text based chat. Users can set a time on their messages, termed *"Snaps"*, after which they are no longer viewable. The times typically vary between 1 and 10 seconds and in its core functionality, it's not possible to make this longer. Once a recipient has opened a Snap (picture message), it can only be viewed for that specified amount of time.

Snapchat quickly became a popular social media app with the proclaimed USP of "self destructing image messaging". This proved popular to a user-base familiar with more conventional methods of communicating such as text-based emails and instant messaging. At a glance, Snapchat may seem like a solution to the issue of Self Destructing Data, but it is not for two reasons:

1. Users share messages through a centralised Snapchat platform, which is a trusted third party to the fullest extent.

2. The messages are not automatically self-destructing. By lack of any other information, we assume they are deleted "manually" by server-side software.

The problems above present several issues from a user privacy perspective. Firstly, 1. above means that users data and control over it is handed over to Snapchat, a trusted third party. The internal workings of Snapchat are unknown and thus in the future a retrospective attack might be made possible. There is a subtle discrepancy between messages that the user can no longer view after a certain point in time and messages that can not possibly be interpreted at all, by anyone, after a certain point in time. In this case, Snapchat is trusted with the former - we cannot view the messages after they have "expired", but we have no control over what Snapchat does with them nor when they are actually deleted.

Significantly, in 2013, Gibson Security found an exploit in Snapchat's API that allowed anonymous hackers to obtain and publish 4.6 million Snapchat usernames and phone numbers[8]. Following on from this, in 2014, the Federal Trade Commission made a complaint to Snapchat which resulted in a company blog post stating they, Snapchat, "can't guarantee that messages will be deleted within a specific timeframe"[9].

Further to this, 2. above, means that even a well-intentioned third party service could accidentally leave exposed data on some machine. This could arise from a bug in the manual deletion code, or just from the complexity of modern data storage systems and caching in networks. Modern, high-load web apps today require lots of redundancy and automatic data backups which make thorough deletion a difficult task.

These things combined mean that apps such as Snapchat are non-solutions. We must search elsewhere for solutions that are both trustless and automatically self destructing.

2.2 The Ephemerizer

2.2.1 Overview

Ephemerizer[10] sets out to attach time limits to data, which can applied to domains such as emails.

It points out that an ideal solution to give control of data expiry over to users might involve giving each individual the *responsibility* to "create, reliably store, certify, advertise" the decryption keys for their data upon request. For example, if Bob sends an encrypted email to Alice, then Bob could store the decryption key on his own machine and temporarily lease it out to Alice or other parties when needed. Whilst this seems to be the strongest form of control we can have over the access of our data, the paper points out that it is impractical as it places a large burden on each individual - for their data to be available persistently, their machines need to have high availability. This would require cost and expertise which are things that we can assume our ideal end users do not have. Essentially, putting the onus of "key management" onto individuals is impractical.

Therefore, the authors suggest having a single centralised service, dubbed the "Ephemerizer" which is responsible for the "create, reliably store, certify, advertise" aspects of user key management. This means that to create and read time-limited data, clients are required to use the Ephemerizer service to deal with the key management.

Whilst there is the trusted Ephemerizer service involved, the solution presented sets out to rely on it as little as possible. As such, the system works so that encryption only requires the Ephemerizer service to advertise the keys and otherwise doesn't require any other interaction. However, decryption requires active participation from the service.

2.2.2 Limitations

Given the ideal properties of a solution to the problem outlined in the introduction section, the Ephemerizer system is ill-suited to our needs as it is not truly "trust-less". The reliance on the third party Emphemerizer service, which could be compromised, is a prevalent attack vector in the current cyber security climate.

The paper considers various attacks on Emphemerizer and proposes that to reduce the chances of retrospectively compromised data, the Emphemerizer service can store all the keys on a tamper-resistant smart card. This assumes that there are trusted people to ensure that the smart card is not stolen. If the Emphemerizer machine were to be compromised and run infected code, the safety of the keys would not be at risk, given that the smart card stores them in isolation.

It is also worth pointing out that the central Emphemerizer service means that even if Bob's machine is compromised, an adversary would not be able to decrypt any of his self-destructing communications after the expiry period.

2.3 DataBox, FreedomBox, etc

2.3.1 Users storing their own data

There is a family of ideas which aim to give users complete control over storage and access of their data specifically by allowing users to store their own data on their own, locally hosted machine or "box". We call this family of ideas "box" type systems - these essentially are the kinds of solutions that are ruled out in the Ephemerizer paper for placing responsibility on users.

Two such works in this place are Databox[11] and FreedomBox[12]. The principle idea is that users store their own data on their own machine, or "box". Users never need to *give* away their *unencrypted* data to a third party for storage and they just store it themselves. This means that users have explicit control over which parties can access their data aswell as confidence that it is being stored securely. The Databox project proposes a combination of storing important metadata on the users "box" and using external cloud-hosted providers to store self-encrypted documents.

These ideas typically manifest physically with the notion of a dedicated, internet-connect machine that runs in the users home. This machine is accessible from the outside world and can be accessed from anywhere in the world, given the user provides valid authentication credentials. In a world where systems like this are commonplace, users simply store things such as their calendars, personal documents, health records and cryptocurrencies on their own machine and where needed, access it themselves remotely, or give partial permission for a third party to access it. Interestingly, the Databox approach to third party access is to run the third-party code (presented as an "app") on the databox itself and then report anonymised results to the remote third party.

Both Databox and FreedomBox have similar notions of "apps" (Databox) and "plugins" (FreedomBox) which allow users to extend the functionality of their box by adding an app, say, to plot the light sensor data of the users mobile data.

2.3.2 DataBox & FreedomBox - the future of privacy?

Ideas such as these are appealing from a user-privacy perspective; they tend towards digital profiles[1] which are in full control of users, rather than fragmented across multiple trusted third parties (Facebook, Email provider, GitHub, Twitter, Medical Companies, etc). Perhaps, this is the future of privacy - although in its current state, there is lots more work to be done.

2.3.3 Issues

As pointed out in Ephemerizer[13] (which we we saw in section 2.2) there are practical issues with getting a user to store data on their own machine:

- As it stands, users need a fair amount of technical knowledge to get such a system up and running. As projects such as Databox and FreedomBox develop, we would hope that they become easier to use for everyday end users.
- The "box" which the user stores their data on needs to be highly available; this is expensive and again, requires technical expertise.

Additionally, as with any such system, there will always be a need for data to be be "leased" in some way; for a remote doctor to view a patients health records, they would need access to the unencrypted data in some form. This leads us to realise that whilst these projects would solve many of the problems about data-ownership, there is potential for them to co-exist with the self destructing data solutions we will see below.

2.3.4 Potential

We conclude that such "box" systems serve as a compelling insight into what could be a huge improvement in giving users control over their data. However, in their current state, they are not quite ready and in a hypothetical world where they are commonplace, it would still be desirable to have self destructing data solutions that don't involve trusted third parties. Thus, integration into these "box" systems are a potential future application of self destructing data schemes.

2.4 Vanish

2.4.1 Overview

Vanish[14] is similarly motivated to find a way to give users back control of their data, by providing a mechanism for it to self destruct. The Vanish paper proposes a way to achieve trust-less, self-destructing data using cryptography and the pseudo-randomness of distributed hash tables, hereon abbreviated as *DHTs*. The method presented in this paper is truly self destructing in that after a certain amount of time, it is impossible to decrypt the data. The key contribution from the Vanish paper is the removal of the trusted third party that is present in Ephemerizer.

[1] Digital profiles meaning the sum of every bit of digital information stored about us.

2.4.2 Self Destructing Mechanism

The solution presented in the paper uses a concept called Vanish Data Objects, hereon VDOs. VDOs are an abstraction that encapsulates sensitive data in a cryptographic object that has an approximate expiry date. Users encrypt their emails/messages into a VDO then send them over the network. Decryption is also done by the end-user at the application level. Note that any third parties which help the VDO travel from sender to recipient will be unable to access the contained data after it expires.

As per the Vanish paper, a VDO, containing data D is created as follows:

1. Choose random key K and use it to encrypt D to get ciphertext C
2. Split K into n different parts using *Shamir's Secret Sharing*[15] with redundancy specified by *threshold*, the % of shares required to reconstruct K.
3. Choose a random access key L and use it to seed a pseudo-random number generator.
4. Generate n random DHT indices $i_0..i_n$ using a pseudo-random number generator, seeded by L.
5. Put the n different parts of K into DHT indices $i_0..i_n$.
6. Construct the VDO: $\langle L, C, n, threshold \rangle$

Unwrapping a VDO into its underlying data object D is the symmetric process of that above.

Most significantly, the key parts put into the DHT in step 5 will be unavailable after some time due to the DHT's churn of nodes pseudo-randomly joining and leaving the network. Once nodes leave the DHT network, they no longer serve the keys which they were responsible for. Additionally, as DHTs are typically made of nodes that are geographically distributed, the Vanish paper observes that obtaining the keys retrospectively is likely to be very difficult from a legal perspective.

2.4.3 Threshold Trade-off

Another key element of this algorithm is the threshold % chosen for the key shares. The key threshold is the % of key shares required to reconstruct the original key. There is an inherent trade-off here - if the threshold is 100%, then if just one key share is lost in the DHT, the data has already destructed. Conversely, if the threshold is very low, i.e. 10%, then the window of potential attack is larger as 10% of the key shares in the DHT will be available for a longer time. Subsequently, increasing the threshold increases security and decreasing it increases availability. Thus, the threshold is a trade-off between security and availability.

2.4.4 Practical Deployment

As a proof of concept of the system, the Vanish team developed a Firefox browser extension which gives users a contextual menu option which allows them to encrypt plaintext into VDOs and decrypt VDOs back into plaintext. Effectively, this allows users to have self-destructing email threads, providing that they wrap the sensitive parts of each message into VDOs.

2.4.5 Security Analysis

For security analysis, the Vanish team laid out that any third party can be considered adversarial. This includes external web services, government agencies and individual machines belonging to the DHT network. A potential attack would involve continuously scraping the contents of the DHT over time in an attempt to build multiple snapshots of the DHT contents. If the snapshots are frequent enough and with big enough coverage, then an attacker can use them to decrypt the contents of a VDO, even after the indices have been ejected from the DHT. The Vanish team concluded that doing this would require an infeasible amount of computational power, costing roughly $860k a year.

2.4.6 Limitations

Guarantees of VDO lifetimes

One disadvantage of the DHT based approach is that there is limited control over the availability of the data. The threshold provides some level of controls, but there is no fundamental guarantee that the data will or will not be available after a given point. The Vanish paper acknowledges this by making the assumption that the user prefers VDOs to be destroyed prematurely than risking exposure to an adversary.

Pre-determined Lifetimes

Another issue is that the Vanish solution requires users to know up front roughly how long the data should be stored for. As pointed out in Neuralyzer[1], this is a strong assumption that makes the system less user-friendly[2]. It forces users to choose when the data will expire, giving them the choice to either be extra cautious or very laxed. When they choose to be extra cautious, there is a risk that data will expire before it has been used for its intended purpose. This means that data would sometimes need to be renewed periodically by the user, which increases the cognitive overhead of using the system. On the contrary, if the user chooses to be very laxed with the expiry time, then this increases its potential (but not guaranteed) availability and subsequently the risk of it having negative consequences in the future. Either way, the requirement to select a time when data expires makes the system less user-friendly.

Sybil Attacks

The original Vanish paper also goes to some lengths to perform security analysis to show that attacking the system is computationally impractical. However in section 2.5 we will see that that this analysis is not exhaustive.

2.5 Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs

Continuing the dialogue started by Vanish, *S. Wolchok et al.* published work showing a way to defeat the Vanish system at much lower financial cost[16] than presented in original paper. It also proposes that alternative approaches to building Vanish are unlikely to be any more secure. In this section we'll abbreviate this paper as *DV*, short for Defeating Vanish.

Following on from the DHT-scraping attack analysed in the Vanish paper, DV suggests that the widespread use of Vanish as originally presented might even encourage a market for e-services that can be used to attack VDOs. Such an e-service might continuously scrape the DHTs and then offer temporal DHT snapshots to adversaries for a fee.

Specifically, DV argues that a Sybil attack can be mounted at reasonably low cost by relying on a particular property of the Vuze DHT (the DHT used by Vanish). Vuze nodes replicate the data they store onto neighbouring nodes, meaning that any adversarial node on the network automatically stores the backed up contents of some other nodes. By using this detail in a Sybil attack, DV shows that it would cost approximately $5900 a year to recover 99% of VDOs. For a state-sponsored or coorporate attacker, this is a relatively small cost to violate the retrospective privacy of 99% of VDOs.

2.6 Neuralyzer: Flexible Expiration Times for the Revocation of Online Data

2.6.1 Overview

Neuralyzer[1] sets out to address some of the shortcomings of Vanish and related work such as Ephemerizer. Systems such as Vanish and Ephemerizer outlined above rely on the user being able

[2]It is difficult for users to predict the time at which their sensitive data should expire in the future

to accurately predict when their data should be deleted. Conversely, Neuralyzer proposes a system where the lifetime of data can extend based on how often it is accessed. For example, an email message could be retained for as long as there is sufficient interest in it. As interest drops over time, the email will "self destruct". The authors develop both a protocol that sets out to achieve this and a demo, called Neuralyzer which shows how a possible system could work. Similar to other works in the field, Neuralyzer aims to provide a degree of retrospective privacy on users data.

Similarly to Vanish, Neuralyzer seeks to achieve a system that harnesses an existing distributed system without requiring it to be modified. Crucially, Neuralyzer seeks to reduce the availability of data based on events such as the data being read. This is a key difference to the work of Vanish where data availability is tied solely to time.

2.6.2 Flexible Lifetimes in Neuralyzer

Unlike Vanish, which relies on DHT churn as a pseudo-random source to delete data, Neuralyzer uses the caching mechanisms of DNS to achieve this. In particular, they harness the "Time to live" (TTL) property of data in DNS caches to gain control over how long the data lives for. They claim that under this approach, it is possible to have flexible expiration times ranging from a few days to a few months.

It is worth noting that a prior publication called EphPub[17] introduced the idea of using DNS infrastructure to achieve self destructing data. Neuralyzer's key contribution is presenting a way to achieve flexible object lifetimes using DNS infrastructure.

Neuralyzer can automatically delete data based on three heuristics:

1. Drop of interest
2. Excessive interest
3. Manual revocation

EDO Creation

Neuralyzer presents a data structure called an Ephemeral Data Object (hereon EDO), which encapsulate the data which will destruct by these heuristics. The EDO data structure is conceptually similar to the VDOs in Vanish.

EDO lifetimes are split into three phrases:

1. Construction
2. Access
3. Revocation

Neuralyzer's construction phase is fundamentally different to that of Vanish. Whilst Vanish splits the key into parts and hides them in a DHT, Neuralyzer encodes the key into the caches of DNS resolvers. Whilst the original idea of using DNS infrastructure to achieve self-destructing properties was presented in EphPub[17], Neuralyzer extended it by allowing data to destruct as interest in it wanes.

EDOs are constructed in Neuralyzer as follows:

1. The data is encrypted using a random key R.
2. Each bit of R is associated with a set of DNS Entries. DNS entries are tuples of the form $\langle$*Domain, DNS Resolver*$\rangle$. This process is termed *DNS Portrayal*.
    - The DNS entries are chosen randomly from a precompiled list, which the authors suggest could be compiled by scraping the web and performing reverse DNS lookups.
    - It is required that the DNS Resolvers initially do not contain the domains in their cache.

- The number of DNS Entries associated with each bit is a variable associated with the threshold, which is explained below.

3. For each bit of the key R that is a *1-bit*, recursive DNS queries are made for all of the DNS entries associated with that bit. This has the effect of the DNS resolvers storing the result of the domain name query in their caches.

4. The EDO is constructed as a tuple containing the encrypted data and the DNS entries for each key bit.

The access phase sheds some light on the reasoning behind the construction phase. An EDO, E, can be decrypted as follows:

1. For each set of DNS entries, corresponding to key bit, i,:

   - Perform a non-recursive DNS query on all the DNS entries.
   - If more than *threshold* entries are cached, then consider bit i of the key to be a *1-bit*.
   - Otherwise, consider bit i of the key to be a *0-bit*.

2. Form a key using all the key bits obtained from the previous step and use it to attempt to decrypt the ciphertext stored in the EDO.

Note that the decryption process in step 2 could fail if the reconstructed key is not the same, indicating that the EDO has expired.

2.6.3 Properties of DNS Caches

The nature of encoding the key in DNS cache entries means that over time, the key will be lost due to domains being evicted from the cache. When the key is lost, the access phase fails to reconstruct the original key, R, used to encrypt the data, meaning that the data is effectively "destroyed".

The use of DNS cache entries also mean that data that is not accessed for a long period of time is less likely to be available. Note that the DNS queries made in the access phase refresh the lifetime of the domains in the cache. This means that the probability of successfully reconstructing a key is proportional to how recently it has been accessed. This is one of the fundamental contributions of the Neuralyzer paper - data can be set to destruct as interest in it decreases. To an extent, this solves the problem in Vanish where the user needs to accurately predict how long they will need the data for, or, alternately, manually request for the lifetime of the data to be refreshed.

The access phase outlined here shows how Neuralyzer can delete data based on the *drop of interest* heuristic mentioned above. This automated deletion via DNS caching mechanisms is effectively the revocation phase of an EDOs lifetime.

2.6.4 Remarks on trade-off parameters

In DNS Portrayal of the EDO construction phase, there are some interesting trade-offs to consider. During DNS Portrayal, each bit of the access key is associated with a set of DNS Entries. The size of these DNS Entry sets plays an important role in the performance of the overall system. For the purposes of our summative discussion here, we will term the number of DNS entries per bit as N. There is also the threshold used in the access phase which determines how many DNS entries must store the domain in their cache in order for the key bit to be considered a *1-bit*. We will term these threshold values x, as in the original Neuralyzer paper.

The parameters N and x present a fundamental trade off in availability and security. Observe that if N is large, then there is more scope for increasing the availability of the key in the DNS caching mechanisms. However, this increased availability comes at the risk of the key being exposed for longer than is safely necessary. A large N means that even if the data hasn't been accessed in a considerably long time, there is a greater probability that the key will still be obtainable by a standard run of the access phase. On the contrary, if N is small, then there is a risk that the key

can dissipate prematurely.

The threshold parameter, x accounts for the uncertainty in DNS caching and serves as an error correction mechanism. We can motivate this by imagining that N is just 1. As DNS is a public resource, users not associated with Neuralyzer query a variety of domains frequently. If a bit in our key is a *0-bit*, then its associated DNS entry only needs to be queried once by any external DNS user for the key to be lost; any user who happens to query the same domain entry will cause the DNS resolver to cache the domain. This means that any Neuralyzer user trying to access an EDO will then erroneously read that bit of the key to be a *1-bit*. When $N > 1$ there is a probability that this *0 to 1* bit error occurs for each of the DNS entries associated with a given bit. The parameter x determines is the number of DNS entry cache hits required for the bit to be interpreted as a *1-bit*. Similarly, when reading the key, *1 to 0* bit errors can occur through standard DNS cache errors or by occasional cache flushes or reboots.

2.6.5 Security Remarks

Neuralyzer is not vulnerable to Sybil attacks in the same way that *Vanish* is. With Neuralyzer's DNS approach, the authors point out that Sybil attacks are not possible as arbitrary nodes cannot volunteer themselves as standard DNS resolvers. In this sense, Neuralyzer has baked-in defenses against these kinds of attacks.

Similarly, the authors conclude that brute force attacks are intractable as the space of possible domain names is too large to snapshot the entire DNS structure at different points in time.

2.6.6 Limitations of Neuralyzer

Performance

A concern with the Neuralyzer implementation is its heavy reliance on DNS. Given that a large N is required to get reliable availability, from a user facing perspective it may take too long to encrypt and decrypt EDOs. A 256-bit encryption key with $N = 20$ requires a maximum 5120 DNS queries to encrypt and decrypt the EDO.

Furthermore, the Neuralyzer system has a time consuming setup phase which must run before the creation of any EDO. In the setup phase, Neuralyzer searches for appopriate DNS resolvers to use for EDO construction. In our experiences of Neuralyzer, this set-up takes a long time (in the order of several minutes) as many of the DNS resolvers are not suitable, or simply do not reliably respond to DNS queries.

It is worth nothing that while the DNS requests could be done in parallel, many of them might need to be manually retried due to the unreliabile nature of UDP[3], the protocol on which DNS runs.

Feasibility of Real World Deployment

One disadvantage of Neuralyzer, is that it requires access to low level DNS operations. In the current ecosystem of browser extensions[4], a Neuralyzer browser extension would require a separate component that runs on users machines that provides the extension with the required DNS operations.

Security of DNS Infrastructure

Additionally, whilst DNS is decentralised in nature, it is not invulnerable to attacks. The Neuralyzer authors point out that Kuhrer et al. found that 20% of DNS resolvers in the DNS infrastructure run a BIND version that is vulnerable to remote code execution vulnerability[18]. Whilst this figure may have changed since the publication of Neuralyzer, it is likely still large enough to be a concern for the security of the system.

[3] User Datagram Protocol

[4] In the Mozilla web extensions API, there is a function `DNS.resolve(domain)`, although it does not allow the caller to specify the particular DNS resolver to send the query to.

2.7 Forgetting with Puzzles

In *Forgetting with Puzzles*[19], Amjad et al. design a system to mitigate the risks of scraping attacks on Self Destructing Data systems. Whilst they look specifically at Neuralyzer, their design is generalised to any SDD scheme which exposes details of key reconstruction in an information table. Their paper aims to protect information tables from deletion attacks as well as the proactive scraping attacks.

We first briefly define what we mean by "deletion attacks" and "proactive scraping attacks" in the context of self destructing data systems:

- **Deletion Attacks** - these are attacks whereby a malicious agent attempts to cause self destructing data objects to delete well before their intended expiry time. A scheme which is vulnerable to deletion attacks can potentially be rendered useless if it is fairly inexpensive for an attacker.
- **Proactive Scraping Attacks** - these are attacks where a malicious agent attempts to proactively collect and decrypt all self destructing data objects (or a large proportion of them) in the public domain. Such an attack means that for an object that becomes of interest to an attacker after it's expiry time, there is a high chance that the attacker will already have it stored from their prior efforts.

The authors of Forgetting with Puzzles show how SDD objects can be protected from the above attacks by protecting their information table using Rivest et al's time locked puzzles[5][20]. These puzzles are configured such that the time it takes for a machine to solve one such puzzle is proportional to the CPU power of that machine, rather than the number of machines available. In other words, the puzzles are designed such that there is no benefit in attempting to solve them in parallel.

The key innovation of the work is to have an SDD object contain *two* information tables τ_1 and τ_2 , each protected by two distinctly different cryptographic puzzles, ρ_1 and ρ_2 respectively. Amjad et al. stipulate that the ρ_1 should be an easier puzzle and ρ_2 harder. The idea here is that if an attacker successfully performs a widespread deletion attack on τ_1, honest users will still be able to reconstruct the data object (albeit in a more timely manner) by solving ρ_2 to get access to τ_2. The difficulty of ρ_1 should be set to significantly increase the cost of a wide-spread deletion attack, yet easy enough to provide an acceptably fast object reconstruction time for honest users. It is worth noting that Ajad et al. also lay out further mechanisms that involve the extension of key lifetime, although they are beyond the scope of this paper.
We will revisit *Forgetting with Puzzles* in a more applied context in chapter 4.

2.8 Neuralyzer: the most apt SDD scheme to date

We have now explored a variety of approaches towards self destructing data. Given the related work studied, most notably Vanish, EphPub & Ephemerizer, we propose that Neuralyzer is the most apt scheme to date, given our goals. In particular, its core strengths are:

1. No reliance on a trusted third party. The DNS infrastructure is decentralised to a convincing degree.
2. Flexible expiration times based on access patterns.

The original Neuralyzer paper presents a standalone prototype implementation for the DNS based scheme, implemented in Python, although this is not a system that is suitable for end-users.
There are some open challenges in deploying Neuralyzer in the real world, some of which are presented in the original paper. We identify the principal challenges to be:

- Performance - with a 128 bit key, with 20 DNS Entries per bit, encryption and decryption requires a minimum of 128*20 = 2560 DNS queries. Whilst they can be parallelised to a high degree, it's still a bottleneck to performance.

[5]Unlike say, the Cryptographic puzzles used in Cryptocurrencies

- Availability (& Security) - in our own practical experiments, we have observed that it's extremely rare that a 128 bit is accurately reproduced with $n = 20, x = 2$ as suggested in the paper. As posed in the Neuralyzer paper, availability trades off directly with security.
- Real World Application - Neuralyzer is yet to be implemented in a way that is accessible to non-technical users.

Chapter 3

FADE: Self Destructing Data Using Search Engine Results

3.1 FADE Overview

In this chapter, we detail our design for a new self destructing data system, FADE, that makes use of search engine results. We approach the design problem iteratively; we begin with an initial design for a system called `PhraseFader`, then develop it into `NumericNewsFader` and subsequently `NumericSequenceNewsFader`. Finally, in section 3.9 we modify `NumericSequenceNewsFader` to produce our final design for FADE. The final FADE scheme is detailed in full in appendix A.1.

3.2 Goals

In chapter 2, we saw a variety of existing related work. Coming closest to our goals is the Neuralyzer system, however there are some problems with it's real world deployment and reliability. We now set out to design a new Self Destructing Data system that meets our original goals set out in section 1.3:

- Well Defined Lifetimes
- Feasible to Deploy in the Real World
- Limited Reliance on Trusted Third Parties

3.3 SDD Scheme Definition

We formally define Self Destructing Data Schemes, hereon *SDD Schemes*, as schemes that encrypt plaintext data in such a way that after some amount of time has passed, it is no longer possible to decrypt and obtain the original data. Specifically, an SDD scheme has the following well-defined entities:

- *Objects*[1] - these are constructs of the form $\langle ciphertext, infoTable \rangle$. *infoTable* contains all the nescessary infromation to attempt to decrypt the data at a future point in time.
- *Object Construction Algorithm* - an algorithm that transforms some input plaintext into an object. The crux of an object construction algorithm is the way in which the encryption key is encoded in the object's information table.
- *Object Reconstruction Algorithm* - an algorithm that takes an object as input and decrypts the ciphertext back into the original plaintext. This involves accessing the information table of the object and performing some action to attempt to reconstruct the key.

[1] Objects are typically termed with respect to the SDD scheme in which they exist; e.g. Vanish[14] has *VDOs*, Vanishing Data Objects and Neuralzyer[1] has *EDOs*, Ephemeral Data Objects.

Note that typically, the object construction algorithm is an operation that will always succeed under normal operating circumstances. Whereas for the object reconstruction algorithm, failure to decrypt the ciphertext is a necessary (and desired) outcome.

Under this definition of SDD schemes, the *Self Destructing* nature is determined by the nature of the information table. Table 3.1 below shows how the Vanish and Neuralyzer schemes fit with our definition above.

SDD Scheme	Objects	Information Table	Object Construction	Object Reconstruction
Vanish	VDOs	DHT Entries	Inserting Shamir's Secret Shares into a DHT.	Retrieving shares from DHT, recombining into key.
Neuralyzer	EDOs	DNS Cache Entries	Encoding key into ephemeral bits, caching them in DNS resolvers.	Inspecting caches of DNS resolvers to reconstruct ephemeral bits.

Table 3.1: Summary of Vanish and Neuralyzer with respect to our definition of SDD schemes.

3.4 SDD Data Sources

The information table in SDD data objects typically links to external sources; Vanish's information tables link to keys in a DHT and Neuralyzer's information tables link to caches in DNS entries. We term these external sources *SDD Data Sources* as they are the components that SDD schemes rely on to achieve their self destructing properties.

Formally, SDD Data Sources can be viewed as an abstract collection of data records that changes over time. We define three properties of an ideal SDD Data Sources:

1. **Churn** - an SDD data source must *change* in some way over time. When an SDD data source changes, some amount[2] of objects created under the SDD scheme are destroyed. The rate that the data source changes has a direct influence on the lifetime of data objects in the scheme.

2. **Intractable Retrospective Access** - it needs to be verifiably difficult for *any* party to view the state of the data source at any point in the past.

3. **Infeasible to Proactively Scrape** - it must be difficult for *any* party to proactively scrape and store a large proportion of the data source across different time points.

Evidently, these are not all possible to their fullest extent. However, they do serve as good ideals from which we can understand the common compromises that SDD schemes must make.

3.5 Harnessing Search Engines

3.5.1 Search Engines as a Self Destructing Data Source

Search engines present themselves as an interesting and novel direction for self destructing data schemes. Search engines are typically *internet-scale* and to a degree, exhibit the ideal properties of self destructing data sources.

Whilst search engines are traditionally indexes of the web, there are now many variants implemented by the biggest search engines: images, news, videos & products (shopping) to name a few. We begin by taking a high level look at how well search engines fit our ideal properties for SDD Data Sources.

Churn

Search engines are effectively scrapers of the largest data source in the world - the web. The web is inherently constantly changing through the constant creation, deletion and modification of web pages. However, one principle challenge of using search engines as an SDD Data Source is

[2]This amount could be zero

understanding the rate at which search results change - whether these be web, image, news, videos or product searches.

Intractable Retrospective Access

Search engines do not come with features that allow you to view search results of the past; the primary purpose of a search query is to give the best results according to the state of the web *at this moment*. Such a feature would be very expensive to implement and would have little use. There is no apparent way to view past search results for a specific query. To put this in some context, it is fairly easy to retrospectively view a data source that is a weather forecast for a particular city on a particular website.

Infeasible to Proactively Scrape

The degree to which this property holds for search engines greatly depends on the kinds of queries that an SDD scheme employs. For example, it would be fairly easy to proactively scrape one word search queries for the most common words in the English dictionary over time. However, it would be much more difficult to proactively scrape a large proportion of the search results for multi-word search queries over time. Note that the feasability of proactive scraping also depends on the churn of search results - e.g., if we find search results for certain kinds of queries to only change once a week or so, an attacker would only need to scrape and store results for all possible queries at an average rate of once per week. On the contrary, high scraping coverage of search results would be much more expensive if the search results were to change at a rate of 1 hour, for example.

However, we must remember that our ideal property for proactively scraping requires that it must be infeasible for *any* party to perform a large coverage of scraping. This definition includes the search engine companies themselves. The extent to which a search engine company could save a large proportion of search results depends both on the resources of the search engine company and the kinds of queries that are involved. Without any concrete scheme in place, it is difficult to ascertain whether it would be realistic for a search engine company to perform such a scraping attack. At this point we note this as a potentially large security risk which we will investigate in more depth in section 4.1.4 once we have explored the details of how a scheme would work.

Summary

We have shown that search engines have high potential as a data source in an SDD scheme. However, the potential of such a scheme requires an in depth analysis of the churn and the *Infeasible to Proactively Scrape* properties.

3.5.2 Canonical Results

Before designing an initial SDD scheme that uses search engine results as a data source, we first formalise the mechanics of search engines and particularly, their results. We do this so that we can analyse the scheme for a particular search engine without losing generality.

Definition of Search Engine Invocation

We define an invocation of a search engine to be made up of a *query* and *search type*. The search type is one of *web, images, news, video, products, etc.* An invocation returns a list of results[3]. Each result has a *heading* which is the leading title of each search result. In figure 3.1 we show an example of a search engine invocation on the DuckDuckGo[21] search engine.

Definition of Combining Functions

A combining function, denoted `f`, is a function that takes a list of outputted search results and transforms it into a cryptographic key suitable for encryption. This will become important later where we show how we can use combining functions to create self-destructing keys from search results. Below we give definitions for some combining functions which we will use later on.

[3]In this work we are only concerned with the first results page.

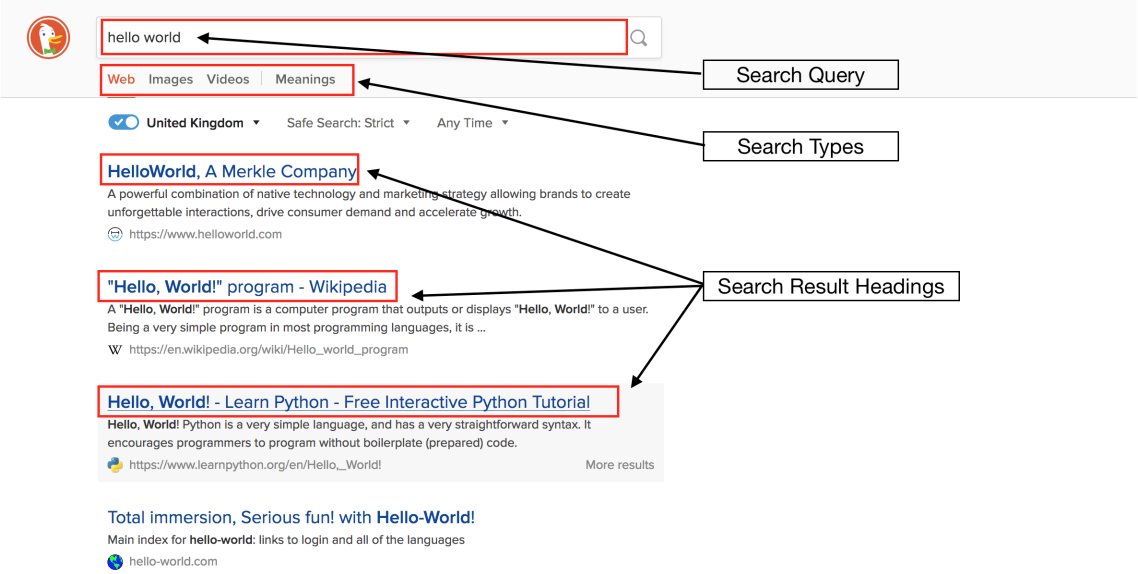

Figure 3.1: An example search query on the DuckDuckGo search engine. The *query*, *search type* and *search result* headings are labeled.

- `AllHeadings` - returns the hash of the concatenation of all search result headings.
- `FirstNHeadings(n)` - returns the hash of the concatenation of the first `n` headings.
- `LastNHeadings(n)` - as above, but working on the *last* `n` headings.
- `FirstNHeadingsSorted(n)` - as `FirstNHeadings(n)` but does not concern itself about ordering of the `n` headings results. This is achieved by deterministically sorting the headings before concatenating them.
- `LastNHeadingsSorted(n)` - as above, but working on the *last* `n`headings.

3.6 `PhraseFader`: an initial SDD scheme

3.6.1 Data Objects

Objects in `PhraseFader` are termed *Fading Data Objects*, hereon FDOs. FDOs are tuples of the format $\langle$`c, q`$\rangle$, where `c` is the ciphertext of the original data and `q` is a search engine query.

3.6.2 FDO Construction Algorithm

1. Generate random search query, `q`, a sequence of random words. We call this sequence of words a random phrase.
2. Query search engine for `q`.
3. Use some combining function, `f`, on the results page (first page only) to construct an encryption key, `k`, for the provided plaintext.
4. Encrypt the data with `k` to get ciphertext, `c` and construct FDO $\langle$`c, q`$\rangle$.

In this algorithm, our information table consists solely of the random phrase search query, `q`. Later we will see how the choice of `f` influences the self destructing properties of the scheme.

3.6.3 FDO Reconstruction Algorithm

The FDO reconstruction algorithm is the reverse of the FDO Construction described above. Given FDO ⟨`c, q`⟩:

1. Make search engine query for q.
2. Apply combining function, `f` to the search results to obtain decryption key k.
3. Attempt to decrypt ciphertext c with key k.

3.6.4 Feasibility

The main difficulty with `PhraseFader` is the challenge of investigating object lifetimes. To get an idea of the lifetimes that would arise, we would need to investigate the churn of search results for a large range of random phrase queries.

Our anecdotal experience shows that the churn of search result for a random phrase are hugely dependent on the words used in the query string; for example, if a search query contained a word recently used by the US president in a press conference, then the results would change very quickly, whereas search results for other words may not change for a long time. A consequence of this is that the time to change for any two random phrase queries is likely to have a huge difference that is entirely dependent on the words.

Therefore, such an investigation into the characteristics of random phrase search results would be extremely expensive, time consuming and would likely not provide any useful results.

3.7 `NumericNewsFader`- an improvement on `PhraseFader`

In this section, we take the main idea of `PhraseFader`and turn it into a more feasible SDD scheme, `NumericNewsFader`. We aim to design `NumericNewsFader`as a scheme that is realistic to analyse as a self destructing scheme.

3.7.1 Overview

`NumericNewsFader` is identical to `PhraseFader` except for the way the search query is generated. Instead of generating a random phrase for a standard web search, `NumericNewsFader` generates a random number for a news search.

The FDO construction phase of `NumericNewsFader` is:

1. Generate a search query `q`, a random number in a predefined range.
2. Query search engine for `q`, using the *news* search feature.
3. Use some combining function, `f`, on the results page (first page only) to construct an encryption key, `k`, for the provided plaintext.
4. Encrypt the data with `k` to get ciphertext, `c` and construct FDO ⟨`c, q`⟩.

Object reconstruction of `NumericNewsFader` remains unchanged from `PhraseFader` , other than changes to account for the adjustments to the query above.

3.7.2 `NumericNewsFader` Motivation

In changing from web searches for random phrases to news searches for random numbers we arrive at a shceme that is easier to measure. Additionally, we are led by the intuition that churn of news search results for a pre-fixed range of numbers is far less volatile than web searches for random phrases.

To explain this intuition, we can use the example of a random phrase composed of 4 random words. The frequency at which each word occurs in the English language on the web varies hugely. For

example, the word rain is far more common than the word petrichor. However, if our search query is a random number in the range 2000-6000, then the rate at which search results change for two numbers in the range, say 3746 and 5421, is likely to be similar.

The change from standard web searches to news searches is an effort to make investigation more tangible; with news searches, we have a smaller, better defined space of search results.

Therefore, `NumericNewsFader` has both the advantages that it is more feasible to measure and could potentially result in more predictable object lifetimes. In the next section we investigate the lifetimes of FDOs created by `NumericNewsFader`.

3.7.3 Object Lifetimes

Random Number Range

We must first establish a random for which the random number queries should be selected from. We observe that news search queries for large numbers, such as 526381645, are likely to give only a few results. It follows that the results for such a large number would seldom change. This would give rise to large, unpredictable lifetimes, not to mention an experiment that takes a long time to run! On the contrary, news search queries for small numbers between 0 and 10 are used a lot in writing. We suspect this would give rise to object lifetimes that are small and largely unpredictable. Therefore, as a starting point we choose the range 2000-6000 for which to select our random number queries for `NumericNewsFader`.

Experiment to Measure Lifetime

We now desire to get an understanding of the object lifetimes that `NumericNewsFader` gives rise to, when using random numbers in the range 2000-6000. Whilst our SDD schemes are applicable to any web search engines, we choose to run our experiment on Google for its popularity and ease of use.

Our experiment begins with a sample of 200 numbers, picked randomly from the range 2000-6000. Every 10 minutes, the Google news search queries are scraped for each number, with the resulting HTML being saved to a database. This experiment ran for four days, giving us a temporal view of search results from which we can analyse.

Our analysis consists of mapping the resulting HTML at each datapoint into the resulting keys that `NumericNewsFader` would produce, for different combining functions `f`. This results in a large array of records of the format $\langle$ `timestamp, query,` $\mathtt{f}_0, \mathtt{f}_1, \mathtt{f}_2, \ldots \mathtt{f}_n$ $\rangle$ where $\mathtt{f}_0, \mathtt{f}_1, \mathtt{f}_2, \ldots \mathtt{f}_n$ are the different combining functions defined in section 3.5.2. From this array of records, we simulate the creation of FDOs for each different query, at the start time. At each timepoint, we observe the number of keys that have changed since the start time. In figure 3.2 we show the plot the % of keys which haven't changed ("still alive") over time.

Experiment Conclusion

Figure 3.2 shows that the choice of different combining functions give us some control over the expiry time of keys under the `NumericNewsFader` scheme. We also observe a "steep decline" in the % of keys still alive. Whilst the degree of the decline partially depends on the combining function, the drop-off is extremely quick in any case. Nonetheless, the earliest time when 50% of the keys are destroyed is for the combining function `allHeadings`, which is around 64 minutes. This amount of time is fairly sufficient for a user-created object lifetime and indicates that the churn of Google news search results are a potentially viable self destructing data source.

3.7.4 `NumericNewsFader` Security

Considering *Intractable Retrospective Access*

Under this system, it's not possible to view the past search results for any Google news queries. This means that this method exhibits *Intractable Retrospective Access*.

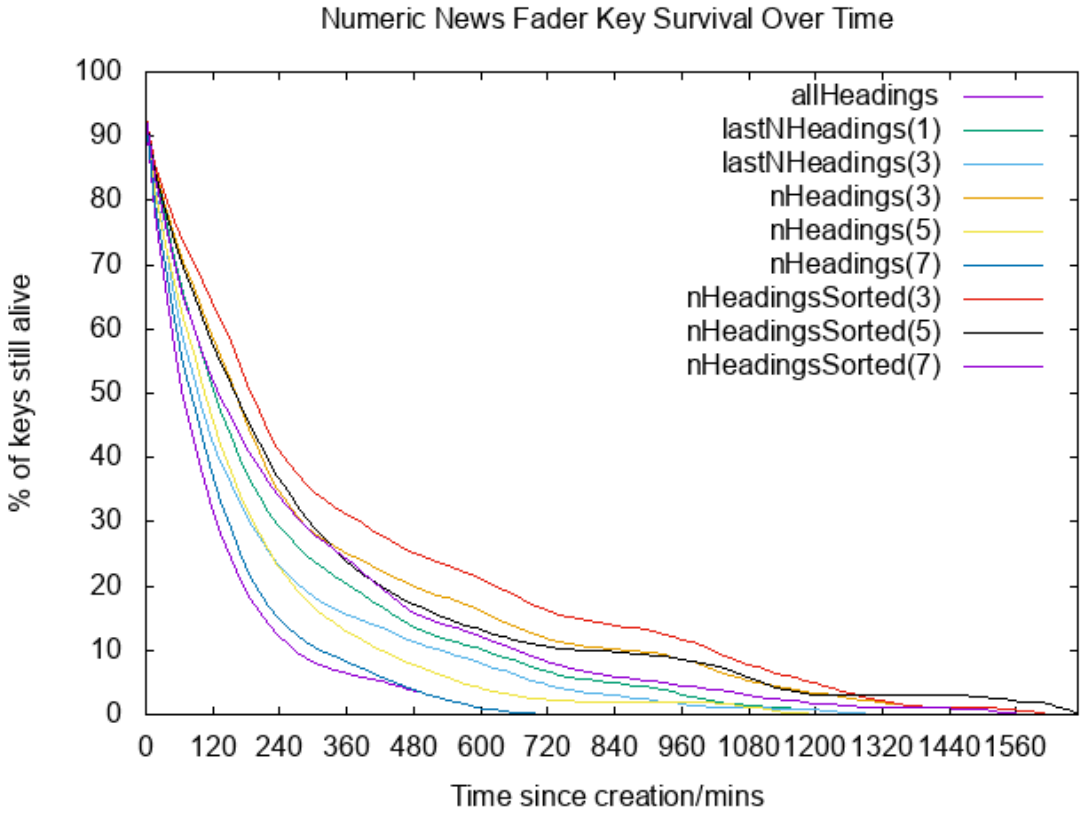

Figure 3.2: `NumericNewsFader` Key Survival Times for all Combining Functions

Considering *Infeasible to Proactively Scrape*

Since the range of numbers we use is just in the small range of 2000-6000, an attacker could scrape the news queries at a high rate fairly inexpensively.

Although unlikely, search engines are considered in our model to be adversaries. We must consider the threat of them saving their own search results in order to mount attacks on retrospective privacy. We will consider this threat in more detail later on.

3.7.5 `NumericNewsFader` Outcome

We have shown that `NumericNewsFader` gives rise to reasonable lifetimes, although we desire to understand them in more depth. However, most significantly, the scheme is vulnerable to a proactive scraping attack, which we seek to mitigate.

3.8 `NumericSequenceNewsFader`

In this section, we design and analyse a new SDD scheme, `NumericSequenceNewsFader` which is based on `NumericNewsFader`. Our main goals are to mitigate the scraping attack vulnerability present in `NumericNewsFader` and to do an in-depth analysis of the resulting object lifetimes.

`NumericSequenceNewsFader` is the same as `NumericNewsFader`, except that instead of queries that are random numbers in a pre-defined range, queries are sequences of random double digit numbers. An example of a query used by `NumericSequenceNewsFader` is 58 24 41 58 90 20. For clarity, we detail `NumericSequenceNewsFader`'s FDO construction algorithm in full below.

1. Generate a random query, q, a double digit sequence of length `L`.

2. Make a news search for query q and apply combining function `f`, to obtain the result to obtain the key, k.

3. Encrypt data using key `k`, into cipher text, `c` and construct FDO $\langle c, q \rangle$.

The FDO reconstruction algorithm is just the symmetric reverse as expected.

3.8.1 Motivation

`NumericSequenceNewsFader` creates FDOs using random queries from a large space. This significantly differs from `NumericNewsFader` which selects from a small range. With `L = 6`, there are 10^{12} different queries that an FDO could use. This substantially increases the financial cost of mounting a successful scraping attack. In chapter 4 we analyse the feasibility of such a scraping attack and the associated costs.

3.8.2 Object Lifetime Investigation

To investigate the lifetimes of FDOs created under this new scheme detailed above, we perform analysis from both an empirical and theoretical perspective and then compare them. Before beginning this analysis we must first define the term "lifetime".

3.8.3 Definition of Lifetimes

The crux of our analysis aims to understand the probability that a key (FDO) has a lifetime of at least time t. Subsequently, we must carefully set out the exact definition of *lifetimes*. Here we show two such ways of measuring object lifetime - Time To Change and Ultimate Expiry. We shed light on the problem of *Zombie Keys* and we subsequently conclude that for our purposes Ultimate Expiry, is the most suitable definition for our use case.

Example Data

Our data can be viewed as a temporal view of search result headings for many different search queries. Table 3.2 below illustrates what our data looks like for a specific search query q. Search results are represented as a list of letters representing the actual search result. For brevity, this example only includes one combining function, `nHeadings(3)`.

t	search results	nHeadings(3)
0	[A, B, C, D, E]	hash(A, B, C)
10	[A, B, C, D, E]	hash(A, B, C)
20	[E, G, H, I, J]	hash(E, G, H)
30	[E, G, H, I, J]	hash(E, G, H)
40	[E, G, H, I, J]	hash(E, G, H)
50	[E, G, H, I, J]	hash(E, G, H)
60	[E, G, X, Y, Z]	hash(E, G, X)

Table 3.2: Example Empirical Data.

Lifetime Definition: Time to Change

The most straightforward definition for lifetimes is the time it takes for a key to change. Using this, in the example above we observe two lifetimes - 20 minutes and 40 minutes. The 20 minute lifetime arises from when `hash(A, B, C)` changes to `hash(E, G, H)` at $t = 20$. The 40 minute lifetime arises from when `hash(E, G, H)` changes to `hash(E, G, X)` at $t = 60$. We call this definition of lifetimes Time to Change or just TTC in short.

Lifetime Definition: Ultimate Expiry

The issue with the TTC definition is that it doesn't take into account something that we term *Zombie Keys*; that is, keys that expire and then come back to life. A simple example of this happening is shown in table 3.3 below.

t	search results	nHeadings(3)
0	[A, B, C, D, E]	hash(A, B, C)
10	[A, B, C, D, E]	hash(A, B, C)
20	[A, B, C, D, E]	hash(A, B, C)
30	[A, B, D, D, E]	hash(A, B, D)
40	[A, B, C, D, E]	hash(A, B, C)
50	[A, B, C, D, E]	hash(A, B, C)
60	[E, G, H, I, J]	hash(E, G, H)

Table 3.3: Zombie Keys Example.

Above we can see that the key `hash(A, B, C)` expires at $t = 30$ and then comes back to life at $t = 40$. The definition of TTC would measure these changes as separate lifetimes which would add undesirable noise to our lifetime observations. It is worth noting here that Zombie Keys (such as those shown above) are not just a theoretical consideration; they pose a significant non-negligible issue that we observe in our datasets.

TTC is particularly lacking from a security analysis perspective as it would mean anything we conclude about the "lifetimes" of objects is misleading; if we were to conclude that after t minutes, 99.99% of keys have expired under the TTC definition, then it could still mean that *after* time t, a users object could come back to life, with a probability that is not captured by our model.

Thus, we define Ultimate Expiry so that the issue of Zombie Keys is taken into account. Put simply in words, we define Ultimate Expiry to be the time it takes for a key to change and *stay* changed for a reasonable amount of time. Clearly we cannot measure that keys never revert to an original state, which is why we wait for a "reasonable duration". For the above data in table 3.3, Ultimate Expiry would measure the lifetime of key `hash(A, B, C)` to be 60 minutes; i.e. ignoring the temporary flip to another key at $t = 30$.

Comparison of TTC and Ultimate Expiry

We now have TTC and Ultimate Expiry as two different definitions of lifetimes. We now compare these two definitions with respect to our desired probability analysis of "the probability that an FDO has a lifetime greater than t". For now, we will denote this probability as $P(Lifetime > t)$. For example purposes we'll use a fixed value of $t = 50$ minutes to compare the two definitions.

For TTC, $P(Lifetime > 50)$ tells us the probability that an FDO survives for at least 50 minutes. However, it tells us nothing about whether or not that key will expire and then come back to life. Effectively, this means that if we were to analyse the risk of an attacker being able to view an FDO after 50 minutes (i.e. violating retrospective privacy), the probability given is not useful. This is because it does not model the fact that keys come back to life.

Conversely, under Ultimate Expiry, $P(Lifetime > 50)$ tells us the probability that an FDO survives for a least 50 minutes *AND* will not come back to life in a reasonable amount of time. Naturally, this is more favourable from a security analysis perspective as it allows us to reason about violations of retrospective privacy. However, the definition of Ultimate Expiry also means that for all times before $t = 50$, the FDO is not necessarily alive for that whole period as there could be key flips in that time. That period is just the "time at which it hasn't yet changed and *stays* changed for a while".

Subsequently, we are presented with a trade-off in analysis; TTC tells us a lot about what happens *before* time t but does not take into account an object coming back to life after time t. Whereas

Ultimate Expiry takes into account what could happen after time t, but compromises by having a weaker assertion on what happens to an object before time t.

We choose to define lifetimes as Ultimate Expiry as this favours analysis from a security perspective. This is because when faced with the choice of compromising accuracy of our usability or security analysis, we prefer to compromise the former. Nonetheless, that is not to stay we can't still attain a complete understanding of the usability of object lifetimes under Ultimate Expiry. We believe that in choosing Ultimate Expiry, we model the underlying data more accurately, giving us better results.

In our data collection, we will assume the "reasonable amount of time" in the Ultimate Expiry definition to be the total time that we have that our data has observed. In other words, if we observe a key not changing back in the entire time span of our data collection, then we consider it to have "changed and not changed back". This is the strictest time that we can assert given the data we have. In section 4.3.3, we will investigate how the subtleties of Ultimate Expiry affect our results.

3.8.4 Empirical Analysis

Experiment Setup

We run an experiment similar in nature to the one for `NumericNewsFader`, except this time we use double digit sequence queries. As stated above, we set `L = 6`.

We begin with a sample of 100 different queries, each a random sequence of double digits, of length 6. Every 10 minutes, we scrape the Google search results for each query and record the keys produced by various combining functions, `f`. Note that this experiment gives per-query data identical to the format of the example data in table 3.2 above.

We then simulate the creation of many FDOs at random start times and observe the expiry times. In figure 3.3, we show how the % of keys still alive changes as time since object creation chances, per by combining function. To complement this, table 3.4 shows summary statistics for each combining function, `f`.

f	mean	median	std
allHeadings	42.7	30.0	33.4
nHeadings(3)	198.3	120.0	219.5
nHeadingsSorted(3)	263.5	149.0	343.8
nHeadings(4)	123.2	80.0	123.7
nHeadings(5)	94.1	69.0	83.5
nHeadingsSorted(5)	121.6	80.0	129.3
nHeadings(7)	64.7	49.0	55.7
nHeadingsSorted(7)	87.6	60.0	80.16
lastNHeadings(1)	80.8	59.0	71.9
lastNHeadings(3)	85.4	63.2	84.2

Table 3.4: Summary Statistics for `NumericSequenceNewsFader` , for different combining functions, `f`. All times are in minutes.

Analysis

From the figures above, we get a rough picture of behaviour of search results for different `f`. Across all `f` we observe an exponential-like decay as expected. We choose `nHeadings(3)` as the combining function for our subsequent analysis. This is because:

- `nHeadings(3)` has the longest mean average lifetime of all of the functions exhibiting our desirable steep decline. Due to the nature of our data collection where we poll every 10 minutes, a longer lifetime will result in more reliable analysis.

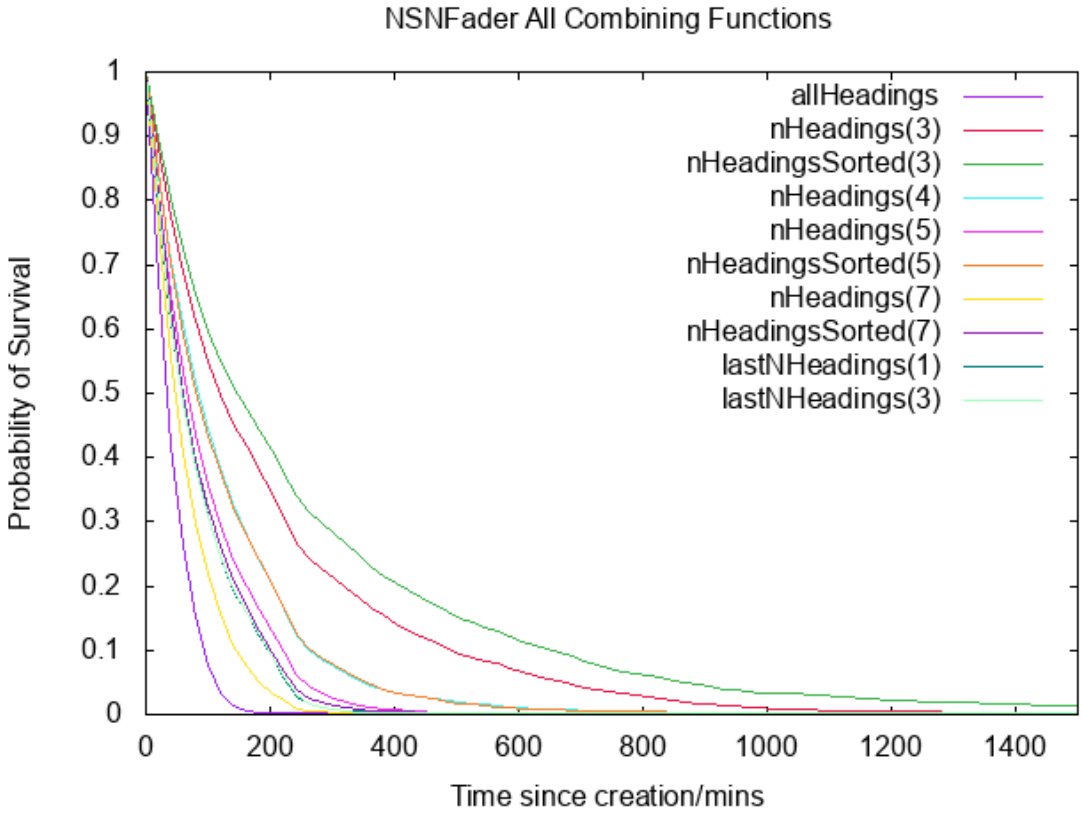

Figure 3.3: `NumericSequenceNewsFader` Survival Times for all Combining Functions

- We expect that there is a higher variance in functions that take into account more search result headings.

Note that despite our focus on `nHeadings(3)` hereon, all subsequent analysis can be applied just as well to any of the other functions above.

Modelling FDO Lifetime Probabilistically

With the above experiment giving us an idea of object lifetimes, we would like to understand them from a probabilistic perspective. We desire to have a theoretical model to complement the empirical data that allows us to calculate a simple estimate for FDO survival times of `NumericSequenceNewsFader`. This is particularly important from the perspective a real world application, where there is likely a desire for strong probablistic guarantees about the expiry times of FDOs.

FDO Lifetime as an Exponential Distribution

We now model the FDO lifetime, t, as a random variable, Y. Given that our previous graphs indicate exponential-like decay, we can model Y as an exponentially distributed random variable:

$$Y \sim \text{Exp}(\lambda)$$

Note that above we use the alternative parameterisation of the exponential distribution where $E[x] = \lambda$, with λ being the expected lifetime of a key, in minutes. From standard results, the maximum likelihood estimator for λ is:

$$\hat{\lambda} = \frac{1}{\bar{x}}$$

For combining function `nHeadings(3)`from table 3.4 above, we have the mean:

$$\bar{x} = 198.32$$
$$\therefore \quad \hat{\lambda} = \frac{1}{198.32}$$

Note that our mean value used here is aggregated across all queries (double digit sequences) in our experiment. In doing this, we implicitly assume that any two double digit sequences, q are probabilistically similar.

We subsequently derive equation 3.1 and 3.2 below:

$$P(Y > t) = e^{-\lambda u} \tag{3.1}$$

$$\begin{aligned} P(l < Y \leq u) & \\ &= P(Y \leq u) - P(Y \leq l) \\ &= (1 - e^{-\lambda u}) - (1 - e^{-\lambda l}) \\ &= e^{-\lambda l} - e^{-\lambda u} \end{aligned} \tag{3.2}$$

These can then be used with $\lambda = \hat{\lambda}_{\mathtt{f}}$, to calculate concrete probabilities.

Comparison of Theoretical & Empirical views of lifetimes

Equation 3.1 gives the probability of survival using our probabilistic model for our chosen combining function `nHeadings(3)`. Recall that from figure 3.3, we have a series that plots our empirical view of the probability of `nHeadings(3)`lifetimes.

We can proceed to compare these by plotting both our theoretical and empirical views on one graph for combining function `nHeadings(3)`. In figure 3.4 below we compare our theoretical view of $P(Y > t)$ for varying expiry time t with our empirical view.

This shows that our theoretical model can be used to fairly estimate the real underlying data. An issue with both series however is that the decline is not as steep as we would desire. We can see that at 60 minutes, there is just around a 60-70% chance that an FDO has expired. We desire for there to be some time at which the probability rapidly declines.

3.9 FADE - an improvement on `NumericSequenceNewsFader`

We desire to improve on the characteristics of `NumericSequenceNewsFader` shown above by modifying it to support some degree of redundancy. As it stands in `NumericSequenceNewsFader`, a user could create an FDO which is fairly likely to expire immediately and has little control over the lifetimes. In this section we show how we can modify `NumericSequenceNewsFader` to use a variant of Shamir's Secret Sharing to give users more control over FDO lifetimes. We outline the changes and then analyse how the new system is affected from both a theoretical and empirical sense, as before. We end up at a design for our final SDD scheme called FADE, which is described in full detail in appendix A.1.

3.9.1 Shamir's Secret Sharing, *SSS*

Recall that Shamir's Secret Sharing[15] (*SSS*, hereon) is a key splitting process that permits threshold encryption. SSS takes a key, k and splits it into n distinct parts (called "shares"), such that a minimum of m parts are required to reconstruct the original key k. As such, SSS is typically formulated as having parameters (n, m). In this formulation, $\frac{m}{n}$ is the fraction of key parts required to reconstruct the original key.

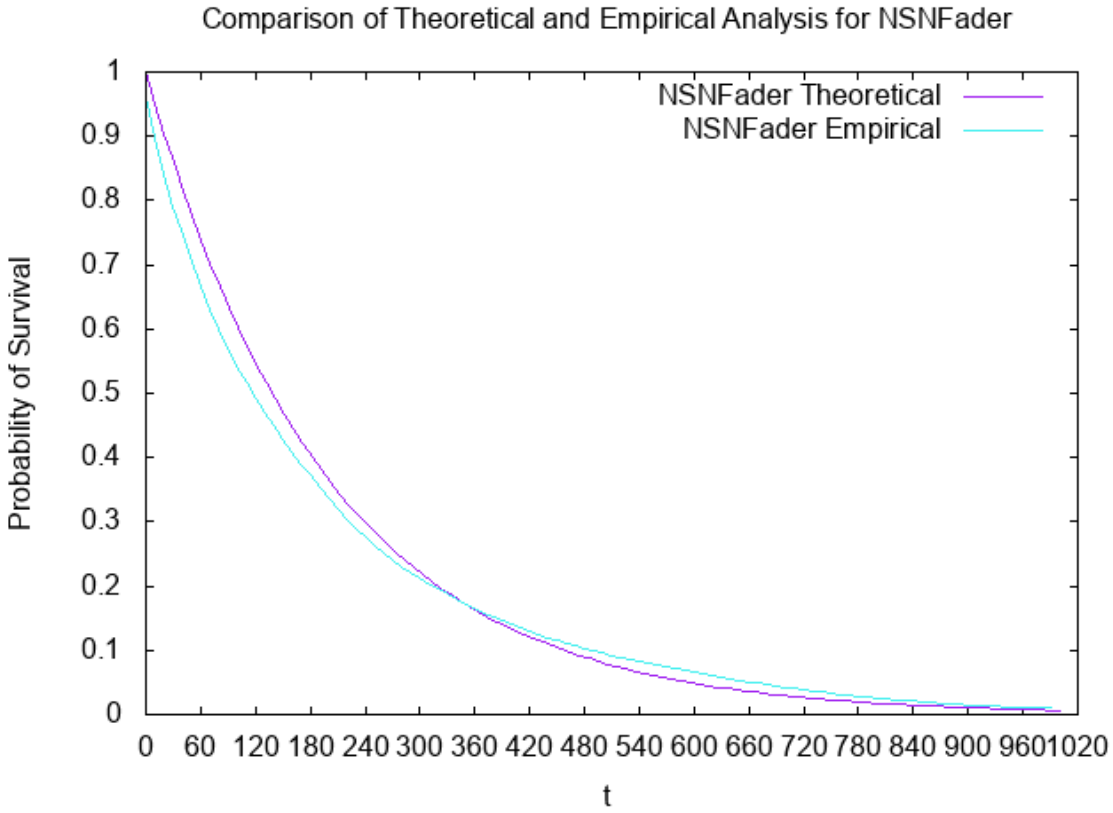

Figure 3.4: Theoretical vs empirical analysis of `NumericSequenceNewsFader` survival time probabilities over time

3.9.2 Shamir's Secret Sharing in `NumericSequenceNewsFader`

The principal idea of Shamir's Secret Sharing is of particular interest to `NumericSequenceNewsFader` as it can allow us to add some degree of redundancy in our object construction process. This would mean that even if search results change, it may still be possible to reconstruct the FDO, up to a certain time.

However, one difficulty of applying SSS to `NumericSequenceNewsFader` is that we desire something akin to a *reverse* application of SSS. In `NumericSequenceNewsFader`, we have key parts which are the results of applying `f` to different news search results pages. From here, we need a way to construct a final key, such that only m of those parts are required to reconstruct it. We cannot directly apply SSS here as it inherently requires us to start with a final key and then split it into parts, rather than start with the key parts and build it into a key. In other words, we require a process that resembles a reverse application of Shamir's Secret Sharing.

Figure 3.5 shows a typical application of Shamir's Secret Sharing (left) and our desired abstraction for adding redundancy to our existing scheme (right). The two diagrams together illustrate the difference between Shamir's Secret Sharing and the abstraction we desire to improve `NumericSequenceNewsFader`.

3.9.3 Reverse *SSS*

Below we describe our design of an abstraction that does a reverse application of *SSS* as described above. We call this process Reverse Shamir's Secret Sharing, hereon *RSSS*.

RSSS makes use of Shamir's Secret Sharing by generating shares that are encrypted with our desired key parts. The shares are created using a randomly generated key. This means that if the

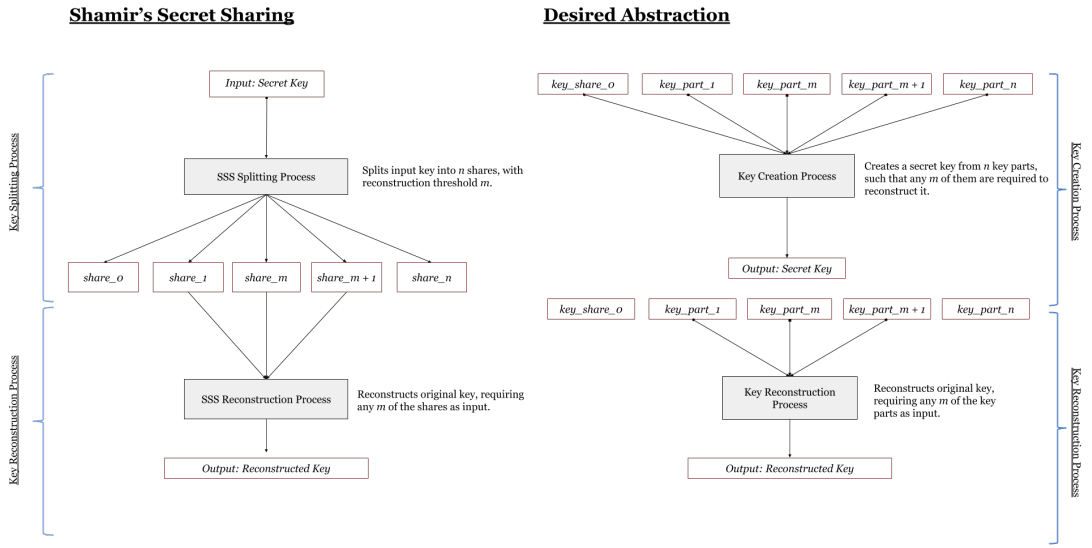

Figure 3.5: Comparison of Shamir's Secret Sharing (left) with our desired abstraction (right)

key parts are unavailable, then it is not possible to decrypt the shares to obtain the original key. This process is detailed in depth in 3 and 4 below.

RSSS consists of two algorithms - one for creating the key from the parts, `RSSS-CREATE`, and one for reconstructing the key from those same parts, `RSSS-COMBINE`. In the algorithms below, we assume the existence of the following procedures:

- `SSS-SPLIT(n, m, key)` - splits *key* into n shares with threshold m, using Shamir's Secret Shares. Returns the list of n shares.
- `SSS-COMBINE(shares)`- combines a list of *shares* back into a key, using Shamir's Secret Shares. Returns the combined key if successful, otherwise throws error. Only m out of the n *shares* need to be valid for the procedure to be a success.
- `ENCRYPT(plainText, key)` - encrypts given plaintext using key provided. Returns the ciphertext.
- `DECRYPT(cipherText, key)` - decrypts given ciphertext text using given key. Returns the cipher text.

Algorithm 1 RSSS-CREATE

procedure RSSS-CREATE($n, m, parts$) ▷ parts is a list of n key parts
    $key \leftarrow$ large random number
    $shares \leftarrow$ *SSS-SPLIT(n, m, key)*
    for i `from 0..`n **do**
        $shares_i \leftarrow$ *ENCRYPT*$(shares_i, parts_i)$
    return key, shares

Algorithm 2 RSSS-COMBINE

procedure RSSS-COMBINE(*parts, shares*)
    for i `from 0..`n **do**
        $shares_i \leftarrow$ *DECRYPT*$(shares_i, parts_i)$
    $key \leftarrow$ *SSS-COMBINE(shares)*
    return key

3.9.4 FADE - a thresholded Variant of `NumericSequenceNewsFader`

We now improve `NumericSequenceNewsFader` using the RSSS abstraction described above. We first modify the object creation phase as such:

1. Generate `n` random queries, q_i, each a double digit sequence of length `L`. Store all the queries in a list, `queries`.
2. For each q_i, query a news search engine and apply combining function `f`, to obtain n key parts.
3. Call `RSSS-CREATE(n, m, parts)` where `parts` is the list of n key parts created in step 2. above. `RSS-CREATE` returns a key, `k` and a list of encrypted shares, `encryptedShares`
4. Encrypt data using `k`, into cipher text, `c` and construct FDO $\langle c, encryptedShares, queries, m\rangle$

The underlying data can be reconstructed by first making queries for every element in `queries` and applying `f` to the results. Then, `RSSS-COMBINE` can be used to obtain the key that was used to encrypt `c`. Finally, that key is used to decrypt `c`, giving the original data.

Note that as the FDO contains `queries`, of length `n` and also the threshold value of `m`, a future reader knows they only need to make `m` queries that result in successful decryption of the corresponding encrypted share. We name this new thresholded scheme FADE, which we describe in full detail in appendix A.1.

3.9.5 Theoretical Analysis of FADE

Previously, in `NumericSequenceNewsFader`, we had just one key, whose lifetime directly determined the FDO lifetime. In FADE, we have `n` key parts, all each with their own lifetimes. Recall that when more than `m` of the key parts are no longer accessible (due to churn of search results), the FDO expires.

In keeping with notation, we use Y as the random variable representing the lifetime of an FDO under FADE. However now instead of one key that determines the FDO lifetime, we have n keys. Thus, we introduce `n` random variables, X_i, each of which represents the lifetime of key part i.

We then have:

$$\forall i: \quad X_i \sim \text{Exp}\,(\lambda)$$

Probability of Successful Reconstruction of a Thresholded FDO

For successful reconstruction of the FDO at time t we require that *at least* `m` out of the `n` key parts have survived until time t.

We first observe that the number of key parts alive at time t is also a random variable. Let Z be a stochastic process, parameterised by a time t, that represents the number of key parts that have survived until time t. We can then formulate:

$$P(Y > t) = P(Z_t \geq m)$$

We observe that:

$$Z_t \sim \text{B}\,(n, P(X_i > t))$$

Where `n`, is the "number of key parts" parameter of the FADE construction algorithm.

Therefore, the probability that at least `m` key parts survive until at least time t can be written as:

$$P(Y > t) = P(Z_t \geq m) = \sum_{k=m}^{n} \binom{n}{k} P(X_i > t)^k (1 - P(X_i > t)^{n-k}$$

Substituting $P(X_i > t) = e^{-\lambda t}$ gives:

$$P(Y > t) = \sum_{k=m}^{n} \binom{n}{k} (e^{-\lambda t})^k (1 - (e^{-\lambda t}))^{n-k} \tag{3.3}$$

Desirable Lifetime of a Thresholded FDO

Our expression for $P(Y > t)$ above calculates the probability that an FDO survives greater than time t. We also desire an expression that gives the probability that a thresholded FDO has a desirable lifetime - that is, a lifetime in the range between l and u.

By subtracting probabilities we get:

$$P(l < Y \leq u) = P(Y > l) - P(Y > u)$$

$P(Y > t)$ occurs when at least `m` of the key parts survive until time u, which we can write as $P(Z_t \geq m)$. Recall that from equation 3.3 above we have:

$$P(Z_t \geq m) = \sum_{k=m}^{n} \binom{n}{k} (e^{-\lambda t})^k (1 - (e^{-\lambda t}))^{n-k}$$

Therefore

$$\begin{aligned} P(l < Y \leq u) &= P(Y > l) - P(Y > u) \\ &= \sum_{k=m}^{n} \binom{n}{k} (e^{-\lambda l})^k (1 - (e^{-\lambda l}))^{n-k} - \sum_{k=m}^{n} \binom{n}{k} (e^{-\lambda u})^k (1 - (e^{-\lambda u}))^{n-k} \\ &= \sum_{k=m}^{n} \binom{n}{k} \left[\left((e^{-\lambda l})^k (1 - (e^{-\lambda l}))^{n-k} \right) - \left((e^{-\lambda u})^k (1 - (e^{-\lambda u}))^{n-k} \right) \right] \end{aligned}$$

Figure 3.6 shows the plot of our theoretical results for $P(Y > t)$ (equation 3.3 above) over varying t, for different configurations of m, n. To make each line series comparable, the ratio of $\frac{m}{n}$ is kept at a constant of 0.2. In our computation of $P(Z_t \geq m)$, we use $\lambda = \hat{\lambda}_{\texttt{nHeadings(3)}} = \frac{1}{198.32}$.

We can immediately see from this that we have created a large window for which there is a high probability that an FDO can be reconstructed. In addition to this, there is now also a steep decline. This indicates an improvement from figure 3.4 where there is a shallow drop off and a small amount of time where the probability of reconstruction is high. The graph above therefore serves as evidence that using a thresholded approach theoretically improves the system. We also generated a similar graph for various different $\frac{m}{n}$. To summarise the effects of varying $\frac{m}{n}$, we fix $n = 50$ and vary the ratio $\frac{m}{n}$. These results are plotted in figure 3.7.

As $\frac{m}{n}$ increases, the decline in survival probability becomes steeper because more key parts are needed to reconstruct the original key. This graph shows us a clear trade-off between our desirable *steep decline* characteristic and expiry time; as desired expiry time, t increases, there is less of a steep decline, indicating a longer window when an attacker could attempt to reconstruct the object. In chapter 4 we will examine this trade-off in more detail.

We have now shown that a thresholded approach can improve the time for which an FDO can be read with high probability, as well as make the decline in probability steeper. We now proceed to perform some empirical analysis and then compare it to our theoretical model.

3.9.6 Empirical Analysis of Fade

Now we have our theoretical model for Y (equation 3.3), we can run an empirical analysis on our data and then compare how closely it reflects our theoretical model.

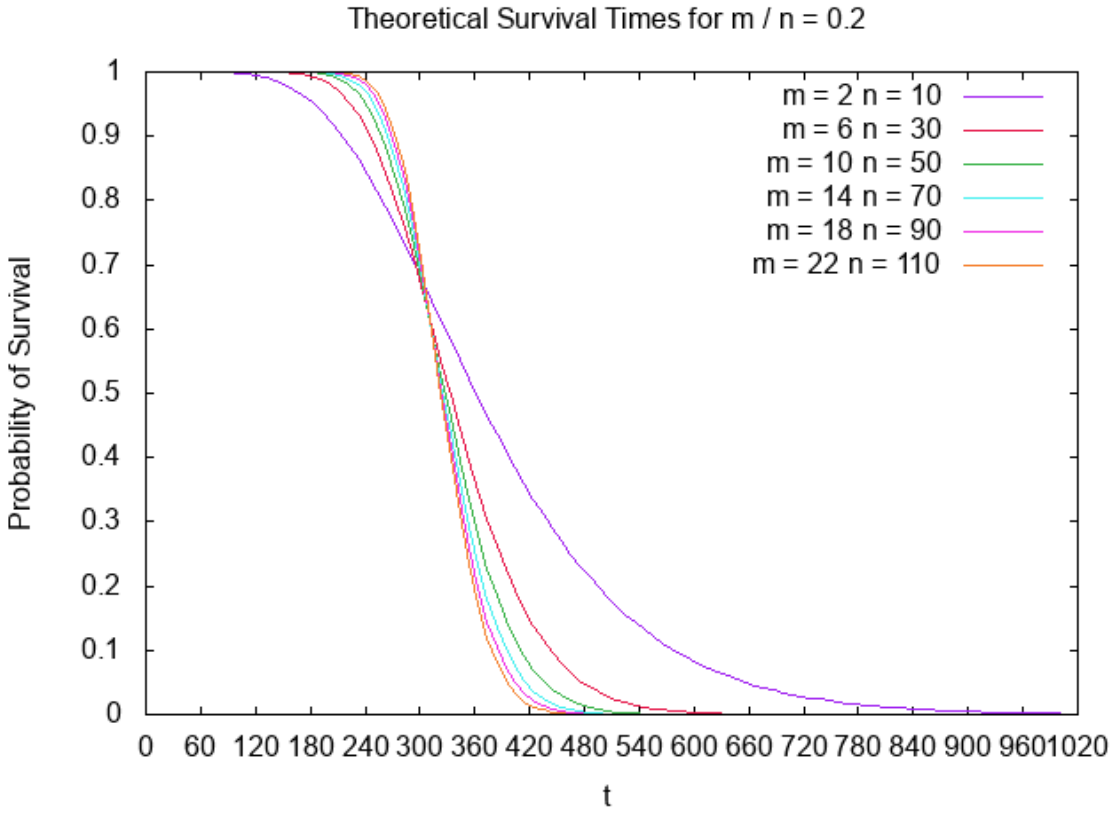

Figure 3.6: FADE theoretical survival times for $\frac{m}{n} = 0.2$.

Recall that our raw data collected from `NumericSequenceNewsFader` are temporal records of the form $\langle$`query, timestamp`$, f_0, f_1 ... f_n\rangle$. This gives us a replay log of news search results for 100 double digit search queries.

Subsequently, we simulate the creation of FDOs at random times, using the FADE scheme. In this simulation, one trial consists of choosing n double digit queries at random and then examining the replay log to find what the expiry time[4] would be of the FDO made up of these queries. That is, the time at which there are less than m search queries whose search result has changed.

In our experiment, we ran $100,000$ of such trials, for our chosen combining function `nHeadings(3)`. We conduct this by varying time, t and plotting the probability that a randomly picked trial has expired at that time. We do this for different configurations of m, n, where $\frac{m}{n} = 0.2$.

In figure 3.8, we show the results of this experiment. The results are an empirical equivalent to our theoretical plot for $P(Y > t)$ in figure 3.6. Figure 3.9 shows a comparison of our empirical results above and the theoretical results shown in 3.6 for varying m, n.

[4]Lifetime is defined as per the description of ULTIMATE EXPIRY in section 3.8.3.

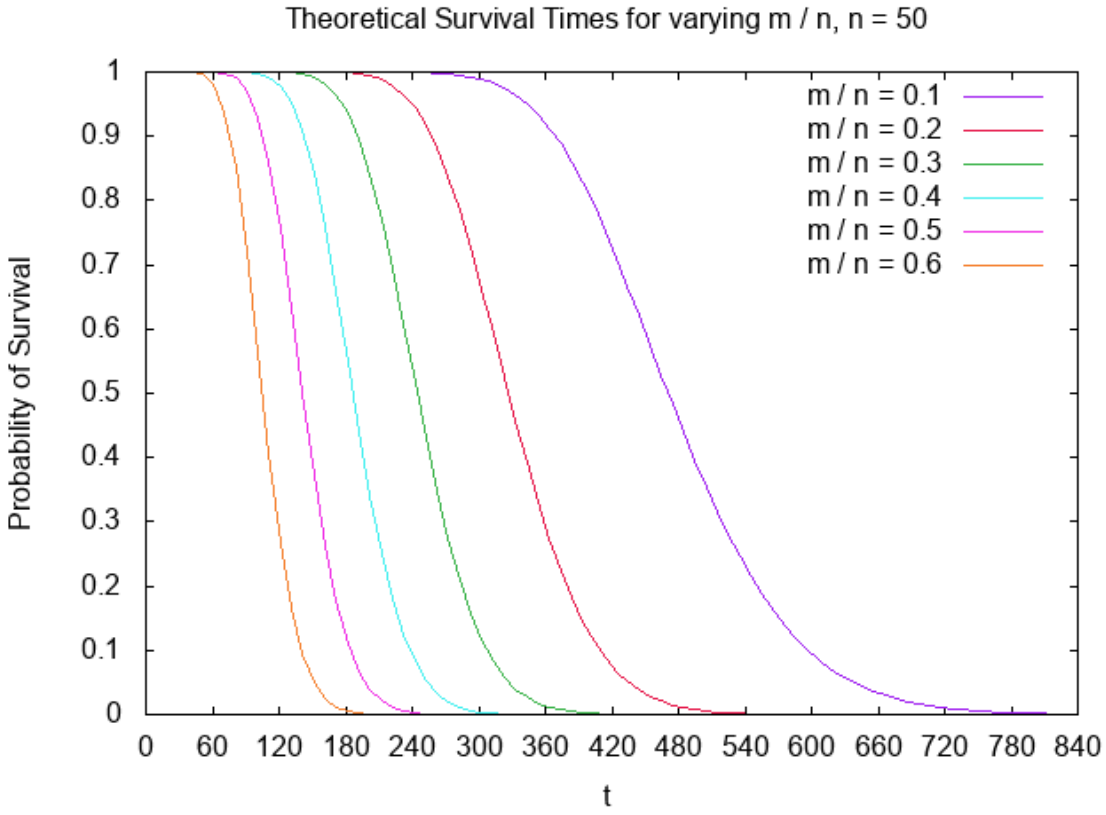

Figure 3.7: FADE theoretical survival times for varying $\frac{m}{n}$ with $n = 50$

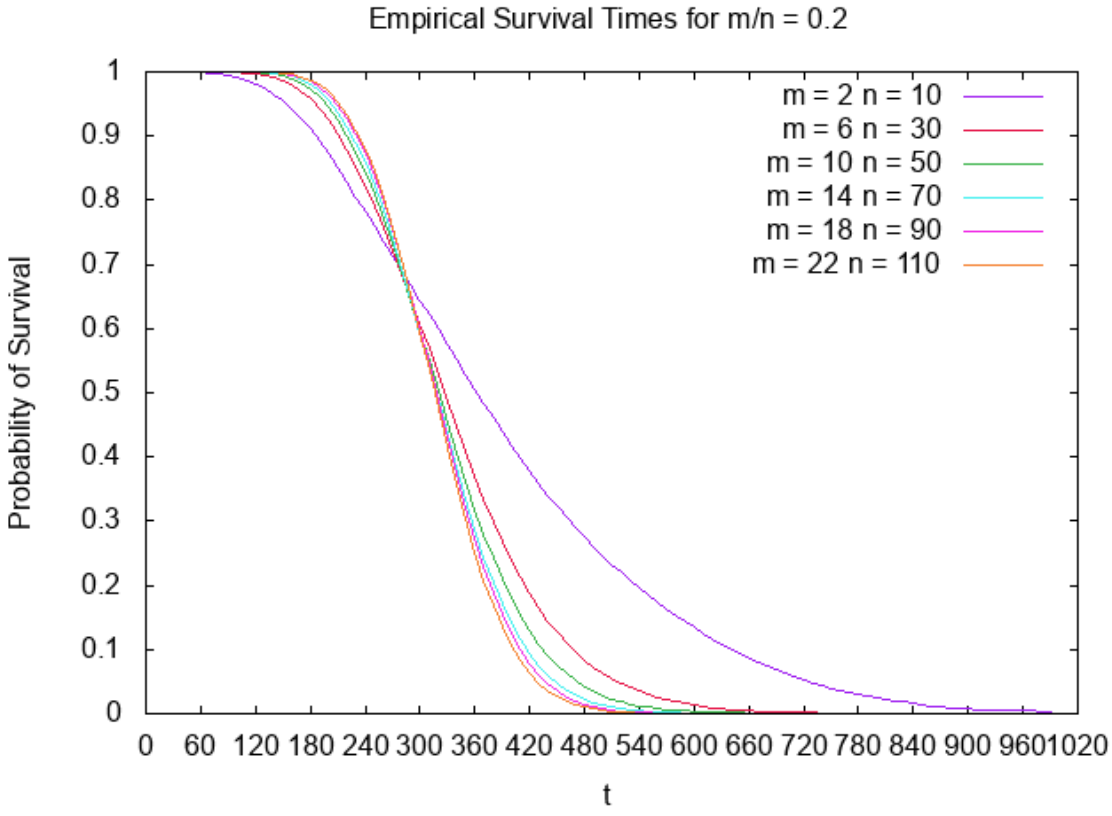

Figure 3.8: Empirical Survival Times for $\frac{m}{n} = 0.2$.

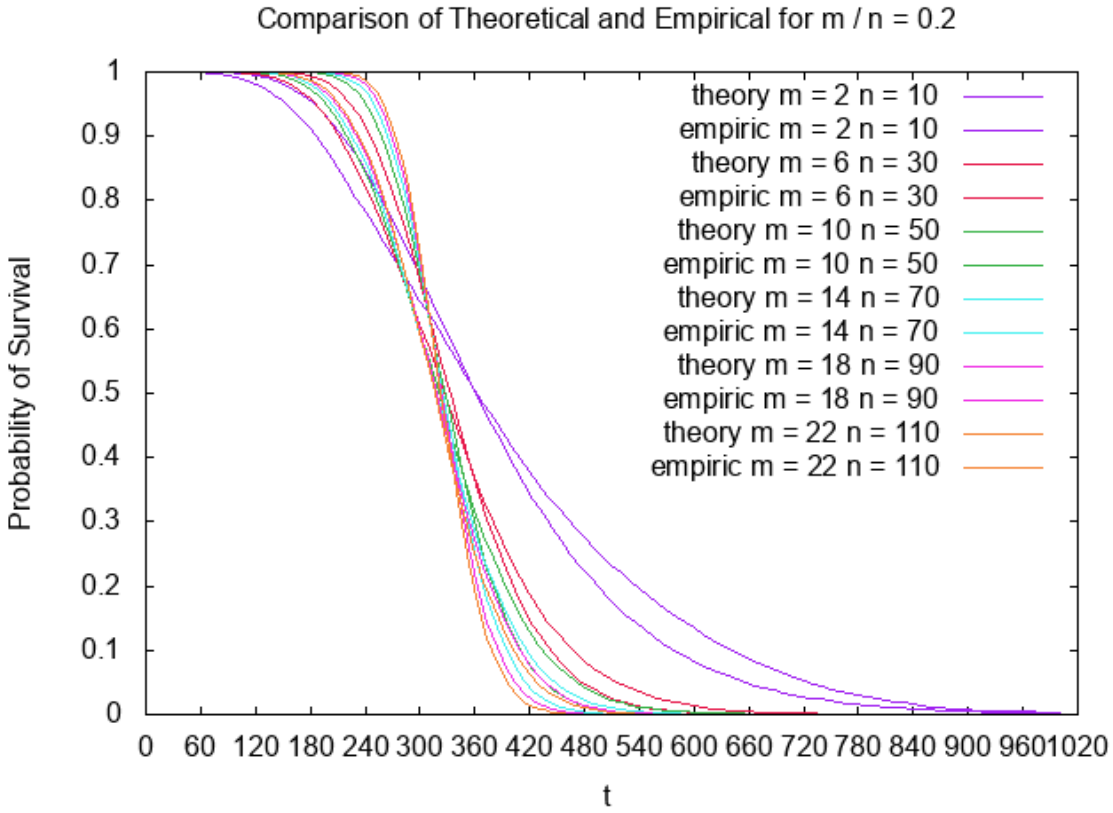

Figure 3.9: Comparison of theoretical and empirical analysis of FADE survival time probabilities. Series of the same colour indicate theoretical and empirical equivalents. Ratio $\frac{m}{n} = 0.2$.

Figure 3.9 shows that our theoretical model reflects our empirical view of the data to a fairly accurate degree. The differences between the theoretical and empirical graphs are likely to do with the variance of expiry times.

3.10 Summary of FADE Analysis

We have now concluded our definition and analysis of our final proposed SDD scheme, FADE. For completeness, a full definition of FADE is provided in appendix A.1.

In our analysis, we've shown that the empirical and theoretical analysis of the scheme are similar, although bear some differences. As we will see in the next chapter, an end product system needs to generate FDOs with user-inputted lifetimes . For this, we can choose to make an estimate from either our empirical analysis, or theoretical analysis. For simplicity, we opt for the latter as the theoretical analysis provides formulas that can be used to give us immediate results.

In the next chapter, evaluation, we use the above results to evaluate the FADE system in the contexts of security and feasibility of real world deployment.

Chapter 4

Evaluation

In this chapter, we will evaluate FADE from the perspectives of *security* and *feasibility of real world deployment*. We also discuss the strengths and limitations of both our theoretical and empirical analysis which serve as the foundation of everything we have shown.

We end this chapter by placing our contribution of FADE in the context of related work, most significantly, Neuralyzer.

Before continuing, we note that our final SDD scheme, FADE, can be generally parameterised by the tuple $\langle searchEngine,\ L,\ m\ ,\ n \rangle$, defined as follows:

- `searchEngine` - the search engine that is used to make news search queries.
- `L` - the length of the numeric double digit sequence query, e.g. the query "15 78 42 93 58 21" has `L` $= 6$. In all our prior analysis and subsequently our following evaluation, we use $L = 6$.
- `f` - the combining function that is used to turn search results into cryptographic keys. In all our prior analysis and subsequently in our following evaluation, we use `f = nHeadings(3)` as detailed in section 3.5.2.
- `m` - threshold parameter; the minimum number of key parts required to reconstruct an FDO.
- `n` - the total number of key parts that form part of an FDO. The ratio $\frac{m}{n}$ denotes the % of keys required to reconstruct an FDO.

We must also constantly have in mind that even though there exists multiple configuration parameters in FADE, an end user of our SDD scheme only has a desired expiry time, t. No other knowledge of the system should be required. As such, we view this desired expiry time t as an input to a creation of an FDO in the final system, where the FDO is configured with parameters m and n that are most appropriate to suite the desired t.

4.1 Security

In this section, we will evaluate FADE from a security perspective. Here, we attempt to provide an exhaustive evaluation of security risks specific to our SDD scheme. For each security risk, we either analyse the risk to show it is not a concern or suggest future work that would help mitigate it.

4.1.1 Agents in Adversarial Model

We define the following agents in our formal adversarial model:

- Sender, S - an honest agent that creates an FDO and sends it to a receiver, R, through a public communication channel[1].

[1] Public here could mean a social networking site, blog post, or monitored emails/private messages.

- Receiver, R - an honest agent that receives an FDO, sent by a sender S.
- Attacker, A - a dishonest agent that wants to maliciously reconstruct a specific FDO *strictly after* it's desired expiry time.

4.1.2 Usability vs. Security Tradeoffs

We observe two simple desired properties of our system:

1. Up to some time *latestReadTime*, anyone can read the FDO with high probability, α.
2. After some time *latestAttackTime* there is an extremely low probability, β that an Attacker (or anyone) can read the FDO.

We can subsequently create two formal constraints on *latestReadTime* and *latestAttackTime*:

1. $P(Y > latestReadTime) \geq \alpha$
2. $P(Y > latestAttackTime) \leq \beta$

This is to say that the probability of successful reconstruction up to *latestReadTime* should be at *least* α (1. above). Similarly, the probability of successful reconstruction after $t = latestAttackTime$ should be at *most* β (2. above).

We illustrate this interplay graphically by setting $\alpha = 0.9, \beta = 0.05$ and observing the resulting values of *latestReadTime* and *latestAttackTime*. Our plot below uses an $m = 2, n = 10$.

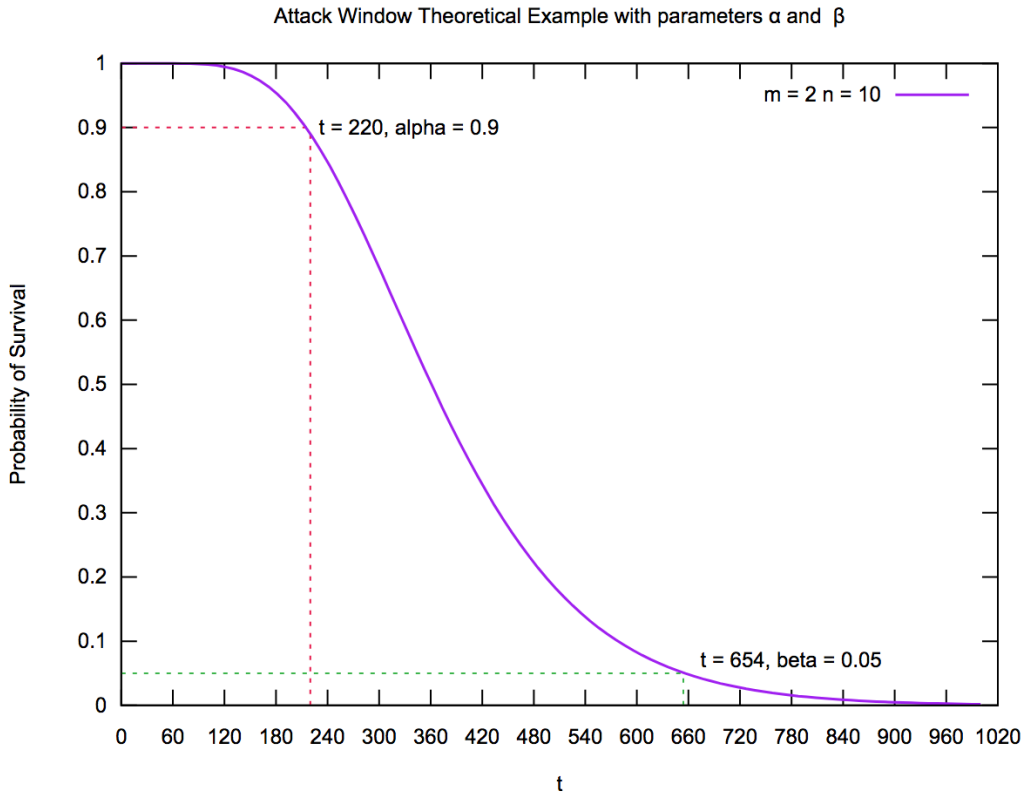

Figure 4.1: Example Theoretical Attack Window for α, β and with $\frac{m}{n} = 0.2$. The red dashed line represents *latestReadTime* and the green dashed line represents *latestAttackTime*.

In practice, we suggest that α should be a high value, such as 0.9. This means that there is a 90% chance of reading an FDO before the desired *latestReadTime*. On the contrary, β should be a low value, such as 0.05. This means that there is just a 5% chance of an attacker reading an FDO after *latestAttackTime*. We observe that this makes α a *usability* parameter of sorts and β a security parameter.

Presenting Lifetimes to End Users

Our theoretical model can be combined with the constraints above to present an interactive, configurable experience to end users. We propose that in an end user-facing product, users should only be required to enter their desired *latestReadTime* and *latestAttackTime*. From there, the system will use sane defaults for α and β[2] and will search for the best m, n values that satisfy the constraints.

The system would then create an FDO with the appropriate m, n. The values of m and n can be calculated using the expression for $P(Y > t)$ shown in equation 3.3. It is worth noting that there are often multiple solutions for m, n that satisfy the constraints. In this basic mode, the m, n picked should be chosen such that n is smallest. This has the effect of minimising the time cost of FDO construction and reconstruction.

Should the user want more control over the trade-off between usability and security, there could be an advanced mode where users can enter all of the parameters: *latestReadTime*, *latestAttackTime*, usability parameter α and security parameter β. The system could then create an FDO with a suitable m, n configuration. In this advanced mode, the user could also be presented with an interactive graph of probability over time, allowing them to input parameters and immediately see the results. Users could be presented with a selection of m, n options (where applicable) as well as an estimate of FDO reconstruction times to help them choose which particular m, n solution to use. We note that for a user in advanced mode of operation, they are likely to care more about the FDO reconstruction time than the FDO construction time; the FDO is constructed only once but could be reconstructed many times once published in the public domain. In section 4.4.4 we look at how the choice of n affects the FDO construction and reconstruction times.

Such setups as shown above give users a *well-defined* idea of the FDO lifetimes as well as a reasonable degree of control over them. In section 4.4 we examine what a final product would look like in more detail.

We have written a script that performs the calculations described above. We show example outputs from the script in table 4.1 below for varying *latestAttackTime* inputs (denoted t in the table), with a fixed *latestReadTime* of 200 minutes. We set $\alpha = 0.9, \beta = 0.05$.

t	m	n
300	26	86
360	12	43
420	7	27
480	5	20
540	3	13
600	3	13
660	2	10
720	2	10
780	2	10

Table 4.1: Example outputs of m and n for varying *latestAttackTime*, t measured in minutes. *latestReadTime* is fixed at 200 minutes with $\alpha = 0.9, \beta = 0.05$.

4.1.3 Proactively Attacking FDOs

Our attacker, A could attempt to attack the system proactively in two different ways:

- **Deletion Attack** - An attacker could perform an availability attack on a wide variety of FDOs if they are able to influence search results for the queries held in their information tables. This equates to what is termed a *premature deletion attack*.
- **Proactive reading of FDOs** - A could attempt to violate retrospective privacy by proactively seeking to read *all* FDOs in the public domain, before their desired expiry time. In this

[2]E.g. $\alpha = 0.9, \beta = 0.05$

attack, A only chooses a specific FDO to attack *after* its desired expiry time. This means that our adversarial model does not permit such an attack.

Feasibility of Premature Deletion Attack in FADE

Schemes such as Vanish and Neuralyzer employ mutable data sources - that is, data sources that users (and most third parties) can influence themselves. In the case of Vanish, this is the Vuze DHT and for Neuralyzer it is the caches in DNS infrastructure. Whilst our data source of search engine results are not intuitively *mutable*, it is still plausible that an attacker could mount a successful attempt to modify the search engine results for specific search queries. We speculate this could be done for example, by employing a script that posts the numbers used in the queries in the comments pages of popular news websites. Since we have no guarantees over the way that search engine algorithms work, this kind of deletion attack can be considered feasible.

Cryptographic Puzzles In FADE

We observe that Amjad et Al. work on *Forgetting with Puzzles*[19] (summarised in section 2.7) shows how Cryptographic puzzles can be used to increase the costs of a scraping attack. It is fairly straightforward to apply the work to FADE. Recall that their key innovation is to protect the information tables of a data object with Rivest-style cryptographic puzzles of varying difficulties. The information tables are called τ_1 and τ_2 and are protected by puzzles termed ρ_1 and ρ_2 respectively.

Forgetting with Puzzles assumes there can be multiple information tables in a single data object, which is not something we have by default in FADE. Whilst this concept is natural for Vanish and Neuralyzer as they can store the keys in distinctly different locations, in FADE we do not have such control over the keys and where they are stored.

However, we could modify our scheme so that our plaintext is encrypted with two keys, k_1 and k_2 such that only one of the keys is needed to decrypt the plaintext. This could be achieved through a application of Shamir's Secret Sharing[15]. Then, k_1 and k_2 can each be encoded by their own distinct information tables through the standard FADE threshold method - Reverse Shamir's Secret Sharing would be applied to each key to generate the key part and then encrypted with search query results. This gives us two information tables that are equivalent to the τ_1 and τ_2 above; accessing the information in any one of the two tables gives us either k_1 or k_2, which can be used to decrypt the plaintext. As per Ajad et al's protecting mechanism, the two tables in the FDO would each be protected by their own Rivest cryptographic puzzle, ρ_1 and ρ_2.

Modifying the scheme as described above increases the cost of both the *Premature Deletion* and *Proactive Reading of FDO* attacks. However, we must bear in mind that it comes at the expense of performance for genuine users of the system, as they also need to solve the puzzles to access the data.

4.1.4 Proactive Scraping of Search Results

We now consider an attacker who attempts to scrape and permanently store search engine results in order to be able to reconstruct a specific FDO in the future, after search results have changed. We begin with an example and then go on to show that for the Google search engine, a scraping attack on just 10% of possible queries is infeasible both technically and financially.

Example Attack

1. At $t = 100$, S creates FDO with $m = 1, n = 1$ (we use these just to keep the example small). The resulting single query, q in the information table of the FDO is "12 20 97 23 26 32". S publishes the FDO in the public domain and stipulates that it should not be accessed after time $t = 200$.

2. Attacker A deploys a script that scrapes search results for all possible queries in the space *AA BB CC DD EE FF*, where each letter pair is a number between $0 - 99$. Note that the query stored in the FDO is included in this search space. The query results are stored in

a simple database. We say that A has computing and financial resources so that they can scrape each query in this space at a rate of r per hour.

3. At $t = 220$, the search results for query q change, meaning that S and any other honest agents can no longer reconstruct the originally published FDO.

4. At some later time (it could be a day, or many years), some social/political event occurs which makes the destructed FDO of significant interest to A. A looks through the database of scraped data and tries to find if there is a result for "20 97 23 26 32". If there is, then A uses it to decrypt the FDO and subsequently violates retrospective privacy.

Financial Feasibility of Result Scraping

The attack described above hinges upon r, the average rate at which A can scrape each query. We now perform some rough calculations to analyse the cost of such an attack. In doing this, we make calculations specific to Google Search as this is the search engine used by all of our existing data.

Scraping and storing search results is something we are familiar with from our empirical analysis. To give an intuitive idea of the costs of scraping, we can look at our empirical analysis, where we used a popular cloud hosting provider.

Our deployment scraped 100 Numeric News Sequence queries every 10 minutes. After 5 days, this was at a financial cost of around £140[3]. We can reduce to say that it costs roughly 28p a day to scrape one query, every 10 minutes.

There are a trillion possible queries, which means we scraped a mere 0.00000001% of the entire space over 5 days. To scrape just 10% of the query space, our attack A would need to be be making 100,000,000,000 queries every 10 minutes. If we assume computing costs scale linearly[4], then this would cost an attacker roughly £28B per day. To reduce the cost af an attack, the attacker could make queries at a smaller rate than every 10 minutes - say, every hour. This would make the attack cost something more like £4.6B per day, which is still a suitably challenging quantity.

We have now shown it would cost around £4.6B for an attacker to get a maximum of a 10% chance of having retrospective access to a query *after* it has changed. Even then, this calculation is only for an attacker that scrapes over a period of one day; for five days, for example, it would be around £23B.

Whilst the calculation here considers the resources required for an attacker to a 10% of retrospectively accessing *one* query, it easily extends to when we consider it for FDOs with parameters m, n. The redundancy created by m reduces the cost for the attacker, as they just need to have $\frac{100m}{n}\%$ of the queries in the FDO scraped. However, it comes intuitively that even having a low ratio $\frac{m}{n}$, the attacker still has financial costs in the order of millions (likely billions) of pounds per day.

It's worth noting that a disadvantage of the calculations above is that they rely on the costs we experienced using a particular cloud hosting provider. Since we used a simple unoptimised scraping architecture, it follows that scraping cost could be made cheaper by a technical solution that focuses on reducing cost. However, we believe that our calculations provide a convincing argument that an optimised attack as such would still be infeasibly expensive.

Technical Feasibility of Result Scraping

If an attacker were financially placed to make queries at such at high volume, it would still be very difficult to do so from a technical perspective. We note that (cloud hosted or otherwise) the scraping attack would require a geographically distributed cluster of scraping and storage nodes. Since they are making requests to a service owned by a search engine (in our case, Google), the

[3]We did use a free trial!

[4]This is a generous assumption that favours the attacker as larger scale scraping would require more nodes and a larger database cluster.

process of scraping on this scale can be likened to an attempted DDoS[5] attack. Although we are not in control of how particular search engines accept and reject traffic, it's fair to assume that attempting to scrape on such a large scale would result in the activation of the search engine's spam filters and anti DDoS systems.

It is worth nothing here, that even when performing the extremely small scale (tracking 100 queries) scraping ourselves for our empirical analysis, we found that scraping rate any more than once every 10 minutes resulted in the activation of Google spam filters.

Considering Search Engine Companies as the Scrapers

It would be fairly straightforward for search engines companies to detect and block queries made by the FADE scheme. At worst, they could archive the results of the queries which could then be used in a retrospective attack. To this end, we conclude that our system in its current incarnation does trust the search engine not to behave maliciously in this way.

We propose a simple extension to FADE that would significantly reduce this risk: instead of using queries on *all* the same search engines, we could modify FADE to make the queries across a variety of search engines. The FDO information table would then contain a list of query, search engine pairs.

For this to be possible, the encryption key would need to be generated such that each at least x of the key parts come from distinctly different search engines. A scheme as such could be created trivially by further combining Shamir's Secret Shares and our Reverse Shamir's Secret Sharing procedure presented earlier.

Ideally, the search engines used would be owned by separate entities and located in distinct jurisdictions to make legal escrow trickier. This way, a search engine company alone would not be able to perform this attack without corroborating with another search engine company. This significantly increases the barriers for a successful attack of this sort.

In its default form, we believe that it is acceptable for FADE to employ just one search engine as the risk of a search-engine company led[6] scraping attack is very low. Regardless, the modification we suggest above would substantially reduce this threat.

4.1.5 Correlation of Search Results over Time

One thing we do not consider in either our empirical or theoretical analysis is the correlation of search results over time. We imagine that after some key part has expired according to ULTIMATE EXPIRY, there is a time after for which the corresponding search query results are different yet related to search results of the past.

As our combining function `nHeadings(3)` hashes the first three search headings, the detail of how the ordering of those headings change over time is lost in our analysis. Subsequently, an attacker could proactively scrape search results and if they fail to reconstruct a key succesfully, could permute the ordering of search results to attempt to recover the ordering at the time of FDO construction. We suggest that the analysis of correlation of search results over time demands a thorough, standalone investigation that substantiates a body of future work.

4.2 Empirical Analysis

The success of our empirical analysis is indicated by the high degree to which it fits our theoretical model. However, there are several factors of it that could be improved to give us a better and more accurate understanding of the data.

[5] Distributed Denial of Service.
[6] Or similarly, an attacker led by a rogue actor within the company.

4.2.1 Fixed Scraping Interval

Our goal in scraping the search results is to get a *uniformly* randomly sampled view of search engine results over time, for a variety of queries. When we scraped the Google search engine, we did so at a fixed interval of once every 10 minutes (per query). This fixed interval is not ideal as it does not result in a *random* set of sample points; if we scrape at a fixed rate of every n minutes starting at time t, then the resulting timepoints at which we scrape simply follow a non-random, predictable sequence of $t, t+10, t+20, t+30 \ldots$ etc. The constant scraping interval increases the risk of non-negligible systematic distortion affecting our results.

To improve this, we could simple set the time interval between scraping to be a value drawn from an exponential distribution, with rate parameter λ. Then, λ could then be set such that *on average*, we scrape results every 10 minutes, yet for any given interval we wait a random amount of time. This would result in a set of sample points that are distributed uniformly randomly over our observation period, resulting in less systematic bias.

4.2.2 Volume of Data Collected

In our data collection, the volume of data we collect can be increased by either tracking more than 100 queries, or by increasing the amount of time for which we observe. While these changes both result in better quality data, they come at the cost of greater financial expense.

In our data collection for FADE, we tracked a randomly selected set of 100 queries over time. In collecting data for such a small proportion of the query space we bias our dataset to whatever is observed in just these 100 queries. From a scientific perspective, it would have been better to track around 1000 queries, albeit at increased cost.

Alternatively, we could have modified the FADE scheme to something where we could measure the search result churn for a high proportion of the query space. However, in doing this we arrive at a quandary whereby if we make it more feasible to collect a higher volume of data (and thus get a better view of the system) then it is also easier for an attacker to perform a proactive scraping attack!

4.3 Theoretical Analysis

4.3.1 Modelling Lifetimes as an Exponential Distribution

Evidently, modelling object lifetimes as an exponential distribution is just an approximation. With this said, our graphs show that our empirically collected data does follow an exponential distribution fairly closely.

In particular, we think that the lifetimes of news search results does not exhibit the memoryless property of an exponential distribution. The memoryless property means that the amount of time until the next event occurs is independent of the amount of time since the last event occurred.

For news search results, this is not the case as it is likely that certain weather, sports and stock market related articulates have some pattern in their publication times. This is one such reason why modeling lifetimes as an exponential distribution is inaccurate.

4.3.2 Usage of Maximum Likelihood Estimator

In all of our theoretical modelling, we use the maximum likelihood estimator (M.L.E) for an exponential distribution as $\hat{\lambda} = \frac{1}{\bar{x}}$ where $\bar{x}$ is the mean of the observed data set. Whilst this does result in our theoretical model matching up fairly well to our empirical results, it could be improved by using an estimator with less bias than the M.L.E[7]. This could be approached by

[7] M.L.E is typically bias as it uses the mean of a dataset, which is fairly sensitive to outliers.

investigating the application of a robust likelihood estimator that uses the median or a weighted likelihood estimator and seeing how this affects our results.

4.3.3 Definition of Lifetime as Ultimate Expiry

Recall that in section 3.8.3, we decided that lifetimes should be defined as per the description of Ultimate Expiry. As a recap, the definition of key lifetimes as Ultimate Expiry is the time it takes for a key to change and *stay* changed for a reasonable amount of time. As explained in section 3.8.3, choosing this definition means that all of our analysis favours the security of object lifetimes as opposed to the usability. This choice was primarily led by the effect that *Zombie Keys* have on the quality of our data collection.

Using our exmaple data for `f = nHeadings(3)`, we measured the average % of time that FDOs are "zombies"[8] in their lifetime to be 15%. This means that for FDOs with an expected lifetime of 100 minutes (for example), it will be available, on average, for around 85 of those. Whilst we would desire this percentage to be smaller, we deem this to be an acceptable trade-off between usability and security.

4.4 Feasibility of Real World Deployment

In this section we discuss how feasible it is to deploy Fade in practice. We believe that a browser extension written in JavaScript is the most fitting application for Fade. Subsequently we target browser-based JavaScript as the target of our analysis. For the hashing in combining functions, we use `sha256` and the encryption uses AES, as in Neuralyzer[1]. We start by discussing some real-world challenges of an implementation and then lay out how a final system could be implemented.

4.4.1 Locality of Search Engine Results

Modern search engines tend to localise results based on where the query comes from geographically. This poses a risk that some users of our system may not be able to reconstruct FDOs if they are not in the same geographical location of where it was created. For the Google search engine, we've found we can bypass the localisation by using the encrypted and private Google search functionality provided by http://encrypted.google.com. This is enough to mitigate the problem.

4.4.2 Spam Filter

As mentioned previously, the double digit number sequence queries used by Fade are distinctly robotic. This means that Google's spam filter blocks queries when they are made in high volume (i.e. large scale scraping).

For FDO construction with n up to 100 it is not so much of an issue. However, if a user attempted to reconstruct many FDOs in bulk (non-maliciously) over a short amount of time, for example, scrolling a Twitter where many tweets are FDOs, there would likely quickly arise a problem with the spam filter. We suggest that future work could attempt to mitigate this problem by using search queries across multiple search engines. This would reduce the number of requests sent to each search engine, thus lowering the risk that the search engine's spam filter will be activated.

4.4.3 Volatility of Search Engine Results in the future

Lifetime Analysis is Prone to Invalidation

As search engine algorithms change over time, it is inevitable that the concrete value of $\hat{\lambda}_{\texttt{nHeadings(3)}}$ found in our analysis will also change. This would detrimentally affect the accuracy of our theoretical and empirical models over time. To overcome this, we propose a high-level modification to the system that allows the model to self improve over time. A suitable adaptive model such as this would mean that FDO lifetime estimates stay fairly accurate.

[8] i.e. more than m of their key parts are flipped to another value

A Self Adapting Model

A self adapting system could involve a network of mututally-trusting users of FADE, who all share anonymised observations about some or all of the FDOs that they attempt to reconstruct. We observe that if a user fails to reconstruct an FDO at some time t after its creation, we learn at least *something* about the probabilistic nature of its lifetime in that it expired by the latest at some time t. Such observations could then be shared and built into a model that allows the system to give users, dynamic and self-improving estimates (likely λ) of FDO lifetimes as time passes.

Data Sharing Network

Such a shared network of data could involve a central exchange service or could be a decentralised network of sorts. We note that for the latter, it would be challenging to produce a system where the nodes do not trust each-other, as it would be difficult to detect when object expiry times are faked by a malicious agent.

Subsequently, we suggest that the task of investigating the prospect of a self-adapting model along with a data sharing network protocol could form the basis of some future work.

4.4.4 FDO Construction & Reconstruction Times

The time to perform FDO Construction and Reconstruction is dominated by HTTP requests. Recall that for FDO Construction and Reconstruction we make at most n HTTP Google News search queries.

Using a simple browser-based JavaScript program, we measured the time it takes to programatically perform n Google news search queries for varying n. The results of this are plotted in figure 4.2.

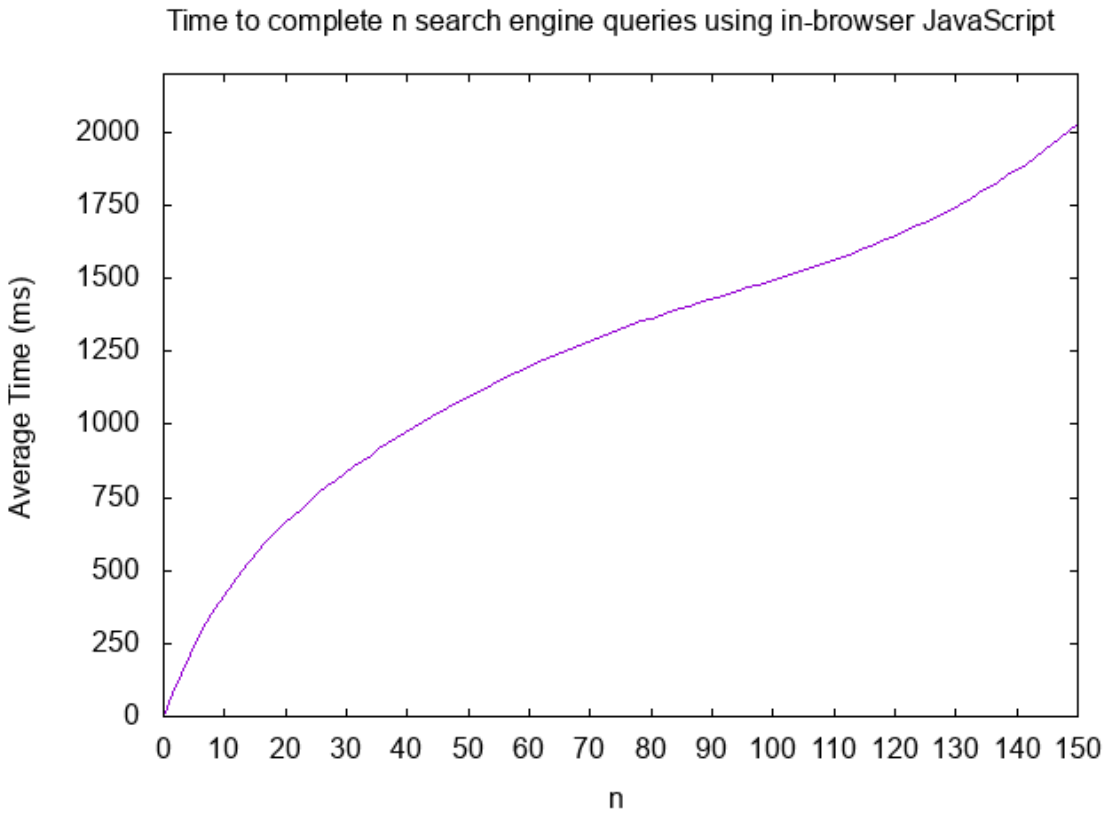

Figure 4.2: Time to complete n Google news search engine queries using in-browser JavaScript

This shows that an FDO can be constructed and reconstructed with a worst case time of around

two seconds for a value of $n = 150$. In practice, suitable values of n are < 100, showing that there is plenty of time leeway for things such as network delays, retries, etc. Additionally, whilst FDO reconstruction has a worst case bound of n, in practice, only around m search queries will need to be made. We deem this upper bound of 2 seconds to be an acceptable level of performance for an end user.

4.4.5 Implementation of User-Facing System

Implementation Difficulty

One of the principal advantages of FADE is that it lends itself to a straightforward and easily deployable implementation. As it only requires client-side cryptography and HTTP requests, this can all be implemented in a browser.

This is unlike Neuralyzer, which requires low-level DNS access to make queries to a specific resolver. The browser web extensions API now includes a `dns.resolve(domain)` method, although it only takes a domain as the argument and uses the operating systems built in DNS stack. This is not sufficient for Neuralyzer. Subsequently, a browser extension implementation of Neuralyzer would need to rely on another piece of middleware software running on the same machine, that exposes the required DNS functionality.

We believe that JavaScript is the best suited language for deploying FADE because:

- Browser Extensions can easily be written in JavaScript.
- JavaScript programs are easy to distribute over the web.
- The combining functions require manipulation of the DOM[9] search engine results pages. In the browser JavaScript environment, DOM manipulation is built-in to the language.

User Interface

For FDO construction, we propose that a user should be presented with two modes of operation, *Basic Mode* and *Advanced Mode*:

1. *Basic Mode* - The user simply inputs the times, *latestReadTimes* and *latestAttackTime* along with their plaintext data. The system creates an FDO for which the probability of reconstructing the FDO after *latestAttackTime* is some value $\leq \beta$ where *beta* is a sensible default such as 0.05. The user interface is two simple text inputs.
2. *Advanced Mode* - The user can input *latestReadTime*, *latestAttackTime*, α, β and their plaintext. The system creates an FDO with m, n that matches their constraints, if possible. For the user interface, the user is presented with an interactive graph where they change the input parameters individually using sliders.

4.5 Comparison of FADE and Neuralyzer

In this section we compare our presentation of FADE to that of Neuralyzer, which we deemed in section 2.8 to be the most apt SDD scheme to date.

4.5.1 Lifetimes

FADE lifetimes vary depending on the configuration of m and n. The longest lifetime of our data occurs by fixing $m = 1$ and increasing n. For $m = 1, n = 170$, we have a maximum lifetime of 32 hours, after which there is an incredibly small ($<< 0.001$) chance of reconstructing an FDO. Whilst this doesn't indicate how useful the lifetime is to an honest reader of the FDO (who wants to reconstruct it with high probability), it reflects the lifetime from the perspective of an attacker.

[9] Document Object Model

If we instead consider FDO lifetime from the view of an honest reader, then for the same configuration of $m = 1, n = 170$, there is a maximum time of about 15 hours when there is a greater than 90% chance of reconstructing an FDO. With this said, it is worth noting that increasing n also increases the time to complete FDO construction and reconstruction. We determine a $n = 100$ to be a realistic upper bound on n, although a user could choose to exceed this if desired.

Neuralyzer Lifetimes

The key difference in lifetimes between our system and Neuralyzer's is that we do not have a method of extending the lifetime of an FDO. Neuralyzer object (EDO) lifetimes can be extended and revoked based on access heuristics and thus are more complex.

Neuralyzer's EDO lifetimes are determined by N, the size of a single DNS potrayal and x, the number of DNS cache hits required for a bit to considered a *1-bit*. In a sense, the ratio x/N in Neuralyzer is comparable to the ratio $\frac{m}{n}$ in FADE, as both provide redundancy.

If we put aside the lifetime extension features, EDO lifetimes in Neuralyzer stand at around 240 hours with $N = 8$; this is shown in figure 4.3, a plot of lifetimes against N, as presented in the Neuralyzer paper.

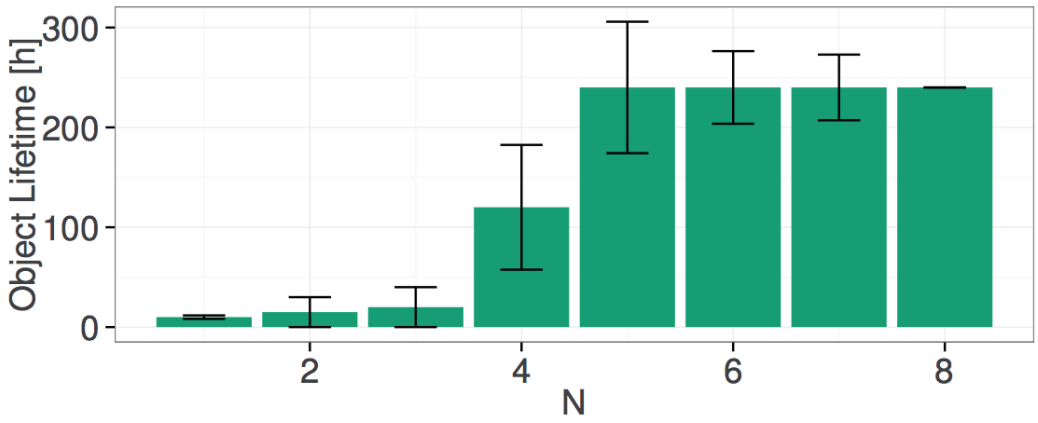

Figure 4.3: Neuralyzer object lifetimes against N, as presented by the authors of Neuralyzer[1].

4.5.2 Performance

Here we consider performance to be the time it takes to perform object construction & reconstruction. The time cost of object construction and reconstruction in both FADE and Neuralyzer, is dependent on the supplied configuration (for the former m, n and latter N, x).

One of the key differences between FADE and Neuralyzer is setup time; FADE requires no setup procedure before performing FDO construction. Subsequently, the time cost of constructing an FDO is bounded by the time it takes to make n search queries. Conversely, Neuralyzer has an initial setup period that runs before beginning EDO construction. This setup period has its own time cost which adds to the overall time to construct an EDO.

Setup Time in Neuralyzer

The initial setup time in Neuralyzer involves finding a random sample of $N * keySizeInBits$[10] DNS resolvers, domain name pairs for which the domain name is not already cached in the resolver.

[10] *keySizeInBits* would typically be 128 as this is a secure key size.

When we created our own prototype implementation of Neuralyzer, we found this setup period to take a significant amount of time. In particular, we observed that queries for many of the random DNS resolvers simply gave errors which meant they had to be retried and then if they still didn't respond, discarded. For $N = 8$ and a key size of 128 bits, Neuralyzer must find 1024 such DNS resolver, domain name pairs[11]. The Neuralyzer paper finds that this would take an average of 0.31 seconds per successful operation[1], which means that at best, the setup would take around five minutes. We note that as DNS relies on UDP, it is incredibly unlikely that one could find 1024 suitable random DNS resolvers without needing to retry at least once.

With this said, a Neuralyzer implementation could run the setup phase pre-emptively for when a user requests an EDO. This would involve searching suitable candidates for DNS Resolver, domain pairs in advance of when they are required. However, because of the random occurences of bit errors in DNS caches, we think that any such pre-emptive setup would still require the pair to be re-verified for suitably before they are used, limiting the extent to which the setup saves time.

Performance Summary

As shown in figure 4.2, FADE has a worst case upper bound of around two seconds for both FDO construction and reconstruction. The property of not having any setup period in FADE makes it much faster than Neuralyzer. Neuralyzer EDOs have a setup time of around five minutes, whereas FADE complete the entire process of FDO construction in less than two seconds. We suggest that this difference in performance is one of the principle advantages of FADE over Neuralyzer.

4.5.3 Degree of Trust In Third Parties

As stated above, its default form FADE, trusts the chosen search engine not to proactively detect queries and scrape them. In section 4.1.4, *Considering Search Engine Companies as Scrapers*, we suggest a modification to our system that would mean this degree of trust is greatly reduced.

By comparison, Neuralyzer employs DNS infrastructure, which is much more decentralised than the search engine approach. Despite this, if owners of DNS infrastructure entities were to corroborate an attack, then there is potentially a security threat. However as Neuralyzer relies on the state of low level details of DNS caches over time, it would be incredibly difficult for such an attack to occur even if multiple parties did corroborate.

Overall, in comparing the security of FADE and Neuralyzer, it is fair to say that Neuralyzer places less trust in third parties.

4.5.4 Comparison Summary

We have seen that Neuralyzer and FADE have different trade-offs in lifetimes, performance and security. We summarise these differences in table 4.2.

[11] This is a reasonable example of a typical Neuralyzer configuration.

Aspect	FADE	Neuralyzer
Object Lifetimes	Static; determined at construction time. Lifetimes can be configured to be between 30 minutes and 32 hours.	Object lifetimes can be extended and revoked. By default, lifetime is a minimum of 20 hours and maximum of 240 hours, without lifetime extension.
Performance	No set-up time. Upper bound on object construction and reconstruction is bounded by two seconds in a worst case. In realistic configurations, construction takes around 0.5 - 1.5 seconds.	Object construction dominated by a roughly 5 minute set-up time. After that, construction and reconstruction is around 3 seconds.
Security	Reasonably secure; extremely difficult for external attackers to scrape search results. The search engine is partially trusted not to proactively detect FDO queries, although this can be mitigated with the proposed extension in section 4.1.4.	Relies only on DNS infrastructure, which is convincingly decentralised. It would be extremely difficult for disparate owners of DNS entities to corroborate to retrospectively attack a data object.
Implementation	Browser extension written in JavaScript; requires only HTTP.	Requires access to low level DNS operations. A Neuralyzer browser extension requires a separate component running on the users machine that provides the extension with the required DNS operations.

Table 4.2: Summative Comparison of FADE against Neuralyzer.

Chapter 5

Conclusion

5.1 Achievements

We have introduced a new Self Destructing Data Scheme, FADE that allows for the creation and reconstruction of Fading Data Objects (FDOs). We showed how FADE can be used with any search engine and specifically analysed how it would work by harnessing the search results of Google news search. We show that the resulting Fading Data Objects have well-defined lifetimes and that the scheme can be deployed into practice in a simple, user-facing browser extension. In section 4.1.4 we determined that our scheme trusts that the chosen search engine does not specifically detect and archive search results for FDO related queries. We deem this to be a security downside in contrast with Neuralyzer, although it is also fairly unlikely. In section 4.1.4 we also presented a modification of the scheme that could mitigate this problem and suggested that this merits future work.

5.2 Application

We believe that FADE can be employed in a Firefox extension to allow end-users to reliably create and reconstruct Fading Data Objects on their favourite websites. Such examples of its use could be for Facebook posts, Tweets, sensitive emails and private messages.

5.3 Future Work

In chapter 4 we suggest several avenues of future work that mitigate some of the challenges presented above, which we outline below:

- **Correlation of Search Results over Time** - in section 4.1.5 we detail the need for a thorough investigation into this aspect of search results. An improved analysis of FADE could take an empirical analysis of the correlation of search results over time and factor it into the theoretical model.
- **Multiple Search Engines to Increase Security** - in section 4.1.4, we suggest how FADE could be modified to make it more difficult for a single search engine company to proactively scrape results of the system. Future work could be undertaken to analyse the churn properties of other search engines and modify the FADE scheme so that is uses multiple search engines.
- **Adaptive Modelling** - in section 4.4.3 we outline the motivation and potential for a system that allows FADE to dynamically improve object lifetime estimates over time. This arises from the observation that our probabilistic analysis of search results is likely to become invalidated over the long term due to changes in search engine algorithms.

Appendix A

Appendix

A.1 Full Description of FADE

FADE is a self destructing data scheme which made up of *Fading Data Objects* (FDOs), an FDO construction algorithm and an FDO reconstruction algorithm. A FADE scheme is parameterised by the search engine it uses, a combining function, f (described in section 3.5.2) and a sequence length, `L`. In addition to this, the FDO construction and reconstruction algorithms are parameterised by parameters m, n, which should be adjusted to obtain the desired FDO lifetime. n is the total number of queries used in the FDO and m is the minimum number of queries that is required to reconstruct the FDO.

A.1.1 FDOs

Fading Data Objects are defined as tuples of the form $\langle c,\ encryptedShares,\ queries,\ m \rangle$. Where the parts are defined as follows:

- `c` - ciphertext of the data encapsulated by the FDO. The ciphertext is the result of encrypting the data using a randomly generated key, k. The key k can be recovered using `encryptedShares` (see below).
- `encryptedShares` - a list of `n` shares, which are Shamir's Secret Shares[15]. The share at index i is encrypted with the key that results from applying the combining function `f` to the search engine news query results of $queries_i$.
- `queries` - a list of `n` queries, which are used to encrypt each share in `encryptedShares`, as describe above. Each search query is a sequence of double digit numbers of length `L`, e.g. "15 73 93 57 23 95", for `L` = 6.
- `m` - the threshold parameter used to produce each share in `encryptedShares` via Shamir's Secret Shares. A minimum of `m` shares need to be decrypted in order to recover the key that encrypts ciphertext `c`.

A.1.2 FDO Construction and Reconstruction Algorithms

Prerequisite: Reverse Shamir's Secret Sharing (RSSS)

The FDO construction and reconstruction algorithms make use of *Reverse Shamir's Secret Shares* (RSSS) which is our design of an abstraction that allows Shamir's Secret Shares to be applied in reverse. The RSSS abstraction is built directly on top of Shamir's Secret Shares and consists of two procedures: `RSSS-CREATE` and `RSS-COMBINE`. Below we provide a formal definition of these which assumes the existence of four procedures:

- `SSS-SPLIT(n, m, key)` - splits *key* into n shares with threshold m, using Shamir's Secret Shares. Returns the list of n shares.
- `SSS-COMBINE(shares)`- combines a list of *shares* back into a key, using Shamir's Secret Shares. Returns the combined key if successful, otherwise throws error. Only m out of the n *shares* need to be valid for the procedure to be a success.

- `ENCRYPT(plainText, key)` - encrypts given plaintext using key provided. Returns the ciphertext.
- `DECRYPT(cipherText, key)` - decrypts given ciphertext text using given key. Returns the cipher text.

`RSSS-CREATE` and `RSSS-CREATE` are defined as follows:

Algorithm 3 RSSS-CREATE

1: **procedure** RSSS-CREATE($n, m, parts$) ▷ parts is a list of n key parts
2: $key \leftarrow$ large random number
3: $shares \leftarrow$ *SSS-SPLIT(n, m, key)*
4: **for** i `from 0..`n **do**
5: $shares_i \leftarrow$ *ENCRYPT*$(shares_i, parts_i)$
6: **return** key, shares

Algorithm 4 RSSS-COMBINE

1: **procedure** RSSS-COMBINE(*parts, shares*)
2: **for** i `from 0..`n **do**
3: $shares_i \leftarrow$ *DECRYPT*$(shares_i, parts_i)$
4: $key \leftarrow$ *SSS-COMBINE(shares)*
5: **return** key

FDO Construction Algorithm

1. Generate `n` random queries, q_i, each a double digit sequence of length `L`. Store all the queries in a list, `queries`.
2. For each q_i, query the chosen news search engine and apply combining function `f`, to obtain n key parts.
3. Call `RSSS-CREATE(n, m, parts)` where `parts` is the list of n key parts created in step 2. above. `RSS-CREATE` returns a key, `k` and a list of encrypted shares, `encryptedShares`
4. Encrypt data using `k`, into cipher text, `c` and construct the FDO $\langle c, encryptedShares, queries, m\rangle$

FDO Reconstruction Algorithm

To attempt to reconstruct an FDO $\langle c, encryptedShares, queries, m\rangle$ into its original plaintext:

1. Perform news search engine queries for every element in `queries` and apply combining function `f` to the results to get a list of key parts, `parts`.
2. Call `RSS-COMBINE(parts, encryptedShares)` to attempt to recover the key, k that was used to encrypt `c`.
3. Decrypt ciphertext `c` using the recovered key k.

Bibliography

[1] A. Zarras, K. Kohls, M. Dürmuth, and C. Pöpper, "Neuralyzer: Flexible Expiration Times for the Revocation of Online Data," *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy*, pp. 14–25, 2016. [Online]. Available: http://doi.acm.org/10.1145/2857705.2857714

[2] BBC News. (2017) Equifax says almost 400,000 Britons hit in data breach. [Online]. Available: http://www.bbc.co.uk/news/technology-41286638

[3] BBC News. (2017) Uber concealed huge data breach. [Online]. Available: http://www.bbc.co.uk/news/technology-42075306

[4] G. Greenwald, "NSA collecting phone records of millions of Verizon customers daily," jun 2013. [Online]. Available: https://www.theguardian.com/world/2013/jun/06/nsa-phone-records-verizon-court-order

[5] The Verge. (2017) Yahoo says all 3 billion user accounts were impacted by 2013 security breach. [Online]. Available: https://www.theverge.com/2017/10/3/16414306/yahoo-security-data-breach-3-billion-verizon

[6] Adobe. (2013) Adobe Blog customer security alert. [Online]. Available: https://theblog.adobe.com/important-customer-security-announcement/

[7] Guardian News. (2018) Facebook says cambridge analytica may have gained 37m more users' data. [Online]. Available: https://www.theguardian.com/technology/2018/apr/04/facebook-cambridge-analytica-user-data-latest-more-than-thought

[8] GibSec. (2013) Snapchat - GibSec Full Disclosure. [Online]. Available: http://gibsonsec.org/snapchat/fulldisclosure/

[9] Snapchat. (2017) Snapchat privacy policy. [Online]. Available: https://www.snap.com/en-US/privacy/privacy-policy/

[10] R. Perlman, "The Ephemerizer : Making Data Disappear," *Network*, pp. 1–20, 2005.

[11] Databox Project. (2018) Databox. [Online]. Available: https://www.databoxproject.uk

[12] FreedomBox Foundation. (2018) FreedomBox. [Online]. Available: https://freedomboxfoundation.org

[13] R. Perlman, "The Ephemerizer : Making Data Disappear," *Network*, pp. 1–20, 2005.

[14] R. Geambasu, T. Kohno, a. a. Levy, and H. M. Levy, "Vanish: Increasing data privacy with self-destructing data," *USENIX security symposium*, pp. 299–316, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855787

[15] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[16] "Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs," *Ndss*, no. March, pp. 1–20, 2010.

[17] C. Castelluccia, E. De Cristofaro, A. Francillon, and M. A. Kaafar, "EphPub: Toward robust Ephemeral Publishing," *Proceedings - International Conference on Network Protocols, ICNP*, pp. 165–175, 2011.

[18] M. Kührer, T. Hupperich, J. Bushart, C. Rossow, and T. Holz, "Going Wild: Large-Scale Classification of Open DNS Resolvers," *Proceedings of the 2015 ACM Conference on Internet Measurement Conference - IMC '15*, pp. 355–368, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2815675.2815683

[19] G. Amjad, M. S. Mirza, and C. Pöpper, "Forgetting with Puzzles : Using Cryptographic Puzzles to support Digital Forgetting."

[20] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release Crypto," 1996.

[21] DuckDuckGo. (2018) About DuckDuckGo. [Online]. Available: https://duckduckgo.com/about