Crumpled and Abraded Encryption: Implementation and Provably Secure Construction

by

Scott Sherlock Griffy

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Charles V. Wright, Chair
Mayank Varia
Wu-chang Feng
Bryant W. York

Portland State University
2019

Abstract

Abraded and crumpled encryption allows communication software such as messaging platforms to ensure privacy for their users while still allowing for some investigation by law enforcement. Crumpled encryption ensures that each decryption is costly and prevents law enforcement from performing mass decryption of messages. Abrasion ensures that only large organizations like law enforcement are able to access any messages. The current abrasion construction uses public key parameters such as prime numbers which makes the abrasion scheme difficult to analyze and allows possible backdoors. In this thesis, we introduce a new abrasion construction which uses hash functions to avoid the problems with the current abrasion construction. In addition, we present a proof-of-concept for using crumpled encryption on an email server.

Acknowledgments

I'd like to thank Portland State University for supporting me while creating this thesis, my thesis committee for their feedback and support, Charles and David for getting me into writing academic papers, my family and friends for enduring my ramblings about my work, and every teacher that took an interest in seeing me learn throughout the years.

Table of Contents

Abstract i

Acknowledgments ii

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Importance of compromise 2

1.2 Abrasion and crumpling 3

1.3 Beneficial side-effects of decryption cost 6

1.4 Motivation for a new abrasion construction 6

1.5 Contributions 7

1.6 Thesis outline 8

2 Related work 9

3 Preliminaries 11

3.1 Time/Memory Trade-offs (TMTOs) 11

3.1.1 Optimal trade-off parameters 15

3.1.2 TMTO improvements 15

3.2 Secret Sharing 16

4 Crumpling implementation 18

5 Abrasion description 23

5.1 Properties of abrasion schemes 23

5.1.1 Ideal properties 23

5.1.2 Relaxed properties 26

5.1.3 Optional properties 27

5.1.4 Practical abrasion 29

5.2 Security Definitions 30

5.2.1 Threat model 30

5.2.2 Security Definitions 31

5.3 Abrasion scheme functions 32

5.3.1 Abrasion scheme initialization 32

5.3.2 Abrasion tag generation 33

5.3.3 Abrasion private key generation 33

5.3.4 Abraded key retrieval 34

6 Abrasion constructions 35

6.1 A simple (but insecure) hash-based abrasion scheme 35

6.1.1 Abrasion scheme initialization 37

6.1.2 Abrasion tag generation 37

6.1.3 Abrasion private key generation 38

6.1.4 Abraded key retrieval 38

6.2 TMTO success chance 39

6.3 Main construction 40

6.3.1 Intuition 41
6.3.2 Description 44
6.3.3 Abrasion scheme initialization 46
6.3.4 Abrasion tag generation 47
6.3.5 Abrasion private key generation 49
6.3.6 Abraded key retrieval 49
6.3.7 Cost calculations 51
6.3.8 Decisions, details and alternatives 52
6.3.9 Probabilistic sharing 53
6.3.10 Main construction success chance 54

7 Security 56
7.1 Legitimate attacker definition 56
7.2 Security game and oracle description 57
7.3 Success calculations 59
7.4 Other attacks 65

8 Results 68
8.1 Summary 68
8.2 Detailed cost analysis 70
8.3 Revisiting abrasion requirements 75

9 Conclusion 78
9.1 Future work 78
9.2 Crumpling and abrasion 79
9.3 Safety and privacy 79

References 81

Appendices 90

A Hellman table success derivation 90

List of Tables

6.1 Important variable definitions 46

8.1 Costs with other parameters 76

List of Figures

3.1 Example of a chain merge in a TMTO 14

5.1 Message decryption chance in an ideal abrasion scheme 25

5.2 Message decryption chance in a realistic abrasion scheme 28

6.1 Probability of success of different TMTO solutions (up to $\approx 86\%$) vs tableless approach, $N = 2^{25}$ 39

6.2 Tableless vs table attacker, MultipleSame construction. $Y = 5, N = 2^{25}$ 42

6.3 Tableless vs table attacker, MultipleDifferent construction. $Y = 5, N = 2^{25}$ 44

6.4 Generation of each element in the abrasion tag in Algorithm 9 48

6.5 Retrieval of each share in an abrasion tag in Algorithm 11 50

7.1 Uniform minimum sum pdf (approximated with 20,000 samples) vs normal approximation 62

7.2 Uniform minimum sum cdf (approximated with 20,000 samples) vs normal approximation 63

8.1 Tableless vs table attacker, success chance vs work 70

8.2 Tableless attacker ϵ to $1 - \epsilon$ success chance 71

8.3 Table attacker ϵ to $1 - \epsilon$ success chance 72

Chapter 1

Introduction

Recently there has been growing controversy over the usage of strong encryption in messaging platforms such as WhatsApp and iMessage [9, 32]. Modern cryptography allows anyone to protect their privacy using cheap computers and networks through strong, end-to-end, encryption. But this same strong encryption can also prevent law enforcement from performing important investigations [53, 32]. Many officials want law enforcement to have the ability to bypass strong encryption for investigation [3], sometimes called *exceptional access*. But some governments abuse this power and have invaded their citizen's privacy using digital methods [14].

Enforcing both privacy and public security is important, but it is not easy to do both simultaneously. These two goals have lead to two extreme opinions on whether to allow the use of end-to-end encryption in mass communication software. One option, referred to as "Going Dark," would mean allowing unbreakable encryption everywhere. This would prevent the recovery of records even when permitted through warrants [53, 30]. The converse, can be referred to as "Going Bright," where we allow governments mass access to recover communication through means such as wiretapping [27].

1.1 Importance of compromise

To understand the importance of a compromise between privacy and safety, we pose a theoretical, yet practical, scenario where a compromise between these two positions is necessary.

Let's say that the FBI is investigating a planned terrorist attack, but doesn't yet know the location. The terrorists are using a messaging platform to communicate and coordinate. If we accept the "Going Dark" path, strong encryption is deployed on the platform and there is nothing the FBI can do to find the location before it is too late. This scenario would favor the "Going Bright" path, because this terrorist attack could be prevented in this case.

Let's now imagine that a group of whistleblowers are collecting evidence to expose a corrupt official at the FBI. If we "go bright," this corrupt official could learn about these whistleblowers and use his or her power in the FBI to fire the whistleblowers. In this case, we would want the "Going Dark" solution, where the whistleblowers' communication was hidden from officials that would use that information to control them.

Another unwanted scenario to consider is where a malicious attacker somehow steals an FBI official's access to the system used for the "Going Bright" solution (access to wiretapping or backdoors). This could occur through some means such as malware on their computer or social engineering. This malicious attacker would now have unauthorized power to spy on Americans.

How can we plan a system that responds correctly in all of these situations, preventing surveillance of users with honest communication, while still allowing officials to investigates threats to public safety?

Ideally, we want a solution that would allow law enforcement to recover exactly

what they need to complete their investigation and nothing else. Furthermore, we want this solution to prevent any malicious adversary from reading any records in the case of a breach.

Many would argue that these criminals could simply use different messaging apps that did not have exceptional access built in [2]. We know that many criminals do not do this, and would be caught if the major messaging platforms had exceptional access [5, 4].

One construction intended to solve this problem is the Escrowed Encryption Standard, which was used by the Clipper chip [49]. There are currently many problems with key escrow which encrypts messages with keys that law enforcement has access to. Encrypting messages with these keys creates a single point of failure for mass, malicious, decryption and provides many opportunities for programmers to code flaws into their security protocols [22].

Would our justice system be able to function without access to digital records? If the answer is no, then compromise is essential. More and more of our communication is occurring online and increasing the need for a solution to this problem.

1.2 Abrasion and crumpling

In 2018, Wright and Varia proposed the concept of *abraded* and *crumpled* encryption to protect records from unlimited investigation while still allowing for some decryption [58, 59]. Diverging from previous schemes, crumpled and abraded encryption require no key escrow or backdoors, ensuring that there is no single point of failure. Instead, these two encryptions require computational work to be done in order to recover the message. This work is divided into a per-message cost and a greater one-time cost to prevent different types of malicious attackers from decrypting the

messages.

Crumpled encryption is an encryption that imposes a specific cost **per-message**. Imposing a per-message cost prevents any resource-bounded recipient of an encrypted dataset from performing an unlimited number of decryptions. The total cost of reversing the crumpled encryption grows as more messages are decrypted which prevents mass decryptions and focuses the target of decryptions to be relevant to investigations. The cost analysis is aided by the extensive work that has recently been done in computing DoubleSHA256 due to the popularity of Bitcoin [47, 17]. By forcing attackers to compute DoubleSHA256 many times, we can calculate the computational "work" required by an attacker to decrypt a record. Using electricity cost we can then convert this into a monetary value. Wright and Varia suggest a cost of $1000 (or larger) per message to limit investigations. When used alone, crumpled-encryption could allow for a resource-bounded adversary to decrypt a few messages, potentially doing harm. We could solve this by increasing the crumpling cost, but that might make the cost of decryption too high, hindering legitimate investigation. This is where *abraded encryption* helps.

Abraded encryption is meant to impose a **one-time** cost on the decryption of many messages. This prevents attackers with small resources from decrypting any records. Generally, the required cost for breaking abrasion is much higher than the per-message cost required by crumpled encryption, measured in the millions instead of thousands of dollars. A high initial cost deters illegitimate attackers who may gain encrypted messages through a data breach rather than through warrants as our legitimate attacker would do. These attackers (we call malicious attackers) will not always have the necessary resources required to break the abraded encryption. If an attacker only wants one message, they will still have to spend millions of dollars. Law enforcement (our legitimate attacker) will want to break the abraded encryption to

decrypt many messages over many investigations, thus amortizing their total cost over time. After an attacker spends the one-time-cost to break the abraded encryption, they pay only the crumpling costs for each message.

Both of these encryptions can produce information to aid in breaking the given key. The extra information is sometimes called a "tag" or a "puzzle." After a tag is created, the key is considered to be "abraded" or "crumpled" allowing for retrieval using the generated tag.

The cost of the abraded encryption is set to ensure that law enforcement can decrypt it while still being high enough to prevent many attackers from being able to decrypt messages. For example, if we had an abrasion scheme that cost $1 million dollars to break, it would not be a significant portion of the budget for the FBI [58], but many attackers would not be able to spend this cost to start decrypting records.

The threat of a low resource attacker (such as a hacker) gaining records is very practical. Encrypted records of messaging platforms are not always stored on secure servers [15] and hacking these servers doesn't require a lot of resources. Wide-spread adoption would push the need for abraded encryption even further as there will be more chance of breach.

The abrasion construction presented by Wright and Varia [58] uses a Diffie-Hellman key exchange [35] with small primes as public key parameters. The authors use a precomputation attack described by Adrian et al. to measure the cost of breaking the scheme [23].

Some constructions for exceptional access involve using a public ledger [36], or requiring physical possession of devices [54]. These are great ideas which we hope can be composed with the scheme used in this thesis. We provide more details on these schemes in Chapter 2. Even if governments wish to include key escrow or backdoors, crumpled and abraded encryption could be applied to the escrowed messages as well.

This would decrease the risk of abuse and breach.

1.3 Beneficial side-effects of decryption cost

We have discussed how crumpled encryption rate limits government decryptions at $1000, but there are also benefits to having any amount required for decryption.

Adding a monetary cost to decryptions has another important effect on tracking how governments perform investigations. We have many systems to track the flow of money in corporations such as enterprise software like Concur [8]. These applications could help expose abuse of gathered records even at lower crumpling costs.

A benefit of abraded encryption over schemes like key escrow is the ability to easily change keys after a data breach. If law enforcement has a data breach and attackers recover the work required to break the abraded encryption, the public key could be changed with no coordination with law enforcement. After the public key is changed, the stolen private key would be useless for future messages. The abrasion construction presented in this thesis does not require agreement for public key choice. Removing this barrier to scheme initialization makes changing the public key much easier.

1.4 Motivation for a new abrasion construction

Crumpled encryption is already well defined and accomplishes its goals [58], but there is still room for improvement with the more complex problem of abraded encryption.

Crumpling leverages the recent work done to compute DoubleSHA256 efficiently to have accurate estimations of cost [58, 47, 17]. Unfortunately, the proposed abrasion construction did not have the same depth of work backing its cost.

The current abrasion scheme (described in Section 1.2) relies on weakened public key parameters such as primes. Many are skeptical of using primes generated by government agencies. NIST published a cryptographically secure pseudorandom number generator (CSPRNG) which relied on specific group parameters which many believed to have a backdoor created by the NSA [55]. This backdoor could have been created by choosing specific primes with algebraic relations that would allow the NSA to predict bits of randomness created by the CSPRNG. An abrasion scheme which is not susceptible to these backdoors is desirable.

The monetary cost of breaking any given abrasion scheme decreases as hardware becomes more efficient. To maintain a fixed cost, the parameters of an abrasion scheme would need to be modified over time to match the efficiency of current hardware. Finding safe parameters to use in these schemes is a difficult problem [19] which would be exacerbated if the parameters were modified frequently.

More analysis of the original abrasion construction is needed to know whether it provides security against attackers that only want to retrieve a single message. The notion of security against this type of malicious attacker is described in Section 5.1 in Property A. Analyzing this existing construction would require significant knowledge of modern attacks on public key schemes and is beyond the scope of this thesis. The original authors of abraded encryption [58] recognized this gap in security and kept their security proofs modular to allow for improvement.

1.5 Contributions

In this thesis, we address the problems with the existing abrasion scheme. As our main contribution, we propose a construction that uses cryptographic hash functions instead of weakened public key parameters. Using hash functions allows us to provide

provable bounds on its security against a malicious attacker. Cryptographic hash functions are also much easier to predictably weaken than public key parameters. We present this main construction is described in Chapter 6.

A second contribution of this thesis is a set of detailed requirements and security definitions for an abrasion scheme. These are defined in Chapter 5. We also present a proof of concept of a crumpled encryption library and an example of an integration of crumpled logging with an email server in Chapter 4.

The abrasion construction presented in this thesis relies on hash functions and *Time/Memory Trade-Offs (TMTOs)* to create an abrasion construction that meets the desired requirements. We review TMTOs in Section 3.1.

1.6 Thesis outline

In Chapter 2, we provide background on the technologies used in this construction and discuss other solutions similar to ours. Details of preliminaries that we make extensive use of in this thesis are reviewed in Chapter 3. A small proof-of-concept implementation of crumpling is presented in Chapter 4. Subsequent chapters focus solely on abraded encryption. We specify properties, present a generic set of functions, and define the security of an abrasion scheme in Chapter 5. Our main construction is presented in Chapter 6 along with some intuition of the statistics involved to compute the success chance of attackers. The equations needed to calculate the security for the main construction are derived in Chapter 7 along with a security game and oracle. We use practical parameters to compute the security and cost of our construction in Chapter 8. Finally, we sum up our contributions and discuss the future of this work in Chapter 9.

Chapter 2

Related work

In this chapter we discuss constructions that are similar to or used by our construction.

Key escrow is a type of encryption scheme that was introduced in the 1990s. Key escrow systems perform an encryption that allows authorized individuals such as law enforcement to decrypt ciphertexts [34].

One construction that seeks to hold investigators accountable is *Accountability of Unreleased Data for Improved Transparency (AUDIT)*, created by Frankle et al. This construction utilizes a public ledger to track police investigations. The resulting publicity is meant to ensure that warrants are properly followed during investigations [36].

An approach, by Savage, ensures that police have physical access of devices that they are investigating. This construction uses secure hardware to *self-escrow* keys into the device. These keys cannot be read by software and instead require physical access for some amount of time to be read. Enforcing physical access reduces privacy violations done over the internet and allows for existing procedures for warrants on physical evidence to easily apply to digital assets [54].

In 1980, Hellman introduced the concept of using *Time/Memory Trade-Offs (TMTOs)* to aid in the reversal of cryptographic functions [40]. His work focused on

finding a key given only the ciphertext of the symmetric encryption scheme DES (Data Encryption Standard) [48]. These TMTOs can help reverse any cryptographic function, including cryptographic hashes. The process of reversing these functions is called cryptanalysis. Hellman's original work showed that stored precomputation could be used to speed up later cryptanalysis. This is where the name "time/memory trade-off" comes from, as it trades off storage space (memory) in order to reduce the time of later decryptions. One common form of a TMTO is called a "Rainbow Table" [50]. Readers may know this as a method to reverse password hashes. TMTOs will be further discussed in Section 3.1.

Abraded and crumpled encryption are similar to time-lock puzzles. Time-lock puzzles impose a time-cost for decryption or proof [52]. Abraded and crumpled encryption differ as they are not meant to enforce a time requirement, but rather a work requirement, measured in dollars. Specifically, time-lock puzzles generally try to find inherently serialized problems, while abraded and crumpled decryptions can be done in parallel.

Abraded encryption is similar to asymmetric encryption, where many have a public key and can encrypt, but only those with the private key can decrypt. Asymmetric encryption has a long history and is used by many systems today. An early example of public key encryption is RSA, created by Rivest et al. [51].

We utilize a technique known as secret sharing, first invented simultaneously by Shamir and Blakley [56, 28]. Secret sharing allows some fraction m of n total "shares" to decrypt a message. We review secret sharing in Section 3.2.

Chapter 3

Preliminaries

In this chapter we provide more depth on technical schemes that we use in our main construction. We first review TMTOs in Section 3.1. Then, we discuss secret sharing in Section 3.2.

3.1 Time/Memory Trade-offs (TMTOs)

Before our construction can be discussed, readers must have some basic knowledge of how a TMTO can be used for cryptanalysis. This section also defines some of the symbols used in equations throughout the paper.

TMTOs can help reverse cryptographic functions such as encryption or hash functions. TMTOs are generic and can be used for any cryptographic functions. The main construction in this thesis uses TMTOs to reverse hash functions. Throughout the thesis, we use the terms "preimage" and "hash result" (sometimes shortened to just "hash") while discussing how TMTOs work. A preimage is an input to a hash function and a hash result is the output of a hash function.

A TMTO "attack" is separated into a precomputation ("offline") phase and a decryption ("online") phase. In the precomputation phase, the target hash function is computed many times, but only some of the results are stored. The stored results are sorted so that they can be efficiently searched during the online phase to reverse

hashes. The stored precomputation of a TMTO can be reused to reverse many hashes.

In order to reverse hashes that we computed but did not store, we compute hashes in *chains*. We also use a *reduction function* which transforms the output of the hash function into a valid input for the hash function. This allows us to create these chains by choosing a *starting point* and computing the hash function and reduction function repeatedly. Chains computed for a specific hash function and reduction function are stored as "rows" in a TMTO *table*.

Equation 3.1 shows the structure of a chain. The reduction function is represented as $R(\cdot)$ and the hash function as $h(\cdot)$. Each starting point of a chain ($p_{i,0}$) is a distinct and valid input to $h(\cdot)$. $(R \circ h)^j$ is a function composition of R and h $\big(R(h(\cdot))\big)$, composed with itself j times.

$$\begin{aligned} & p_{i,0} && \rightarrow R(h(p_{i,0})) && \rightarrow (R \circ h)^2(p_{i,0}) && \rightarrow \ldots && \rightarrow (R \circ h)^t(p_{i,0}) \\ = & p_{i,0} && \rightarrow p_{i,1} && \rightarrow p_{i,2} && \rightarrow \ldots && \rightarrow p_{i,t} \end{aligned} \tag{3.1}$$

There are m chains in each table ($0 \leq i < m$). Only values $\sum_{i=0}^{m}(p_{i,0}, p_{i,t})$ are saved from the precomputation. All other values are discarded to save storage space. This means that the storage space required for a table is proportional to m. The total amount of work (w) put into the table is proportional to $w = m * t$. This work is measured in invocations of the hash function $h(\cdot)$.

When reversing a certain hash result, c, where $h(x) = c$, we first apply the reduction function to c $\big(R(c)\big)$ and check if the result is stored as an endpoint in our table. If this value doesn't exist in our table, we apply the hash function and the reduction function again $\big(R(h(R(c)))\big)$ and check if this result is an endpoint of the table. The hash function and reduction function are repeatedly computed until we find a $p_{i,t}$ in the table that matches a value computed on c $\big(p_{i,t} = (R \circ h)^j(R(c))\big)$. The chain is

then recomputed from the stored starting point $p_{i,0}$ to find a preimage that generates the given hash $\big((R \circ h)^{t-j-1}(p_{i,0}) = p_{i,t-j-1}, H(p_{i,t-j-1}) = c\big)$.

Because of hash collisions, we may find a preimage (p') such that $h(p') = h(p) = c, p' \neq p$ where p is the "correct" preimage for c. This collision is a type of *false alarm* and adds to the cost of our online phase. There is also a chance that $h(p)$ was never computed in the table. The chance that $h(p)$ is in the table is called the table's probability of success, labeled as $\Pr[S_{\text{table}}]$.

Hellman proves that the cost of false alarms can't increase online cost per table (T_{table}) by more than 50% [40]. This means that our search cost for a TMTO table will be $T_{\text{table}} \leq t * 1.5$ where there each chain in the table is of length t.

To compute this table, we choose a number of starting points (p_i) in the preimage space such that $\forall i \in \{0, 1, ...m - 1\}, p_i \in \{0, 1, ...N - 1\}$ where N is the number of inputs to the hash function. Each starting point is distinct, as using the same starting point would result in an identical chain and add no value to the table. We store a p_i with the resulting end point $\big((R \circ h)^t(p_i)\big)$ in the table to represent each chain.

There is a chance that two or more of these rows (chains) may merge at some point, causing the rest of the chain to be duplicated: $(R \circ h)^y(p_i) = (R \circ h)^u(p_k), i \neq k$. Figure 3.1 shows an example of two chains merging. These *chain merges* become more likely as the table becomes larger. Chain merges cause stored TMTO precomputations to become less effective as m (the size of the table) increases. In other words, the success $\Pr[S_{\text{table}}]$ increases sublinearly with the work put into the table w. This problem has motivated most of the innovation in the field of TMTO constructions. *The most complex parts of the main construction presented in this thesis are designed to overcome the diminishing returns of TMTO constructions.*An attacker that does not compute a table to reverse $h(x)$ (a *tableless attacker*) does not encounter these merges which gives them an advantage over the table attacker.

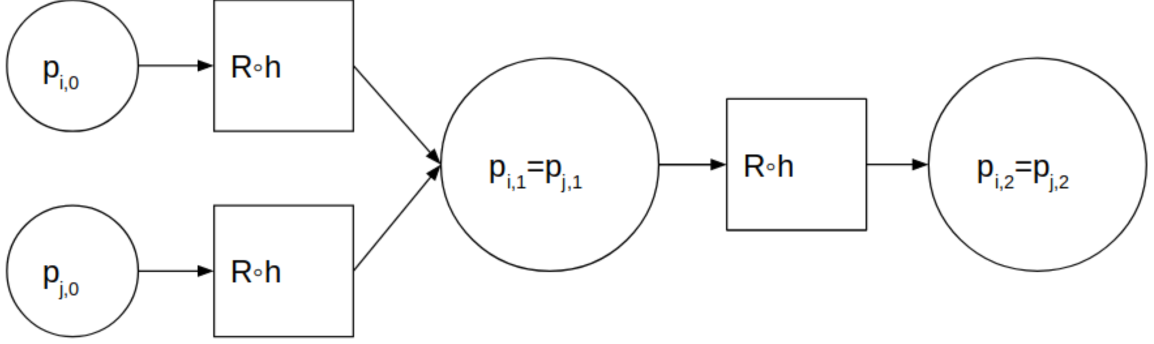

Figure 3.1: Example of a chain merge in a TMTO

Hellman developed a method to combat merging chains by computing multiple tables (l tables) on the same hash function $h(\cdot)$. Each table uses a distinct reduction function ($R_i(\cdot), i \in \{0, 1, ..., l-1\}$). These tables may have collisions, but their chains do not merge because they use different reduction functions. Using multiple tables greatly increases the chance of success of a TMTO solution, but also increases our online time as we now need to compute a different function on the target hash for each of the tables. Because we must search multiple tables, our online cost (T) becomes $T = l * T_{\text{table}} = l * t * 1.5$. Computing these tables also multiplies our storage space required (M) by l, $M = l * m$.

When we can reverse a target hash using the stored TMTO precomputation we call this a "success." We derive the probability of success for Hellman's TMTO in Appendix A for any given TMTO parameters (l, m, t). This probability of success is shown in Equation 3.2.

$$\Pr[S_{\text{TMTO}}] = 1 - exp\left(\frac{l}{t}\int_{u=0}^{u=\frac{mt^2}{N}} \frac{1-e^{-u}}{u} du\right) \tag{3.2}$$

3.1.1 Optimal trade-off parameters

The most "efficient" trade-off occurs when $m = t = l$, where m is the number of rows stored in each table, t is the length of each chain, and l is the number of tables computed [40]. In this case, efficiency is measured as the sum of the rows required (M) and the online compute time (T) over the number of invocation of the hash function ($m * t * l$).

Many TMTO authors focus on analyzing tables computed to exactly N invocations of the hash function, where N is the input space of the hash function $h(\cdot)$ [42]. Using the optimal efficiency trade-off, this means our parameters are $m = t = l = N^{\frac{1}{3}}$. Stopping at at N invocations strikes a decent balance between gaining a significant success chance and losing too much work to chain merges. While more careful adjustments of these parameters may yield a better trade off for abrasion, we use this "standard" trade-off as a guide for the parameters in our scheme ($m = t = l = N^{\frac{2}{3}}$).

3.1.2 TMTO improvements

A variant of Hellman's TMTO solution uses a technique called "Distinguished Point tables" (DP tables) in order to reduce the number of merging chains in TMTOs. DP tables were originally suggested by Ronald Rivest [33]. The distinguished points method was formalized and improved by Borst et al [29]. Borst el al. also introduced the notion of perfect tables in the same paper. A perfect table is a TMTO construction that removes merging chains during precomputation. This reduces the number of false alarms and the size of the table.

An innovation by Avoine et al. reduces the cost of false alarms using a method called "checkpoints." In this method, extra information is added to the table to reduce the online cost to check for false alarms [24].

We did not consider DP tables or checkpoints for this construction. Instead we simply prove that a basic TMTO can meet the requirements of an abrasion function and leave details to be completed by an attacker. Using better TMTO constructions can only strengthen our scheme as we measure the security of our scheme by comparing the efficiency of an attacker that builds a TMTO against an attacker who does not compute any tables.

Oechslin created Rainbow Tables in 2003 [50] which uses a small number of tables with rotating reduction functions. We show the success probability of this construction in Section 6.2. Our main construction does not use rainbow tables as it is difficult to compute the probability of success for large values of N, which is required for our scheme to impose practical initial costs. There is far more research on Hellman tables, ensuring that our analysis is more robust.

3.2 Secret Sharing

Secret sharing, first invented simultaneously by Shamir and Blakley [56, 28], is a cryptographic function that encrypts a secret k by generating n different “shares.” Any subset of these n shares (of fixed size m) can be used to derive the secret.

One of the first secret sharing algorithms used polynomial interpolation [56]. Each share is an evaluation of the polynomial at a different point. The polynomial is of $(m-1)$ order and n evaluations of the polynomial are generated. Any polynomial of $(m-1)$ order can be perfectly interpolated with any m different points. This means that any m of the n generated shares can be used to recreate the polynomial. We then set the secret as the y-intercept (or another, secret evaluation) of the polynomial so that any party with m shares can find this secret through interpolation. The arithmetic for these polynomial evaluations is done over a prime field to ensure it is

more secure.

In our main construction, we use secret sharing in a generic way. This generic secret sharing scheme has two functions: SSGen, a function that generates shares for a secret, and SSDecrypt, a function that finds the secret using the shares.

SSGen takes a secret and scheme parameters m, n as input. The function creates and returns a set of n shares, labeled s ($|s| = n$). Many secret sharing schemes also generate a ciphertext c [43].

$$s, c \leftarrow \text{SSGen}(k, m, n), m \leq n$$

SSDecrypt is a function to retrieve a secret k. The function accepts the ciphertext generated by SSGen and a set of shares. This set of shares (s') is a subset of s of size m ($|s'| = m, s' \subset s$).

$$k \leftarrow \text{SSDecrypt}(c, s')$$

We use secret sharing in our main construction in Section 6.3. Secret sharing improves the success chance of our legitimate attacker in our abrasion scheme. The legitimate attacker's probability of success with secret sharing is derived in Section 6.3.10.

An alternative to secret sharing for our main construction is described in Section 6.3.9. This improves the performance of our scheme, but we do not analyze the security when using this alternative method.

Chapter 4

Crumpling implementation

In this section we present a proof of concept of crumpled-encryption in C and modify an email server to use it. We chose to install crumpled encryption into Postfix [18], which exists on 34.31% of email servers as of 2018 [16].

Crumpled encryption is meant to impose a per-message cost on an attacker that wants to decrypt records. After a key has been crumpled, any attacker must compute many hash functions in order to retrieve the key. The number of hash function invocations is variable depending on how many possible inputs to the hash function there are. Knowing the expected number of hash invocations, we can determine the monetary cost of this decryption using known hardware efficiency rates for Bitcoin mining [17]. The monetary cost is the cost of electricity spent by this adversary.

This implementation mostly follows the construction outlined in the original abrasion/crumpling paper [58]. A random public nonce is added to prohibit TMTOs from being used to reduce the cost of many decryptions. Adding this requires us to store a *tag* to reverse crumple-encrypted messages. This tag stores information like the nonce to ensure decryption is possible.

The header file for the crumpled-encryption library (*crumple.h*) includes a number of declarations. One of these declarations is the crumpled message struct shown in Listing 4.1. This struct stores information needed to recover a message, including

the ciphertext. A serialized version of this struct is stored in the Postfix logs. The function in Listing 4.2 performs the actual crumpled-encryption on a message and stores it in a struct. A key can be crumpled by providing it in place of a message. The library provides a method for writing the structs to log files using file descriptors, shown in Listing 4.3.

Listing 4.1: Crumpled message struct

```
struct crumpled_msg{
        unsigned char* enc_msg; // the ciphertext of the message (
            ↪ crumple-encrypted)
        unsigned char keygen_hash[32]; // hash to break
        unsigned char nonce[32]; // nonce to prohibit TMTOs
        int bits; // difficulty of the puzzle
};
```

Listing 4.2: Crumpled encryption function

```
void crumple_enc(unsigned char* message, struct crumpled_msg* cmsg);
```

Listing 4.3: Serialize crumpled struct function

```
void serialize_crumpled_msg(FILE* fd, struct crumpled_msg* cmsg);
```

In *crumple.c*, the crumpled-encryption library implementation, OpenSSL was used to perform the cryptographic functions [13]. The "getrandom" Linux system call was also used to generate randomness [10].

A small executable is shown in Listing 4.4 which uses the crumpled encryption library to do a single crumpled-encryption. This takes a plaintext and a difficulty as input and writes out the serialized crumpled ciphertext.

Listing 4.4: Crumple encryption binary

```
// crumple_enc.c
int main(int argc, char **argv){
        int num_bits = atoi(argv[2]); // read in the difficulty
        unsigned char* text = argv[1]; // read in the plaintext
        struct crumpled_msg cmsg; // initialize a struct
        cmsg.bits = num_bits; // set the difficulty
        crumple_enc(text, &cmsg); // call crumpled encryption library
        serialize_crumpled_msg(stdout,&cmsg); // write the crumpled
            ↪ encryption out
}
```

This binary is used by a script which is called by Postfix. This script is shown in Listing 4.5. It takes in an email from Postfix and crumple encrypts the TO and FROM fields in a log file before sending it along the Postfix message pipeline.

Listing 4.5: Crumpled logging script

```
#!/bin/bash
# this script is stored in /opt/crumple/crumplelog_postfix.sh


# take note of the date
received_date=$(date)
# need to call the binary that we created that will crumple-encrypt a
   ↪ single message
crumple_enc_binary=/opt/crumple/crumple_enc
# compute the crumpled encryptions
# $3 is the FROM field, $4 is the TO field.
```

```
# The 50 here is the difficulty (in bits)
from_crumpled=$($crumple_enc_binary $3 50)
to_crumpled=$($crumple_enc_binary $4 50)
# write the crumple-encrypted message to the log
printf "email:%s:\n" $received_date >> /var/log/crumpled_email.log
printf "\tfrom:\n\t\t%s\n\tto:\n\t\t%s\n\n" $from_crumpled
   ↪ $to_crumpled >> /var/log/crumpled_email_headers.log
# allow email to continue through Postfix (reach receiver)
/usr/sbin/sendmail $@


exit $?
```

We make two edits in the Postfix configuration file shown in Listing 4.6. These edits modify Postfix to crumpled encrypt logs.

Listing 4.6: Postfix configuration for crumple encrypted logging

```
# here are excerpts from the Postfix configuration file: /etc/postfix/
   ↪ master.cf
...
# This line pipes smtp traffic through a custom filter
smtp inet n - n - - smtpd -o content_filter=crumplelog-pipe
...
# This line creates the custom filter and points it at our crumpling
   ↪ script
crumplelog-pipe unix - n n - - pipe
       flags=Rq user=vmuser argv=/opt/crumple/crumplelog_postfix.sh -
          ↪ oi -f ${sender} ${recipient}
```

An online tutorial aided the creation of this crumpling bash script and modifications to the Postfix configuration [7].

Listing 4.7 is an example of a crumple encrypted log as generated by the crumpled logging script. This examples shows a log file with the crumple-encrypted details of a single email.

Listing 4.7: Crumple encrypted log

```
email:Mon Apr 22 20:05:12 PDT 2019:
       from:
              |bits:50|hash:e89f0b6e70f8d3dadae77963a835b55b-62
                 ↪ ef62a6f7f9774a8a4044d792a25c25|ciphertext:78
                 ↪ a6760f57d49f7631a7f6d9d4482ec0-02366
                 ↪ ddcfb2bab6c3a39eadb4d5dc5a8|nonce:
                 ↪ ab08bf410892fe5f54b3ced722d652fc-
                 ↪ ef43f4b283343d0474a6e93c2aa91dfd|
       to:
              |bits:50|hash:e74e93d351e245101375741d70f1b739-
                 ↪ e465e4f0ce4926e184bf6a0468fa494b|ciphertext:89
                 ↪ a2251e429245a8ec505c33d0b93071-9
                 ↪ e9aadb36587a3dcb57060d3f11941f4|nonce:142
                 ↪ e792ceb6cd4c8b4017355390a25e7-
                 ↪ c944412fc72edd12ae4038aefc3804bc|
```

Chapter 5

Abrasion description

Abrasion is an encryption scheme invented by Wright and Varia [58]. We reviewed abrasion in Section 1.2. Abraded encryption imposes a large one-time cost on the decryption of many messages. An initial cost prevents an attacker who doesn't have adequate resources from decrypting even a single record. This cost is created using puzzles which require computational work to solve.

In this chapter, we'll first describe in detail properties that an abrasion function should accomplish in Section 5.1. We describe how to determine the security of an abrasion scheme in Section 5.2. Then we describe what the top-level functions (API) of an abrasion scheme should be in Section 5.3.

5.1 Properties of abrasion schemes

In order to discuss abrasion functionality effectively, we present desirable properties of abrasion functions in categories: Ideal, Relaxed, and Optional.

5.1.1 Ideal properties

These are properties that an ideal abrasion construction should do. Current constructions do not achieve all of these properties.

Property A - Cost to decrypt 1 message is equal to the cost to decrypt n messages.

Description: Our legitimate attacker will want to decrypt a large number of messages (n messages). Other, malicious, attackers may benefit from decrypting only a single message. For example, a resource-bound hacker might retrieve some encrypted records in a security breach. Abrasion should prevent this hacker from retrieving a single message, just as a strong encryption would. Ideally, the function would require the same cost for decrypting any number of messages. We refer to this cost as I for "initial cost." This relationship is shown in Figure 5.1.

Property B - Concrete lower bound on cost to decrypt 1 message.

Description: An abrasion construction should ensure that an attacker has to spend a one time cost (I) to start decrypting messages. Ideally, this lower bound is not probabilistic. When the work spent (w) is less than some fixed amount ($w < I$), the adversary should have negligible (2^{-128}) chance of decrypting any records. This property is also shown by the step-functions probabilities in Figure 5.1

Property C - Many can use a public key to encrypt.

Description: There are many companies that provide messaging services containing important information for investigations. We want each of these entities to be able to use abrasion to encrypt their records without having the key to decrypt each other's records. This is a property that many public key schemes have. Public keys can also be changed to ensure that different parties can require work done for different messages if desired. A real-world application would be if multiple countries used this to secure their digital evidence. We wouldn't want the precomputation

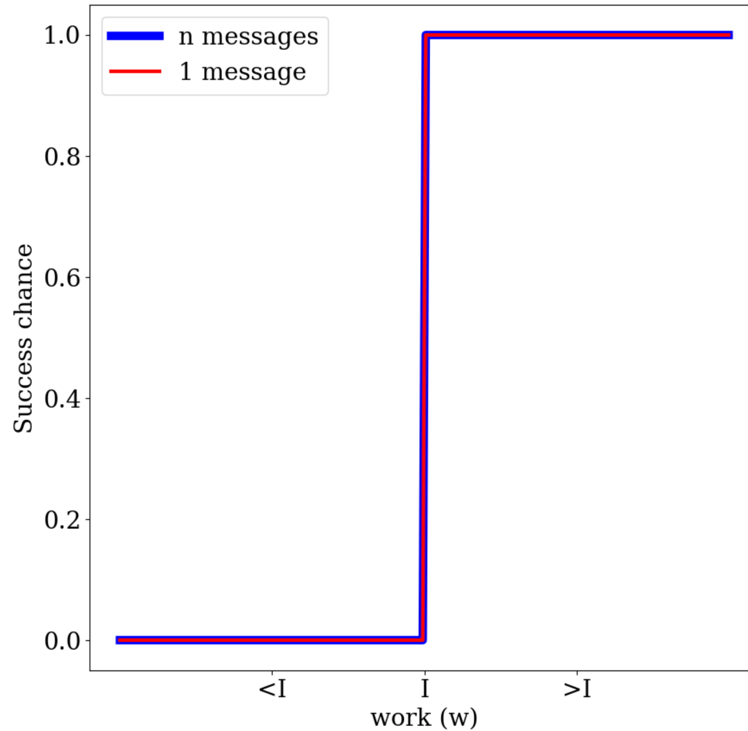

Figure 5.1: Message decryption chance in an ideal abrasion scheme

done by Australian law enforcement to allow decryption of American records or vice-versa.

Property D - Private key can be stored for future use.

Description: We want a legitimate attacker to be able to store a key to be used in future investigations. This motivates our construction to use TMTOs in order to be able to save a representation of the initial precomputation for later look up.

Property E - Predictable cost.

Description: The cost to decrypt is easily calculated and is based on

cryptographic functions with rigorous cost analysis.

Property F - Low cost per-message.

Description: After precomputation for a given public key, all future abraded key retrievals have very low cost.

5.1.2 Relaxed properties

These are properties from the ideal section that have been relaxed so we can analyze schemes that come close but are not perfectly ideal. Specifically, these represent Property A and Property B in a more attainable way.

Property G - Cost to decrypt 1 message is *close* to the cost to decrypt n messages

Description: Having these two costs be equal is ideal, as noted in Property A, but this is difficult to achieve using real-world constructions. This property relaxes the requirement of Property A. This relaxation is illustrated in Figure 5.2 by the malicious attacker (red line) rising earlier than the legitimate attacker (blue line). In order to stay "close" to the malicious attacker, we require our legitimate attacker's work to scale sublinearly with the number of messages decrypted. This relaxed property defines the "security" of an abrasion function. An abrasion scheme is secure against a malicious attacker if they must spend a large fraction ($\frac{1}{R}$) of the work that a legitimate attacker spends. We formally define this security in Section 5.2.2. In this work, our legitimate attacker uses $R = 13.2$ times the amount of work to decrypt 2^{45} messages compared to an illegitimate attacker's cost to decrypt 1 message. To put this in perspective, with only a crumpling scheme, an attacker who decrypts

2^{45} messages spends 2^{45} times the amount of work as an attacker that decrypts 1 message.

Property H - Probabilistic lower bound

Description: This relaxation of Property B allows the scheme to allow a small chance for a message to be decrypted with small cost. This means that at $w = I$ our illegitimate attacker has an ϵ chance of decryption. This ϵ is larger than 2^{-128} but still very close to zero. For this work, we use an ϵ of 2^{-45}. This relaxation is shown by the "s-curve" (or sigmoid) shape of the lines in Figure 5.2.

The differences between Figures 5.1 and 5.2 illustrate the relaxation of Property G and Property H. These two relaxed properties are more formally defined in Section 5.2.

5.1.3 Optional properties

These are desirable properties that improve the scheme, but aren't required for an abrasion function.

Property I - Freedom of key choice.

Description: Anyone can initialize the abrasion scheme and assure others that it is not compromised. Public keys for some asymmetric encryption can possibly have backdoors [55]. An abrasion function would be stronger if it were resistant to backdoors that would allow an attacker to subvert the initial cost I. For a scheme without this property, we believe that multiparty computation could be used instead [60, 38]. Using multiparty computation requires mass coordination and can have backdoors if they have a vulnerability [37] so it is not desirable.

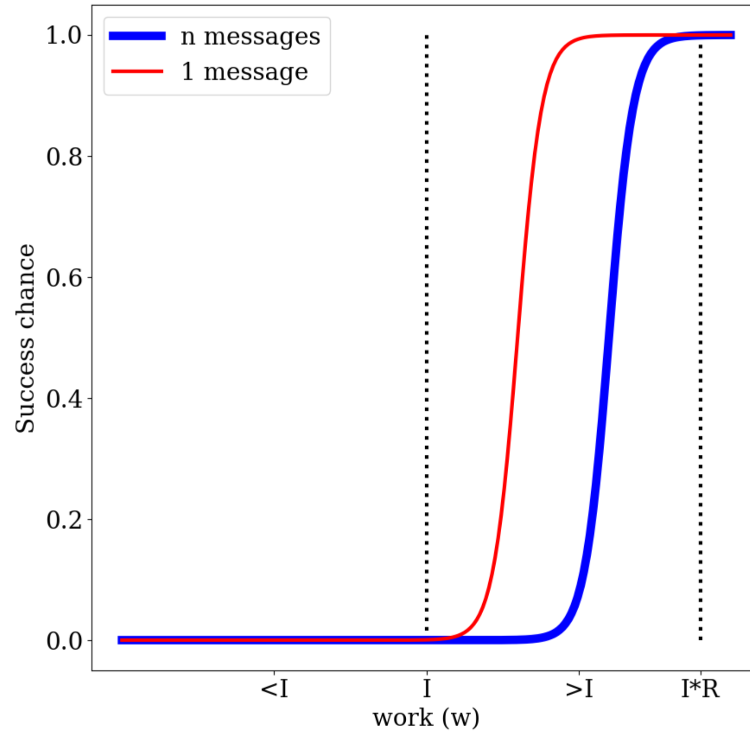

Figure 5.2: Message decryption chance in a realistic abrasion scheme

Property J - Low code impact on existing applications

Description: If existing programs can be modified easily, there is less of a chance of security breach due to programmer error

Property K - Non-transferable private key.

Description: Private key is hard to steal from a facility. In our case, the key is very large in size, requiring exfiltration of petabytes of information.

5.1.4 Practical abrasion

In order for an abrasion scheme to be practical, it must be able to fulfill requirements while not imposing a large cost on key storage and per-message decryption. We calculate these costs for our final construction in Chapter 8.

To further analyze a given abrasion scheme, we look at the budget of law enforcement agencies and the resources of malicious attackers. Over 2017 and 2018, the FBI requested almost $60 million for projects to solve the "Going Dark" problem [31, 45].

To compare this with the work that a malicious attacker could spend, we look at the value of data and the resources of attackers. If a hacker doesn't have enough resources, or if there isn't enough value to gain by breaking an abrasion scheme, their malicious activity will be prevented. Abrasion can't prevent attackers with large resources from decrypting records, but we provide examples of smaller attacks that could be prevented. The average value of stored PII for a data-driven company ranges from $3.6 million to $1 billion depending on the size of the company [57]. A breach doesn't usually result in all of a company's records being released. As a more specific example, a report showed that an individual data seller on the black market could make $2 million over 4 years by selling stolen credit cards [41]. As for the resources of cybercriminals, ransomware is easy to track because of public payments. Cryptolocker, a ransomware operation, resulted in $3 million in payments [1]. A smaller cybercrime operation profits around $50,000, but there are larger operations that profit over $1 billion [46].

In Chapter 8, we show that our abrasion construction can prevent attackers with less than $3 million from decrypting any records, while staying within the FBI's "Going Dark" budget. While there are larger cybercriminal operations that could break any abrasion scheme, these statistics should convince the reader that there are

attackers that would be thwarted by our practical abraded encryption scheme.

5.2 Security Definitions

In this section, we describe how to evaluate the security of an abrasion scheme. First, we give an idea for the threat model we plan to protect against in Section 5.2.1. Next we formally define security definitions based on this threat model in Section 5.2.2. In Chapter 7, we will use these definitions to evaluate the security of our final construction.

5.2.1 Threat model

In this section we describe our threat model for this scheme. To satisfy Property G and Property H we look at the probability of success of an adversary that simply wants to decrypt a single message. If the abraded encryption scheme meets these properties, this adversary should have to spend a proportional amount of work to our legitimate attacker. This legitimate attacker will store a private key for the abrasion scheme, whereas the malicious adversary does not store any of their computation.

To quantify this, we will be bounding the work done by a malicious adversary to decrypt one message. We will compare this to the work required for our legitimate attacker to derive a private key that can recover n messages. We calculate this comparison as a ratio shown in Equation 5.1.

A ratio is more useful than a difference between these costs (computed by subtraction). An insecure scheme could impose a fixed difference between a legitimate and malicious attacker by adjusting I. For example, an abrasion scheme could achieve a difference of $1 million by requiring $1 million for a legitimate adversary and $0 for a malicious adversary. It is more difficult for schemes to achieve a low ratio and in

our construction, the ratio does not change significantly as I is adjusted.

$$\text{Ratio}(R) = \frac{\text{legitimate work for } n \text{ messages}}{\text{malicious work for 1 message}} \tag{5.1}$$

For example, if the ratio is 2, a legitimate attacker has to spend twice as much work as a malicious attacker.

We also assume that this adversary has recovered an entire abrasion-encrypted database of g records while still only needing to reverse one.

The term "legitimate attacker" is left partially undefined to allow individual abrasion schemes to define this. In our scheme, the legitimate attacker precomputes a TMTO, which adds to their work required. We also leave "work" to be defined by the scheme. In our construction, "work" is the number of invocations of the hash function. We convert this work into dollars in Chapter 8.

5.2.2 Security Definitions

Definition 1. ϵ-n-g-R-Security

Let $w_{\text{legitimate}}$ be the work spent by a legitimate attacker to have a $1-\epsilon$ chance of solving n different abrasion tags.

Let $w_{\text{malicious}}$ be the work spent by any attacker to have an ϵ chance of solving 1 record, given g different abrasion tags.

An abrasion scheme is ϵ-n-g-R-secure if $R \leq \frac{w_{\text{legitimate}}}{w_{\text{malicious}}}$.

Definition 2. λ-R-Security

An abrasion scheme is λ-R-secure if it is ϵ-n-g-R-secure where $\epsilon = 2^{-\lambda}$, $n = 2^{\lambda}$, and $g = 2^{\lambda}$.

We present Definition 2 in order to compress the idea of Definition 1 to make it

easier to discuss. λ represents the idea of "very large" and "very small" parameters. An ideal scheme that achieves Property A and Property B would be 128-1-secure. Our construction instead meets Property G and Property H using a smaller λ and larger R.

We leave the cost I out of this security definition. Our scheme is easily tweaked to achieve different values of I with the same ratio (R). The value I may also vary from year to year as hardware efficiency increases.

Because there are many choices that this adversary could make, we need to find a way to calculate the bounds on their probability of success at any given amount of work. In Chapter 7 we look at an adversary that has access to a very powerful oracle function to show that our main construction is secure.

We evaluate our scheme as being 45-13.2-secure for $I \approx$ \$3 million in Chapter 8. This means that, with work below I ($w < I$), an adversary only has a 2^{-45} chance to successfully decrypt 1 message, given 2^{45} messages. This security is sufficient to deter many low-resource attackers while still not overwhelming the cost of government.

5.3 Abrasion scheme functions

An abrasion scheme is defined as a collection of functions used to facilitate abraded encryption. We will list them in this section.

5.3.1 Abrasion scheme initialization

$$pk \leftarrow \text{AbrasionSchemeInitialization}(Q, \lambda, R)$$

The initialization function takes a cost parameter Q and security parameters λ, R. It returns an abrasion public key (pk) and may generate public parameters for use in

other functions of the scheme. This cost Q is distinct from I (the initial cost) as I will change as hardware becomes more efficient. Given specific hardware, Q should be proportional to I. In our main construction, Q is the number of invocations of the hash function. This initialization function is deceiving as, for our main construction, we calculate R using other parameters from the scheme and acceptable values for R are found through trial and error.

5.3.2 Abrasion tag generation

$$a \leftarrow \text{AbradeKey}(pk, k)$$

To perform an abraded encryption, we "abrade" the key given to us (k), then use k for the symmetric encryption of a message. To abrade the key, we compute an abrasion tag (a) and present it to law enforcement along with encrypted records upon warrant. k itself is not be given to law enforcement. The tag (a) is breakable for an abrasion private key (sk) computed using pk. This tag allows an attacker to retrieve k after performing the initial cost of computing sk.

Messaging service providers will use this function to abrade their keys used to store records. We call the applications that compute abrasion tags as "clients." Generally in client-server schemes, the client does less work, similar to the application that performs the abraded encryption. We do not have the notion of a "server" in abrasion schemes.

5.3.3 Abrasion private key generation

$$sk \leftarrow \text{AbrasionPrivateKeyGeneration}(pk)$$

This scheme will generate a private key based on a public key. Deriving this private key is intended to take $R * Q$ work to compute.

5.3.4 Abraded key retrieval

$$k \leftarrow \text{AbradedKeyRetrieval}(sk, a)$$

Using an abrasion private key sk, this function will retrieve the abraded key k associated with an abrasion tag a. If our scheme is secure there is almost no chance of any attacker reversing k from a without first spending Q work. This notion of security was defined earlier in this section and is required to meet Property G and Property H.

Chapter 6

Abrasion constructions

In this chapter we will introduce our main construction in Section 6.3. But first, we introduce simpler, but insecure, abrasion constructions to help the reader understand the concept of a hash-based abrasion scheme. These simpler constructions are presented in Sections 6.1 and 6.3.1. We explain why these constructions fail to meet our requirements for an abrasion function in Section 6.2.

6.1 A simple (but insecure) hash-based abrasion scheme

In this section, we will show the reader how a simple hash-based abrasion scheme could work. This scheme is not secure, but introduces concepts that are used in more secure schemes. We define a secure scheme as one that meets our security definitions defined in Section 5.2.2.

We can see from Property D and Property F that we will need a way to allow our legitimate attacker to compress precomputation into a “private key” that we can store and reuse. We will use hash functions and TMTOs to achieve this.

To ensure our scheme costs millions of dollars, we must do many hash computations. Without TMTOs, storing the results of all these computations would take zettabytes of storage. This would be impractical and means our construction must accommodate a legitimate attacker that computes a TMTO.

A malicious attacker will forego TMTO creation to decrypt a single message quickly. Because of this, we refer to a malicious attacker as a *tableless attacker* and a legitimate attacker as a *table attacker*.

TMTOs can be used to reverse hash results. This means that our legitimate adversary can reverse hashes we store in the abrasion tag to recover the preimage. To allow for decryption, the hash function's input space must be restricted. A full sized hash function is computationally intractable to reverse, even with TMTOs A smaller search space is possible to iterate through with some cost. We measure the cost by calculating the number of hash function computations needed to reverse the abrasion tag.

To ensure that reversing a preimage will allow the attacker to defeat the puzzle, we store a "weakened" hash result of a nonce. An attacker can then reverse the hash function to retrieve this nonce. We use this nonce to encrypt the key and store the ciphertext in the abrasion tag. We call this process "abrading" the key.

We assume that hash functions are random oracles. Random oracles will deterministically give us truly random bits. The output of a random oracles can only be reversed if the exact preimage is guessed again. Assuming this property makes security assumptions easier to prove and is a commonly used model [26].

To create a weakened hash, we will use a strong hash function $\big(H_s(\cdot)\big)$ and reduce the size of the unknown input space. This strong hash has a large, deterministic, truly random output $\approx$ 256-bits. Reducing the input size is done by feeding the hash function a public key pk concatenated with a short random nonce x. The length of the nonce dictates the initial cost of the scheme.

We now describe this scheme (labeled as "SingleHash") using the interfaces declared in Section 5.3.

An assignment of the form: $r \xleftarrow{\$} \{0,1\}^k$ is a random assignment of r from the set

of all bit-strings of size k.

We use a symmetric encryption scheme (Enc, Dec), such that:

$$c = \text{Enc(k,m)}$$

$$m = \text{Dec(k,c)}$$

6.1.1 Abrasion scheme initialization

Algorithm 1 $pk \leftarrow$ SingleHash::AbrasionSchemeInitialization(Q, λ, R)

$pk \xleftarrow{\$} \{0,1\}^b$

$N \leftarrow Q$

N is used as a parameter for other functions in this scheme. It defines the size of the weakened hash function input space. Setting this equal to Q will ensure that our initial cost I for this scheme is proportional to Q. The bit length of pk (b) is large ($\approx$ 256-bits) so that it is difficult to compute a TMTO that solves multiple abrasion schemes with different pk.

6.1.2 Abrasion tag generation

Algorithm 2 $a \leftarrow$ SingleHash::AbradeKey(k, pk)

$x \xleftarrow{\$} \{0, 1, ..., N-1\}$

$a \leftarrow \{a_h, a_c\} =\leftarrow \{H_s(pk||x), \text{Enc}(x, k)\}$

x is the nonce used to encrypt the abraded key k. x is discarded after this computation.

6.1.3 Abrasion private key generation

Algorithm 3 $sk \leftarrow$ SingleHash::AbrasionPrivateKeyGeneration(pk)

$sk \leftarrow$ BuildTMTO($H_s(pk||\cdot)$ mod N)

The legitimate attacker now computes a TMTO on the cryptographic function $h(\cdot)$ where $h(x) = H_s(pk||x) \bmod N, x \in \{0, 1, ..., N-1\}$. This TMTO is the private key sk. We label the function that builds this table as "BuildTMTO," which takes a cryptographic function and builds a TMTO to reverse it. The process of creating a TMTO is described in section 3.1. We calculate the cost of a legitimate attacker assuming that they use Hellman's TMTO [40].

6.1.4 Abraded key retrieval

Algorithm 4 $k \leftarrow$ SingleHash::AbradedKeyRetrieval($sk, (a_h, a_c)$)

$x \leftarrow$ Lookup(sk, a_h)

$k \leftarrow$ Dec(x, a_c)

After precomputation, they will be able to reverse many hash results by performing a lookup in the TMTO. This lookup takes substantially less work than precomputation.

Unfortunately, our table (legitimate) attacker needs to waste a lot of work. This is because, in order to build a TMTO table that will retrieve close to 100% of messages, much of the work will be duplicated due to merging chains in the TMTO tables. Chain merges are described in Section 6.2. The probability of success using a TMTO to reverse a single hash function levels off at $\approx$ 86%. The malicious (tableless) adversary does not need to waste their work creating this table, and therefore spends

far less work. Because of the large difference in work, this simple abraded encryption therefore does not satisfy Property G. We will explore this problem in Section 6.2.

6.2 TMTO success chance

To ensure the tableless (malicious) adversary has to spend resources similar to an attacker that computes a table, we will need to analyze the success chances of various TMTO strategies.

We derive the success chance of Hellman tables in Appendix A.

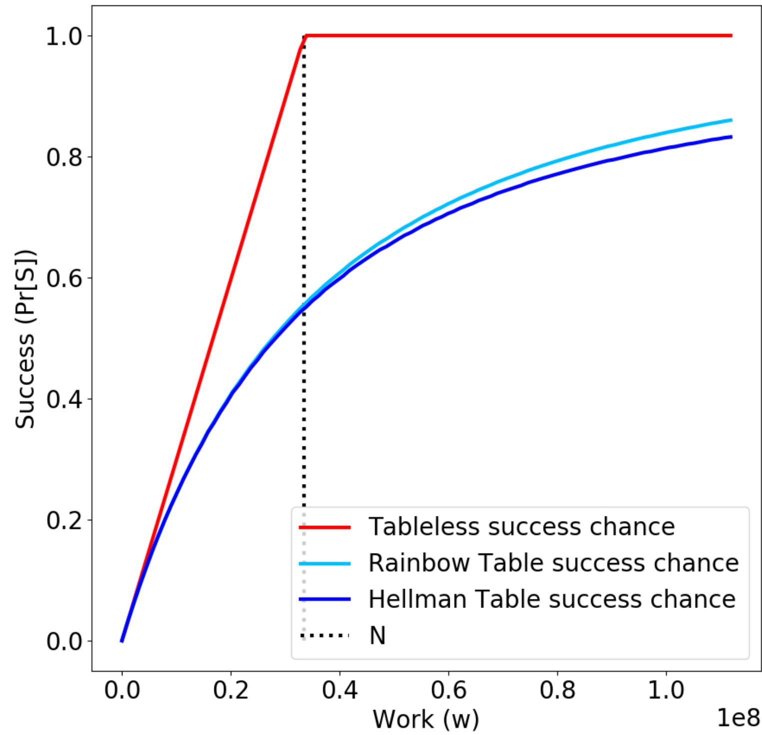

Figure 6.1: Probability of success of different TMTO solutions (up to $\approx 86\%$) vs tableless approach, $N = 2^{25}$

We have included the success chance of Hellman's TMTO solution in Figure 6.1.

As can be seen from this graph, the probability of success levels off as more work is put into the table. This gives us diminishing returns as we put more work into the table. If we used a single hash as an abrasion function, this drop off of efficiency would severely limit a table attacker compared to the tableless attacker.

The probability of success of a rainbow table was described by Oechslin in 2003 [50]. The success chance of a rainbow table attack is graphed against work in Figure 6.1. We can see that rainbow tables perform slightly better than Hellman's TMTO solution. Rainbow tables were not evaluated as a method for our legitimate attacker as Hellman tables are much better understood and there is a greater depth of research into them [25].

We did not measure the success probability of tables built with distinguished points (DPs). We discuss DP tables in Section 3.1.

Figure 6.1 graphs the success chance of a tableless attacker that only has one abrasion tag. When an attacker has multiple abrasion tags, they have an even greater chance of breaking a single message. This is because all abrasion tags share the same weakened hash functions. This increases the chance of an attacker recovering a preimage while iterating through the input space by allowing them to check if a hash result solves any of the abrasion tags available to them. Our main construction in Section 6.3 is provably secure against an adversary with many abrasion tags and we calculate the security ratio (R) of our main construction in Chapter 8.

6.3 Main construction

In this section, we first describe the intuition in for our main construction Section 6.3.1. We then formally present the main construction in Section 6.3.2.

6.3.1 Intuition

The graph in Section 6.2 shows that using a single weakened hash function to create an abrasion scheme results in an insecure scheme. With one hash function, a tableless attacker gains a reasonable success chance with less work than a table attacker. Using multiple distinct hash functions can modify the success chance of these different attackers. In this section we will introduce two insecure schemes, labeled as: “MultipleSame” and “MultipleDifferent.” These schemes will illustrate methods and statistics that our main construction uses in Section 6.3.2.

First we examine what happens when we modify the “SingleHash” scheme to use multiple hashes on the same preimage (nonce ‘x’). This scheme is called “Multiple-Same” as we hash the *same* preimage *multiple* times. We do not describe the functions for “MultipleDifferent” that do not vary much from the “SingleHash” scheme such as the function for initialization.

The “AbradeKey” function now generates a modified abrasion tag. Assume Y is a scheme parameter generated during initialization. Y will determine how many distinct hash functions there are.

Algorithm 5 $a \leftarrow \text{MultipleSame::AbradeKey}(k, pk)$

$x \overset{\$}{\leftarrow} \{0, 1, ..., N-1\}$

$\forall i \in \{0, 1, ..., Y-1\}, ah_i \leftarrow H_s(pk + i || x) \bmod N$

$a_c \leftarrow \text{Enc}(x, k)$

$a \leftarrow \{\{\forall i, ah_i\}, a_c\}$

The retrieval function is similar to the simple scheme in Algorithm 4. The difference is that now the attacker has multiple ah_i and can choose which one to reverse.

Because we add i to pk while computing $H_s(\cdot)$, each hash function is distinct. This

means that our table attacker can build multiple TMTO solutions. These TMTOs will have no collisions or merges with each other because they are computing distinct cryptographic functions. Because they only have to reverse one of the hashes, their probability of success increases dramatically. With a large value for Y, we can boost an attacker's success chance to around 100%.

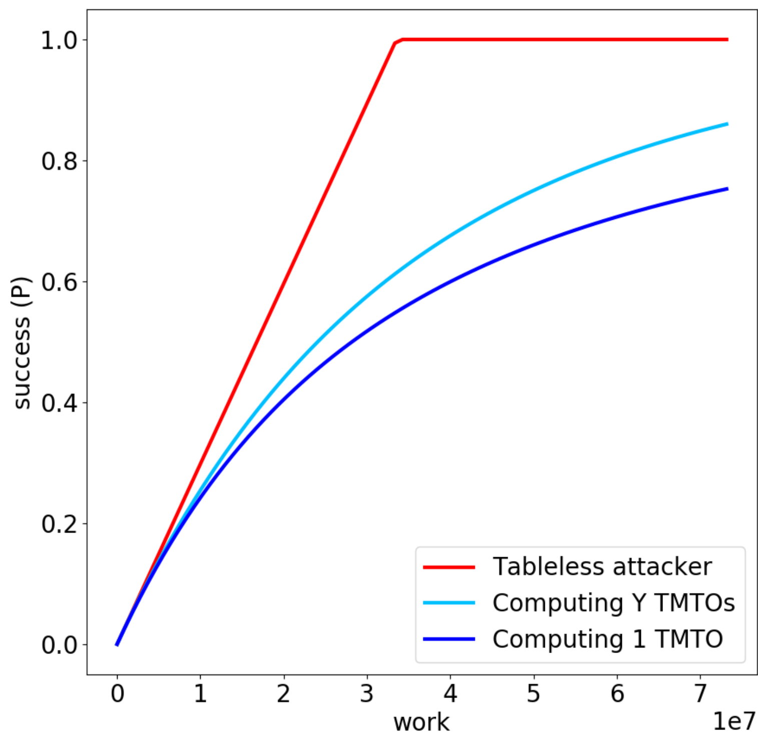

Figure 6.2: Tableless vs table attacker, MultipleSame construction. $Y = 5, N = 2^{25}$

Unfortunately, our tableless attacker retrieves the nonce very quickly as well, making this scheme insecure. As shown in Figure 6.2, our tableless attacker immediately has a significant chance to defeat the abrasion puzzle with very low work. A malicious adversary could get lucky and decrypt a record with a small amount of work in the "MultipleSame" scheme.

A different way to modify the scheme is to use different preimages for the distinct hashes. We concatenate all the preimages and hash it to derive a key. We will use this key to encrypt our abraded key. Because this key is never stored, we call it the ephemeral key (e). We call this modified scheme "MultipleDifferent," because we use different preimages to compute multiple hash results.

Algorithm 6 $a \leftarrow$ MultipleDifferent::AbradeKey(k, pk)

$\forall i \in \{0, 1, ..., Y-1\}, x_i \xleftarrow{\$} \{0, 1, ..., N-1\}$

$\forall i \in \{0, 1, ..., Y-1\}, ah_i \leftarrow H_s(pk + i || x_i) \bmod N$

$e \leftarrow H_s(x_0 || x_1 || ... || x_{Y-1})$

$a_c \leftarrow \text{Enc}(e, k)$

$a \leftarrow \{\{\forall i, ah_i\}, a_c\}$

Now an attacker has to retrieve all Y preimages to derive the ephemeral key required to decrypt k.

Algorithm 7 $k \leftarrow$ MultipleDifferent::AbradedKeyRetrieval(sk, a)

$\forall i \in \{0, 1, ..., Y-1\}, x_i \leftarrow (\text{Lookup}(sk, ah_*))$

$e \leftarrow H_s(x_0 || x_1 || ... || x_{Y-1})$

$k \leftarrow (\text{Dec}(e, a_c))$

Modifying the scheme this way means that our tableless attacker has a much harder time. There is a much smaller chance that they will get lucky and decrypt all Y preimages. Specifically, to calculate their success, we sum uniform distributions. Summing these distributions normalizes the tableless attacker's chance of success and ensures that at a small amount of work, they have little chance of success. The success chance for various attackers in the "MutlipleDifferent" scheme is shown in Figure 6.3. As can be seen, the success chance of the tableless attacker is starting to resemble the graphs in Figure 5.2, used to illustrate Property G and Property H.

The "MultipleDifferent" scheme has a different problem from the "MultipleSame" scheme. The table attacker has a much harder time increasing their chance of success. They must put large amounts of work into all the tables to ensure they can recover many messages. They lose most of their work due to chain merges while building these large TMTOs.

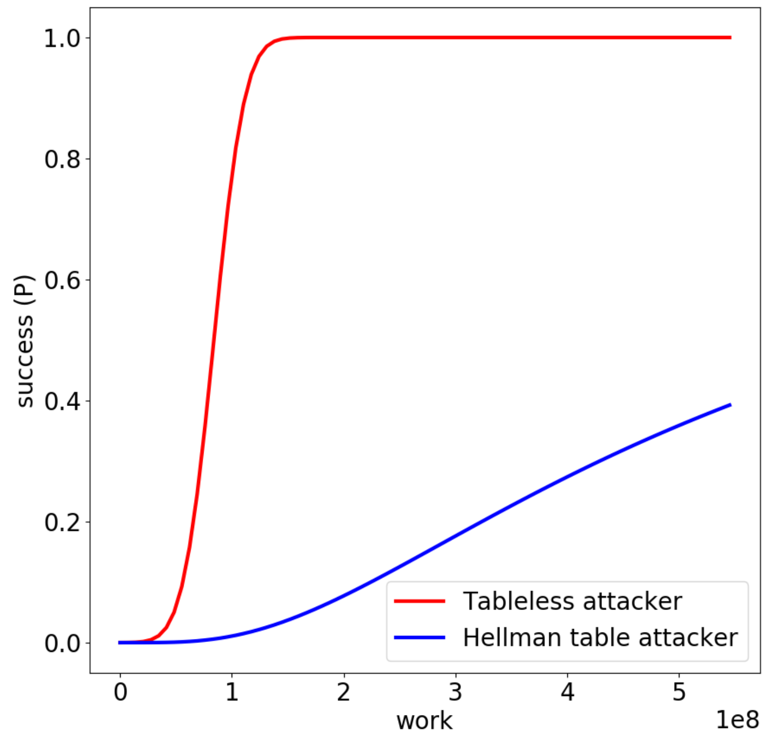

Figure 6.3: Tableless vs table attacker, MultipleDifferent construction. $Y = 5, N = 2^{25}$

6.3.2 Description

In this section, we present our main construction. In this construction, the attacker must retrieve a subset X of the Y preimages (x_i). We show that this gives

us the desired properties of both of the two previous schemes, "MultipleSame" and "MultipleDifferent." Our main construction ensures that the tableless attacker does not have a significant chance of decrypting a message until they've done a significant amount of work. At the same time, this scheme allows our table attacker to retrieve many messages without wasting too much work. The table attacker does not need to compute unreasonably large TMTOs to achieve $1 - \epsilon$ success chance. We call this scheme the "X-of-Y" abrasion scheme because it requires the attacker to reverse X of Y hashes. In the following sections, we define the functions that make up our construction. These functions follow the outline for an abrasion scheme described in Section 5.3.

We review the variables used in this section in Table 6.1.

Assume $\text{SSGen}(k, m, n)$ is a secret sharing generator and $\text{SSDecrypt}(c, s')$ retrieves those secrets as described in Section 3.2.

Assume $(c \leftarrow \text{Enc}(k, m))$ is a strong ($256 - bit$) authenticated symmetric encryption with related decryption function $(m \leftarrow \text{Dec}(k, c))$. Because it is authenticated, an authentication tag will be generated with with the ciphertext. The authentication tag is used by the attacker to verify that they've retrieved the correct preimage. If there were no way to make this verification, our attacker would not know when they had broken the puzzle. We address the security of this authentication tag and the ciphertext in Section 7.4.

Table 6.1: Important variable definitions

Variable	Meaning
I	Required cost in dollars to decrypt 1 message
Q	Required work in hash invocations to decrypt 1 message
R	Work ratio, legitimate attacker over malicious attacker
Y	Number of distinct hash functions/preimages
X	Required reversals/preimages (out of Y)
N	Search space of hash function
λ	Security parameter that defines g, n and ϵ
g	Number of abrasion tags given to malicious attacker (2^{λ})
n	Number of abrasion tags decryptable by legitimate adversary (2^{λ})
ϵ	Success chance of malicious attacker ($2^{-\lambda}$)
$1-\epsilon$	Success chance of legitimate attacker
$\Pr[S_*]$	Success chance
w	Work (in hash invocations)
l, m, t	TMTO parameters
k	Key to be abraded
pk	Public key (bit string)
b	Bit length of strong key ($b \geq 256$)
sk	Private key (set of TMTOs)
x	Preimage
H_s	Strong hash function
ah	Abrasion hash
e	Ephemeral key
s	Secret shares
sc	Secret share ciphertext
c	Extra ciphertext of secret sharing

6.3.3 Abrasion scheme initialization

Algorithm 8 $pk \leftarrow$ XofY::AbrasionSchemeInitialization(Q, λ, R)

$pk \xleftarrow{\$} \{0,1\}^b$

$X \approx \frac{Y}{2}$

$N \leftarrow \frac{Q*R}{Y}$

The public key (pk) is generated in the same way the simple hash scheme generates the public key in Section 6.1.1.

N is the size of each weakened hash function. Using the *matrix stopping rule*, our attacker will spend N work on each of the Y hash functions. To ensure that our scheme costs at least Q work to break at the given R, we must set N appropriately so that $N * Y = Q * R$.

In order to achieve a small R a large Y must be used. X is set appropriately to ensure the table attacker has $1 - \epsilon$ probability of success when building tables where $w = N$. This leads to a value of X that is about half of Y. We calculate the success chance of our table attacker for this construction in Section 6.3.10.

6.3.4 Abrasion tag generation

Algorithm 9 $a \leftarrow$ XofY::AbradeKey(pk, k)

$\{s, c\} = \text{SSGen}(k, X, Y)$

$\forall i \in \{0, 1, ..., Y - 1\}, x_i \xleftarrow{\$} \{0, 1, ..., N - 1\}$

$\forall i \in \{0, 1, ..., Y - 1\}, ah_i \leftarrow H_s(pk + i || x_i) \bmod N$

$\forall i \in \{0, 1, ..., Y - 1\}, e_i \leftarrow H_s(pk + 2 * Y + i || H_s(pk + Y + i || x_i))$

$\forall i \in \{0, 1, ..., Y - 1\}, sc_i \leftarrow \text{Enc}(e_i, s_i)$

$a \leftarrow \{\{\forall i, ah_i, sc_i\}, c\}$

The key portions of Algorithm 9 are depicted in Figure 6.4 which shows how each element in the abrasion tag is derived.

First, we run secret sharing to retrieve the secret shares s. Then we pick Y random nonces from the search space. The abrasion hashes are then generated (ah_*). These are meant to be attacked during decryption. Then we generate the ephemeral keys (e_*) using different public keys for the hash function.

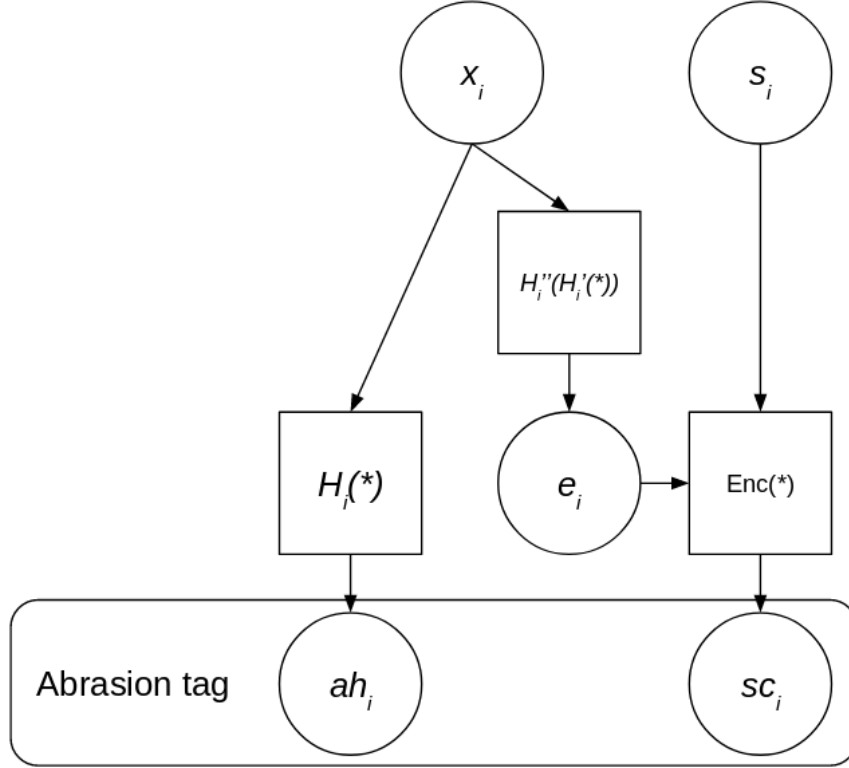

Figure 6.4: Generation of each element in the abrasion tag in Algorithm 9

We compute the hash twice while generating the ephemeral keys to ensure that they are not the target of an attack. If we simply used x_i to encrypt each share, an attacker could iterate through the input space and search for the given MAC to decrypt s_i. We discuss this attack in Section 7.4.

The secret shares are then encrypted with the ephemeral keys. All e_i and s_i are discarded at this point. This leaves us with only the ciphertexts of the shares (sc_*).

The abrasion tag, 'a,' is now stored on disk to be presented upon warrant along with records encrypted with k. The abraded key, 'k,' can be kept in secure memory to do a number of abraded encryptions. This helps amortize the disk space and computation usage by the client as well as cost-per-message for the table attacker. This key should be scrubbed intermittently and the generation function should be run

again to get a new key and tag. Using the same k for long periods of time increases the damage done by data breaches as an attacker will be able to decrypt more messages with a single k. This enhancement is considered only to increase the performance of the client. In this thesis, we considered breaking a single tag to be synonymous with decrypting a single message, ignoring the fact that messaging platforms could use k to encrypt multiple messages.

6.3.5 Abrasion private key generation

Algorithm 10 $sk \leftarrow$ XofY::AbrasionPrivateKeyGeneration(pk)

$\forall i \in \{0, 1, ..., Y-1\}, sk_i = \text{BuildTMTO}(H_s(pk + i||\cdot) \text{ mod } N)$

The legitimate attacker now computes Y TMTOs, one for each of the distinct hash functions. The function "BuildTMTO" is described in Section 6.1.3.

Their work is split evenly among the different TMTOs. This maximizes the effectiveness of their work. To achieve $\approx 100\%$ success, they will need to compute each hash function $\frac{Q*R}{Y} = N$ times. This amount of work follows the "matrix stopping rule" described in Section 6.2.

6.3.6 Abraded key retrieval

Algorithm 11 $k \leftarrow$ XofY::AbradedKeyRetrieval($sk, a = \{\{ah, sc\}, c\}$)

$\forall i \in \{0, 1, ..., Y-1\}, x_i \leftarrow \text{Lookup}(sk_i, ah_i)$

$\forall i \in \{0, 1, ..., Y-1\}, e_i \leftarrow H_s(pk + 2 * Y + i||H_s(pk + Y + i||x_i))$

$\forall i \in \{0, 1, ..., Y-1\}, s_i \leftarrow \text{Dec}(e_i, sc_i)$

$k = \text{SSDecrypt}(c, s'), s' \subset s, |s'| = X$

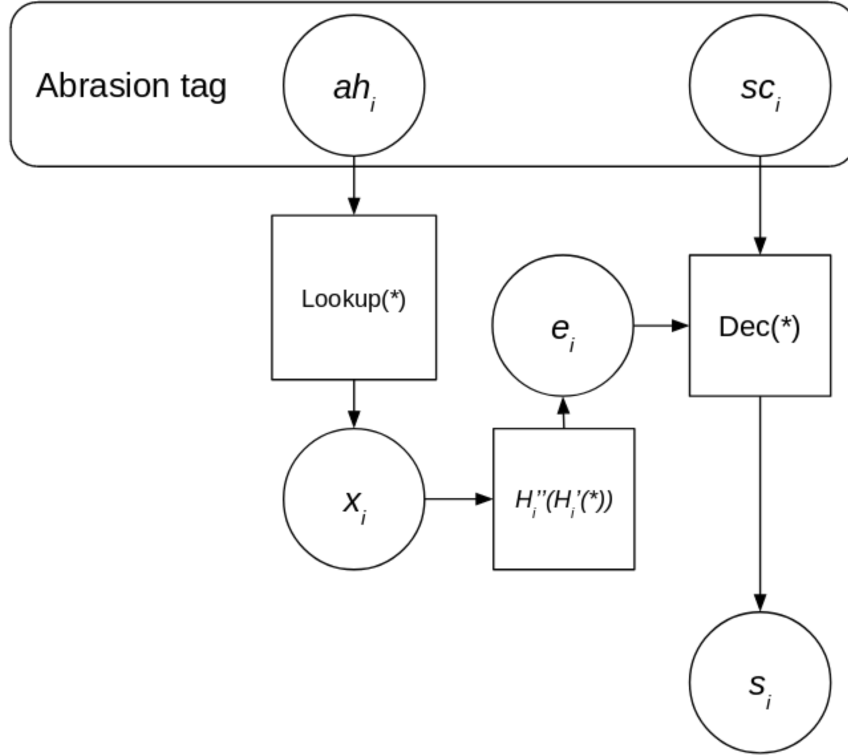

Figure 6.5: Retrieval of each share in an abrasion tag in Algorithm 11

We illustrate the key parts of Algorithm 11 in Figure 6.5. This diagram complements Figure 6.4 to show both encryption and decryption in our abrasion construction.

After computing each of the TMTOs, a table attacker will lookup each of the abrasion hashes (ah_i) in their tables. They can stop once they've retrieved X preimages (x_i). The attacker then regenerates the ephemeral keys (e_i) and uses them to decrypt the shares (s_i). Using the X shares they've decrypted, the attacker can retrieve the abraded key k from the secret sharing scheme using SSDecrypt. Because we only require that $|s'| \geq X$, many of the lookups in the TMTOs can fail without preventing the table attacker from decrypting the abrasion puzzle.

6.3.7 Cost calculations

A table attacker performs $Q * R$ hash function invocations. This work is evenly split among the Y distinct hash functions, creating a different TMTO for each of the hash functions.

Using Hellman's TMTO, the storage for each TMTO (M) will be $m * l$ rows as each TMTO contains l tables of m rows. The total storage will be $Y * m * l$ rows. Because our table attacker splits their work evenly among the TMTOs, $m*l*t = \frac{Q*R}{Y}$, where m, l, and t parameterize each of the TMTO. Because we use the standard trade off, $m = l = t = \left(\frac{Q*R}{Y}\right)^{\frac{1}{3}}$. This makes the total number of rows, $Y * m * l$, equal to $Y * \left(\frac{Q*R}{Y}\right)^{\frac{2}{3}}$.

For each row, we need to store the starting point and the end point. We can represent all m starting points for each table in $log_2(m)$ bits and each ending point of the weakened hash as $log_2(N)$ bits. Although there are more possible starting points (up to N), we only need $log_2(m)$ bits to store all possible values for a single TMTO table. To vary starting points between tables, the index of the table is appended to the starting point when computing chains. This method was shown by Barkan et al. [25].

In the online phase, our table attacker will search at most Y TMTOs to retrieve the X preimages. Searching a single TMTO requires $T = t_{\text{table}} * l = t * 1.5 * l$ invocations of the hash function. This means our online cost to reverse an abrasion tag (T_{tag}) is $T_{\text{tag}} \leq T * Y = t * 1.5 * l * Y$. The 1.5 in this calculation accounts for false alarms as described in section 3.1.

To create an abrasion tag, the client will have to invoke the hash function multiple times. Each weakened hash result requires an invocation of the hash function and each ephemeral key requires two invocations of the hash function. Because there are

Y hash results and Y ephemeral keys, this means the client will compute the hash function $Y * 3$ times for each abrasion tag.

In Section 6.3.8 we explain how we can redefine the hash function (H_s) to modify these costs.

6.3.8 Decisions, details and alternatives

We can modify $H_s(\cdot)$ independent of the rest of the scheme. Equation 6.1 represents defining H_s as computing DoubleSHA256 u-times. Increasing u increases the cost of our attackers (Q) without increasing the space required to store the private key.

$$H_s(\cdot) = (\text{DoubleSHA256})^u(\cdot) \tag{6.1}$$

Running DoubleSHA256 multiple times will also increase the cost of the client and online cost of the table attacker. To keep our scheme practical, we only increase u to 500. Each of these invocations of DoubleSHA256 is performed using a distinct public key (derived from pk) to prevent stored preimages from helping a malicious adversary. We describe how even a malicious attacker could store computations to break a single tag faster if we used indistinct hash functions in Section 7.2. To make our construction secure, we use large numbers of shares that are not supported by the popular secret sharing libraries [11, 12]. The best performance analysis of secret sharing schemes currently states that generating a number of shares around $Y \approx 40$ takes about .22 seconds [21]. We speculate that an implementation that scales to meet our needs would not be difficult to create and would still be performant. If future work finds that secret sharing is not practical at this scale, an alternative scheme could be used. We describe this alternative in Section 6.3.9.

6.3.9 Probabilistic sharing

Secret sharing schemes are designed to break up keys to split among different parties. We discuss secret sharing in Section 3.2. These schemes require hard guarantees on the number of keys required to decrypt the secret. With probabilistic sharing, we forego these properties to relax the computational intensity.

The idea of this scheme is to statistically require X shares out of Y. The scheme assumes that shares are found through blind guessing. Any attacker must blindly guess preimages in the random oracle model, so this assumption should hold for our abrasion scheme.

With regards to our hash abrasion scheme, probabilistic sharing replaces SSGen and SSDecrypt with PSGen (Probabilistic Share Generation) and PSDecrypt (Probabilistic Share Decryption) respectively.

PSGen creates a ciphertext c, along a pool of Y shares ($|s| = Y$) similar to SSGen. Some number of shares ($Y - d$) are duplicates of other shares. There are only d distinct shares.

$$\{s, c\} = \text{PSGen}(k, X, Y), X \leq Y$$

PSDecrypt is a function to retrieve k.

$$k = \text{PSDecrypt}(c, s')$$

.

The set, s', contains only the distinct shares of s. This means the size of s' ($|s'|$) is d. The value of d is set so that while reversing preimages blindly, the expected number of reversals is X. d will generally be less than X ($d \leq X$).

An ephemeral key (e) is generated during PSGen.

$$e = H_s(s'_0||s'_1||...||s'_{d-1})$$

This ephemeral key is used to encrypt the secret k using a randomized encryption and the ciphertext (e) is returned along with the pool of shares (s).

$$c \leftarrow \text{Enc}(e, k)$$

We do not analyze how using this would affect our security. We propose probabilistic sharing as an option to be considered if traditional secret sharing becomes too costly.

6.3.10 Main construction success chance

We describe TMTOs in Section 3.1 and derive an approximation of a lower bound on the success chance of multiple Hellman tables in Appendix A.

A binomial distribution can be used to model the probability of exactly X successes out of Y trials given probability p. Our table attacker splits their work evenly among Y different TMTOs, each giving the same probability of success ($\Pr[S_{\text{TMTO}}]$). Because we only require X out of Y hashes to be reversed successfully, our table attackers success chance to reverse a single abrasion tag ($\Pr[S_{\text{tag}}]$) can be calculated with the binomial distribution as shown in Equation 6.2. Where $\text{Binomial}(k, n, p) = \binom{n}{k} \cdot p^k(1-p)^{n-k} = \Pr[k \text{ successes out of } n \text{ trials, each with } p \text{ chance of success}]$

$$\Pr[S_{\text{tag}}] = \sum_{i=X}^{i \leq Y} \text{Binomial}(i, Y, \Pr[S_{\text{TMTO}}]) \tag{6.2}$$

Because our legitimate attacker wants to recover n messages, we multiply our success n times, shown in Equation 6.3.

$$\Pr[S_{\text{legitimate}}] = \prod_{i=0}^{n} \Pr[S_{\text{tag}}]$$
$$\Pr[S_{\text{legitimate}}] = \left(\sum_{i=X}^{i \leq Y} \text{Binomial}(i, Y, \Pr[S_{\text{TMTO}}]) \right)^{n} \quad (6.3)$$

By using a lower bound for $\Pr[S_{\text{TMTO}}]$, we can calculate a lower bound for $\Pr[S_{\text{legitimate}}]$ using Equation 6.3. When computing the work of our legitimate attacker ($w_{\text{legitimate}}$), we evaluate what value of w causes Equation 6.3 to equal $1 - \epsilon$ where ϵ is very close to zero. This calculation is used in Section 8 to evaluate the security of our scheme.

Chapter 7

Security

In this chapter, we define our legitimate attacker in Section 7.1. A security game with an oracle is presented in Section 7.2. This game and oracle are specifically created for our main construction, described in 6.3.2. This security game is not indended to be used to evaluate other constructions. We then show how to calculate the success chance of an adversary in this security game in Section 7.3. The security game and oracle allow us to find values (λ, R) for Definition 2 from Section 5.2. We also mention other attacks in Section 7.4 and explain how our construction is security against them.

7.1 Legitimate attacker definition

Our legitimate attacker creates a private key that can be used to decrypt n abraded keys with $1 - \epsilon$ probability, where n is a security parameter from Definition 1. We calculate the success of this attacker in Section 6.3.10. We define "work" in our construction as computations of the hash function $H_s(\cdot)$. This legitimate attacker does not actually have to decrypt each message, but only create a private key that has the potential to decrypt n messages. Leaving the cost of each decryption out of this definition is realistic as our per-message cost will be offset when we combine this scheme with a crumpling scheme. We set 2^λ to be much larger than the number

of messages that law enforcement will decrypt. This large chance of success ensures that law enforcement will never be unable to decrypt an important record with their computed private key.

7.2 Security game and oracle description

To calculate the security ratio in Definition 2, we will need to bound the success chance of any adversary. Any attack will involve searching the input space of hash functions to find preimages. An attacker can switch their focus among different hash functions during this search. While attacking a set of abrasion tags, there are many preimages to find in each hash function. This gives the adversary many choices to make. For example, one strategy is switch to searching untried hash functions after finding a single preimage. Another strategy could be to reverse many preimages from a single hash function and then move on to get a better chance of finding matching preimages. While continuing to find preimages, the attacker could have many combinations of matching preimages, each suggesting a different optimal decision on which hash function to search next. Instead of calculating the optimal decision at any point, we create a security game and give an adversary access to a powerful oracle. This game and oracle are crafted in a way that makes it easy to calculate the success chance of the optimal strategy and ensures that this bounds any adversary without the oracle.

While describing this game and oracle, we will represent any set of the form: $\{0, 1, ..., x-1\}$ as $\mathbb{Z}_x$.

We assume that hash functions are random oracles in this security proof. This is described in Section 6.1 and is commonly used in security proofs [26]. This allows us to predict exactly how much work is required to reverse a hash.

In this security game, the adversary can only query the oracle once.

First the adversary is given g different abrasion tags. Viewing these tags, the adversary selects a permutation of the input space for each hash function. This is the order in which the adversary will pick preimages to compute and compare with the abrasion hashes, ah. The adversary can choose different orderings for each hash function and must choose these orderings before querying the oracle. The adversary must use these orderings to search the hash functions after querying the oracle. Fixing these orderings is a reasonable restriction as there is no information (without this security oracle) that would suggest a better search order. There is no information to be gained because we treat these hash functions as truly random functions

The adversary submits 2 values to the oracle: the order of preimages which they will search for in each hash function: $f_j, j \in \mathbb{Z}_Y$, and the g abrasion tags they have: $a_i, i \in \mathbb{Z}_g$.

These search orderings (permutations of $\mathbb{Z}_N$) that the adversary gives to the oracle are represented by Y bijective functions $f_j, j \in \mathbb{Z}_Y, f_j(y) \neq f_j(z), y \neq z$. Each order function has the same input and output space of $\mathbb{Z}_N$ ($f_j : \mathbb{Z}_N \rightarrow \mathbb{Z}_N$).

The adversary can generate all permutations ($f_j, j \in \mathbb{Z}_Y$) at no cost. All communication between the adversary and the oracle is considered to be free when calculating the adversary's work.

The oracle picks out an abrasion tag (a_m) and indicates X hash functions, represented as the set, H_X. The tag, a_m, is the easiest to break of the g supplied abrasion tags and H_X defines the easiest hash functions to search to find preimages for that tag. Choosing a_m and H_X is dependent on the given the permutations of $\mathbb{Z}_N$, f_*. The oracle returns this information ($\{a_m, H_X\}$) to the adversary.

The adversary now searches through the X given hash functions (H_X) and finds preimages for the abrasion tag that the oracle revealed to them (a_m). The adversary

tries potential preimages in their pre-defined orderings, f_*, until they have recovered X correct preimages. Recovering X correct preimages completes the solution of the abrasion puzzle as the adversary can now use secret sharing to recover k.

Because this adversary is given the optimal hash functions to attack (H_X), their strategy is optimal for their order chosen. There is no better way to choose an order to search preimages because we assume they are truly random functions. We can now see that there's no practical adversary (without this oracle) that could retrieve a message faster than an adversary with this oracle. This means that bounding the success chance of this adversary with the oracle will bound any adversary without the oracle.

After the best H_X is revealed, the adversary cannot store work to gain success faster. They gain nothing because saved computations would only help reverse preimages in the same hash function. They only need to reverse a single preimage for each hash function and saved computations will not help them break the other preimages which are hashed with different hash functions. An example of this is where $x_i = x_{i+1}$. If the hash functions were not distinct, a malicious attacker would know that these two preimages were the same by comparing the abrasion hash of each. The attacker could then refer to their previous computations to reverse x_{i+1} after reversing x_i.

7.3 Success calculations

In this section we will show how to calculate the probability of success of a malicious adversary with this oracle, $\Pr[S_{\text{malicious}}]$, for a given amount of work $w_{\text{malicious}}$.

The work spent by this adversary is now the minimum value of the set of the

minimum work required for each of the submitted messages, $(min(w_i))$.

$$w_{\text{malicious}} = min(\{min(w_0), min(w_1), ..., min(w_{g-1})\})$$

The work to break each hash function in a single abrasion tag can be found by looking at the search order f_j submitted by the adversary for each hash function h_j. We define $x_{i,j}$ and $w_{i,j}$ as the preimage and the work required for the j-th distinct hash function of the i-th abrasion tag in g. We know the work required to recover one of these preimages will be the order of $x_{i,j}$ in the adversary's preimage search. This means work for each hash is now: $w_{i,j} = f_j(x_{i,j}), j \in \mathbb{Z}_Y$. Because the attacker only needs to reverse X preimages, the work to reverse a specific abrasion tag is now $min(w_i) = min_X(f_0(x_{i,0}), f_1(x_{i,1}), ..., f_{Y-1}(x_{i,Y-1}))$. Where $min_X(*)$ is the sum of the X smallest values in a given set. We can now express $w_{\text{malicious}}$ in terms of $x_{i,j}$ in Equation 7.1.

$$\begin{aligned}
w_{\text{malicious}} &= min(\{\forall i \in \mathbb{Z}_g, min(w_i)\}) \\
w_{\text{malicious}} &= min(\{min_X(f_*(x_{0,*})), min_X(f_*(x_{1,*})), ..., min_X(f_*(x_{g-1,*}))\}) \\
w_{\text{malicious}} &= min(\{min_X(f_0(x_{0,0}), f_1(x_{0,1}), ..., f_{Y-1}(x_{0,Y-1})), \\
&\qquad min_X(f_0(x_{1,0}), f_1(x_{1,1}), ..., f_{Y-1}(x_{1,Y-1})), \\
&\qquad ..., \\
&\qquad min_X(f_0(x_{g-1,0}), f_1(x_{g-1,1}), ..., f_{Y-1}(x_{g-1,Y-1}))\})
\end{aligned} \tag{7.1}$$

When each abrasion tag is generated, we pick the preimages $x_{i,j}$ from a scaled uniform distribution. This means that $f_j(x_{i,j})$ is also a random variable from a scaled uniform distribution.

$$w_{i,j} \sim N * Uniform(0, 1)$$

We sum the work for the lowest X preimages, which means that the work follows a sum of the lowest scaled uniform distributions.

$$min(w_i) = min_X(w_{i,*}) = \sum_{k=1}^{X} w_{i,(k)} \sim N * Uniform(0,1) \tag{7.2}$$

Where $w_{i,(k)}$ is an order statistic of the k-th lowest work required of the preimages in the i-th abrasion puzzle.

To calculate this, we sample this sum of order statistics many times and fit a normal curve to a histogram of the samples. The normal approximation is shown in Figures 7.1 and 7.2. The code used for the approximation is shown in Listing 7.1. We also calculate the error of this approximation, which is 5%. This error was calculated by summing the absolute error between the normal pdf and a histogram of the samples. We used 100 bins from the minimum to the maximum sample for the histogram. This approximation also take into account attacks described in Section 7.4.

Listing 7.1 shows an excerpt from the python script used to find the construction ratio. In this excerpt, we fit a normal distribution to the sum of minimum uniform distributions.

Listing 7.1: Fit normal to order statistics

```
min_x_samples = [sum(sorted(scipy.uniform.rvs(loc=0,scale=N,size=Y))[:
    ↪ X]) for _ in range(0,20000)]
mean,stdev = scipy.norm.fit(min_x_samples)
```

The probability that the work of our adversary is less than any given work, w, $(w_{\text{malicious}} < w)$ is equal to the probability that there exists an abrasion tag in g that

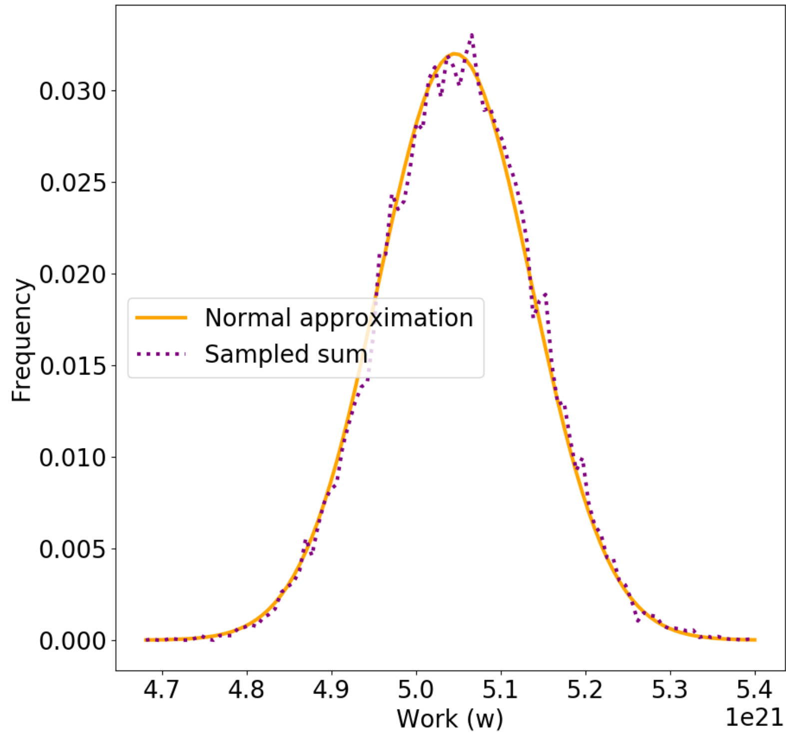

Figure 7.1: Uniform minimum sum pdf (approximated with 20,000 samples) vs normal approximation

is easier to break:

$$\Pr[w_{\text{malicious}} \leq w] = \Pr[\exists i \in \mathbb{Z}_g, min(w_i) \leq w]$$

The probability that such an i exists is the inverse of every $min(w_i)$ being greater than w:

$$\Pr[\exists i \in \mathbb{Z}_g, min(w_i) \leq w] = 1 - \Pr[\forall i \in \mathbb{Z}_g, min(w_i) > w]$$

The probability of every $min(w_i)$ being greater than w is the product of the

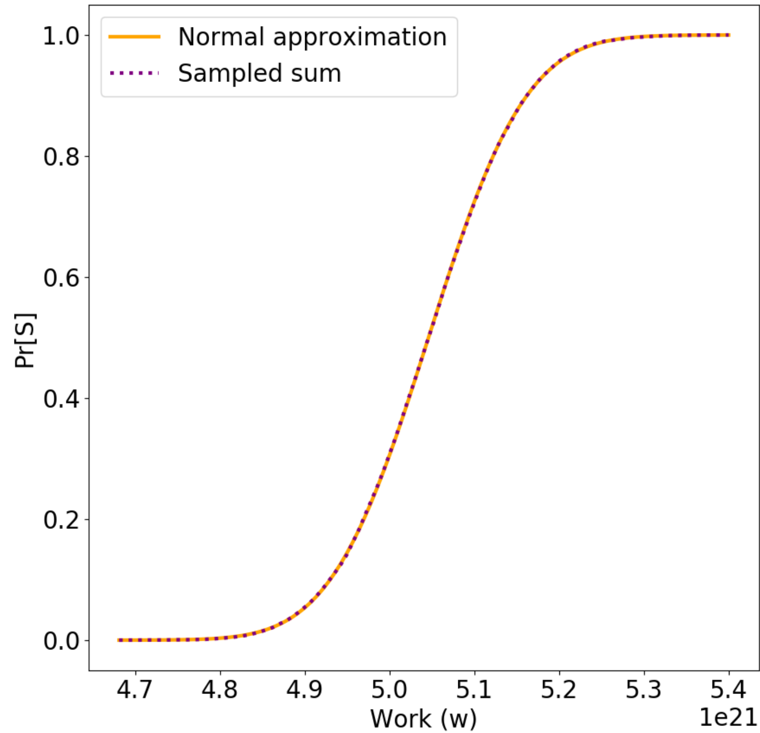

Figure 7.2: Uniform minimum sum cdf (approximated with 20,000 samples) vs normal approximation

probability that each message is greater.

$$\Pr[\forall i \in \mathbb{Z}_g, min(w_i) > w] = \Pr[min(w_i) > w]^g$$

We invert this so that we can use a cumulative density function to calculate it.

$$\Pr[\forall i \in \mathbb{Z}_g, min(w_i) > w] = (1 - \Pr[min(w_i) \leq w])^g$$

$$\Pr[w_{\text{malicious}} \leq w] = \Pr[\exists i \in \mathbb{Z}_g, min(w_i) \leq w] = 1 - \Pr[\forall i \in \mathbb{Z}_g, min(w_i) > w]$$

$$\Pr[w_{\text{malicious}} \leq w] = 1 - (1 - \Pr[min(w_i) \leq w])^g$$

Because we found that $min(w_i)$ can be approximated by a normal distribution, we can use the cumulative density function of the normal distribution to compute $\Pr[min(w_i) \leq w]$ and with that, easily compute $\Pr[w_{\text{malicious}} \leq w]$.

The probability of success of this adversary ($\Pr[S_{\text{malicious}}]$), at a given work ($w$), is the probability that the required work $w_{\text{malicious}}$ is less than that work.

$$\Pr[S_{\text{malicious}}] \text{ at work } w = \Pr[w_{\text{malicious}} \leq w]$$

We can now calculate the work when $\Pr[S_{\text{malicious}}] = \epsilon$. This value gives of a lower bound on the work of any malicious adversary as shown in Section 7.2.

In hash-based abrasion using TMTOs, we define our legitimate attacker as an attacker that precomputes a TMTO to decrypt abrasion tags. We call this a "table attacker." The calculations for the probability of success for the table attacker are derived in Section 6.3.10. We now know how to calculate the values needed to compute our security as defined in Section 5.2.2, $w_{\text{legitimate}}$ and $w_{\text{malicious}}$.

We calculate the work for our table attacker where our probability of success is $1 - \epsilon$. This allows us to derive a ratio, R.

$$\begin{aligned} R =& \frac{w_{\text{legitimate}}}{w_{\text{malicious}}} \\ R =& \frac{\text{Work at } \Pr[S_{\text{table attacker recovers } n}] = 1 - \epsilon}{\text{Work at } \Pr[S_{\text{tableless attacker recovers 1 given } g}] = \epsilon} \end{aligned} \tag{7.3}$$

We also factor in the cost of storage into this ratio in Section 8.2.

Different values of Y and X will yield different results. In Section 8 we give concrete values. These optimal parameters yield a ratio of $R \leq 13.2$ when $\lambda = 45$. This means our scheme is 45-13.2-secure by Definition 2. We believe this is a sufficient ratio to claim that this scheme accomplishes abrasion Property G and Property H.

7.4 Other attacks

A malicious attacker could also attack other parts of the scheme, such as secret sharing.

As described in Section 6.3.4, each share is encrypted with a strong key that results from a weak hash. The result of this encryption is a ciphertext sc_i and an authentication tag (t_i). This authentication tag was not shown in Algorithm 9 but was assumed to be generated along with each sc_i. Generally, the authentication tag is computed on the ciphertext of an encryption. This encryption mode is known as "Encrypt-then-MAC" [39]. Using "Encrypt-then-MAC" means that an attacker could recompute the exact same authentication tag on the ciphertext if they find the correct key.

To perform a brute-force search, a malicious attacker only needs a conditional to test if they've found the correct preimage. For each preimage, the adversary can use a brute-force attack on either the intended hash functions, ah_i, or the authentication tag for that share, t_i. The two conditionals necessary to perform these brute-force searches are shown in Equations 7.4 and 7.5. All of these variables except for x_i are given to the adversary, allowing them to brute-force search for x_i using either conditional.

$$H_s(pk + i||x_i) \stackrel{?}{=} ah_i \tag{7.4}$$

$$\text{MAC}(H_s(pk + 2 * Y + i||H_s(pk + Y + i||x_i)), sc_i) \stackrel{?}{=} t_i \tag{7.5}$$

This second conditional allows for a potentially easier path to recover preimages as the attacker's search order for t_i could yield x_i faster. This means that the easiest way to find a preimage is the smallest of a pair of uniform variables. These two

uniform variables are the order in which the adversary would recover the preimage through either the abrasion hash ah_i or the authentication tag t_i. Using this fact, we can show an updated calculation for $min(w_i)$ in g in Equation 7.6. We previously derived $min(w_i)$ in Equation 7.2. The function $f'(\cdot)$ is the ordering the adversary picks for the distinct hash functions used to create the ephemeral keys. We multiply this second value in the pair by 2 because two invocations of the hash function $H_s(\cdot)$ are used to create the ephemeral key during the brute-force attacks which doubles the work required. We can approximate this with a normal distribution as shown in Figures 7.1 and 7.2.

$$
\begin{aligned}
min(w_i) = min_X(&min(f_0(x_{i,0}), 2f'_0(x_{i,0})),\\
&min(f_0(x_{i,1}), 2f'_1(x_{i,1})),\\
&...,\\
&min(f_0(x_{i,Y-1}), 2f'_1(x_{i,Y-1})))\\
min(w_i) = min_X(&\forall j \in \mathbb{Z}_Y, min(f_j(x_{i,j}), 2f'_j(x_{i,j}))) \qquad (7.6)
\end{aligned}
$$

Even if we removed this authentication tag from our scheme, secret sharing could also give a malicious attacker other ways to brute-force preimages. If the attacker has recovered $X-1$ correct shares ($|s^*| = X-1$), they can brute-force search the last share they need regardless of the secret sharing scheme used. The conditional needed for this brute-force search is shown in Equation 7.7. In this brute-force search, attacker can choose a plaintext (p) and retrieve the ciphertext and abrasion tag c, a. This p could be a message sent over the messaging platform that is performing the abraded encryption. This c is different than the c described in our construction in Section 6.3 and is instead the result of the encryption of p with key k. Allowing

an attacker this ability is a reasonable attack scenario known as a chosen plaintext attack. After the attacker finds an x_i that satisfies this condition, they can recover the last share and learn k.

$$
\begin{aligned}
e' &= H_s(pk + 2 * Y + i || H_s(pk + Y + i || x_i)) \\
s' &= \text{Dec}(e', sc) \\
k' &= \text{SSDecrypt}(s^* + \{s'\}) \\
\text{Dec}(k', c) &\stackrel{?}{=} p
\end{aligned} \tag{7.7}
$$

Bounding all attacks using the secret sharing scheme is difficult and relies on properties of the secret sharing scheme. To make our abrasion construction independent of which secret sharing scheme is being used, we assume that an attacker can always brute-force search for ephemeral keys to break the scheme.

To ensure this attack does not affect our security significantly, we require computation of the hash function H_s twice when deriving keys from each nonce, x_i. This double hash makes it much more costly to retrieve a secret share using the authentication tag compared to reversing an abrasion hash (ah_*) as intended. The table attacker only has to compute this double invocation of H_s this when verifying a preimage, meaning it doesn't affect their online cost significantly.

Chapter 8

Results

In this chapter, we use real-world parameters to calculate the security and cost of our scheme. First, we will show the parmeters used, the security ratio (R), and important costs in Section 8.1. Then we show calculations of other costs associated with a scheme intialized with these parameters in Section 8.2. We show costs for other sets of parameters in Table 8.1.

8.1 Summary

For reference, we provided a number of definitions of important variables in Table 6.1 in Section 6.3.2.

Established in Section 5.2, we want to calculate the ratio of the work done by our legitimate attacker vs the work done by a malicious attacker $\left(R = \frac{w_{\text{legitimate}}}{w_{\text{malicious}}}\right)$.

Using calculations in Chapter 7, we can now calculate bounds on $w_{\text{malicious}}$. Shown in Section 6.3.10, we can calculate the cost of our table attacker ($w_{\text{legitimate}}$). In Section 6.3.7 we show how to calculate other associated costs with the construction.

We set concrete parameters: $Y = 6114$, $X = 3035$, $g = 2^{45}$ (trillions of records), $\epsilon = 2^{-45}$, $n = 2^{45}$, $N = 2^{63}$, and define $H_s(\cdot)$ as 500 computations of DoubleSHA256. With these parameters, we find that our table attacker spends \$40.2 million computing the hash function and our security ratio (R) is ≤ 13.2. This means that if

our legitimate adversary spends \$40.2 million on a private key that can decrypt 2^{45} records, an adversary that decrypts a single message will spend \$3 million.

The required work for a table attacker is low enough to allow law enforcement to investigate. At the same time, the cost of a tableless attacker (\$3 million) is high enough to deter smaller attackers from decrypting records. The resources of these attackers were estimated in Section 5.1.4. Our main construction is the first abrasion scheme that provably provides these properties. Allowing decryption by a table attacker while preventing attacks by malicious attackers means that our construction achieves Property G and Property H.

These parameters were found through experimentation. We follow the "matrix stopping rule" described in Section 6.2 and fix $Y, N, H_s(\cdot)$ to give us a \$40.2 million cost, then found the X that gives our table attacker $1-\epsilon$ success chance. The security ratio, R, was then computed using these parameters.

Larger values of Y and X give better ratios, but also impose a larger cost for computing abrasion tags and increase storage requirements. The best ratio of $\frac{X}{Y}$ stayed around 50% regardless of the values of Y or N. Using our approximations, we were able to compute this ratio up to $Q = 2^{80}$ which is an abraded encryption that could reasonably cost billions of dollars to break. At any reasonable value of I (from \$1 million to \$100 million), our scheme stays secure with acceptable security ratios.

We only compare invocations of the hash function when calculating the security ratio. Factoring in storage requires us to factor in hash efficiency and power costs which makes analyzing schemes much more complicated.

A comparison of table and tableless success chances is shown in Figure 8.1, which starts at 0 work and continues to the work required where $\Pr[S_{\text{legitimate}}] = 1 - \epsilon$. The "s-curve" parts of these two attackers' probabilities are shown in Figures 8.2 and 8.3.

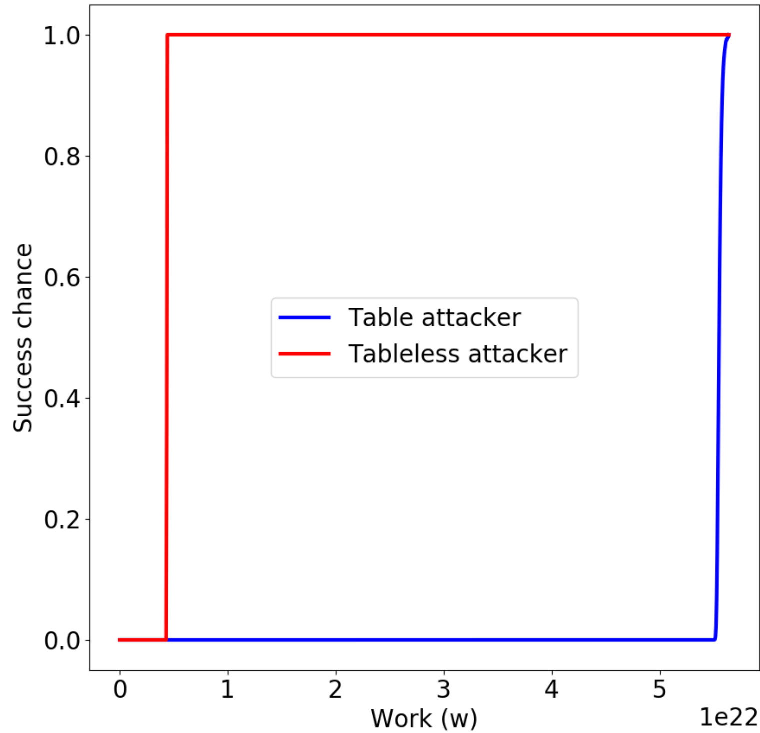

Figure 8.1: Tableless vs table attacker, success chance vs work

8.2 Detailed cost analysis

Calculations showed that the table adversary would need to perform $Q * R = N * Y \approx 5.64 * 10^{22}$ computations of the hash function H_s.

We used calculations from the original paper on crumpled and abraded security [58], to estimate the cost of electricity that a table attacker will use to derive a private key for this abrasion puzzle. The calculation of our legitimate attacker's cost to create a private key is shown in Equation 8.1.

$$\frac{5.64 * 10^{22} * 500\ Hashes}{1.02 * 10^{10}\ Hashes/Joule} * \frac{1\ kWh}{3,600,000\ Joules} * .0523\frac{\$}{kWh} \approx \text{ \$40.2 million} \quad (8.1)$$

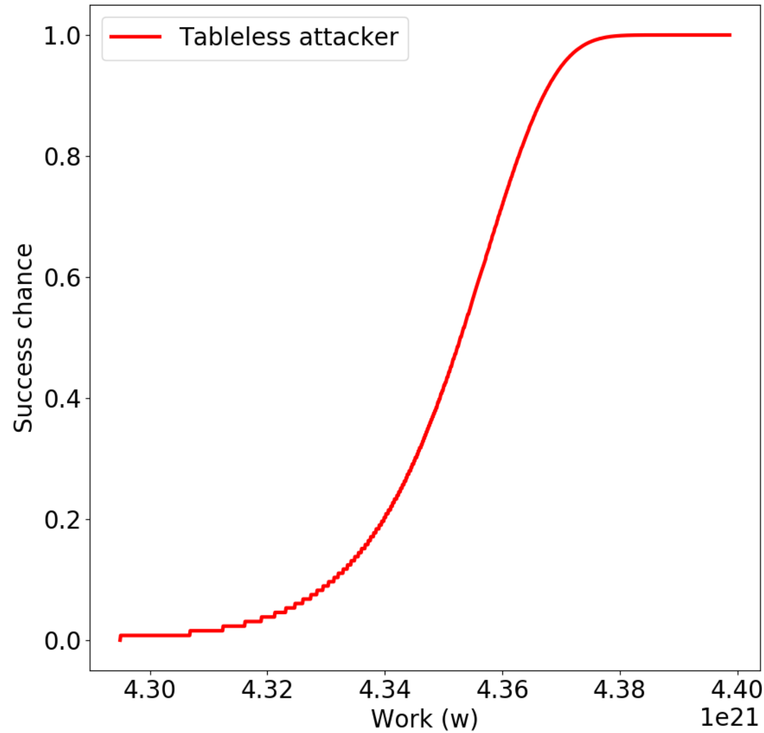

Figure 8.2: Tableless attacker ϵ to $1-\epsilon$ success chance

This assumes that the government purchases very powerful hashing equipment, the Antminer S9, which has an efficiency of $1.02 * 10^{10} Hashes/Joule$ [17, 58]. The cost of purchasing this hardware is not calculated. When calculating the ratio, we assume a tableless attacker purchases the same hardware as the table attacker.

We find $M * Y$ and multiply it by the storage requirement for each row to get the total storage cost, shown in Equation 8.2.

$$(m * l) * Y * (\text{start point bytes} + \text{end point bytes}) = \text{bytes of storage}$$
$$\left(\frac{5.64 * 10^{22}}{6144}\right)^{\frac{2}{3}} * 6144 * \left(\left\lceil\frac{63}{8}\right\rceil + \left\lceil\frac{log_2(m)}{8}\right\rceil\right) \approx \text{ 296 PB} \tag{8.2}$$

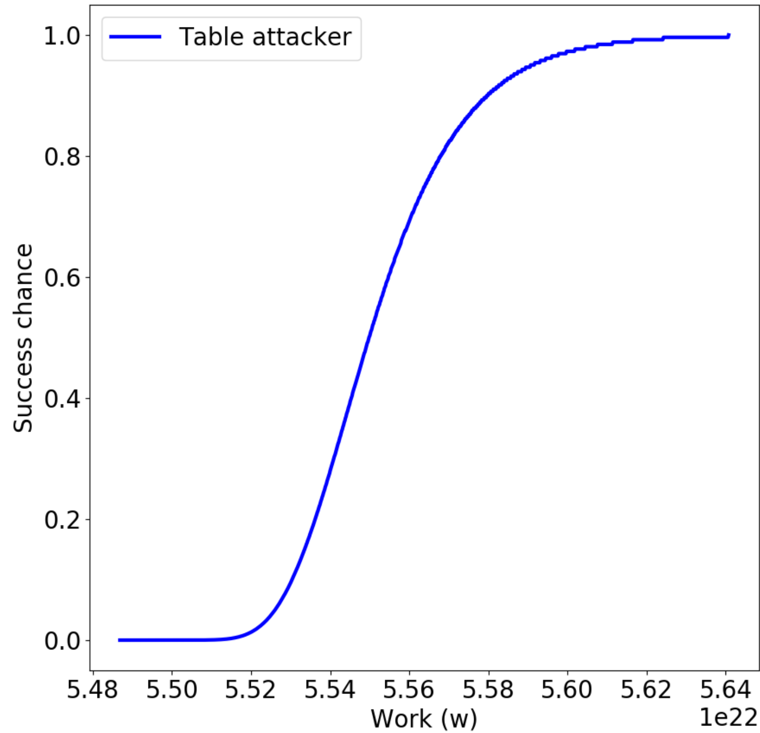

Figure 8.3: Table attacker ϵ to $1-\epsilon$ success chance

The cost to store these TMTOs would vary. Using projected hard disk drive costs for 2019 [6], we calculate the cost of storing a private key in Equation 8.3.

$$296 \text{ PB} * 1,000,000\frac{GB}{PB} * 0.03\frac{\$}{GB} = \$8.8 \text{ million} \tag{8.3}$$

This storage cost does not overwhelm our electrical cost. This pool of storage can be split up into $Y = 6114$ different searchable arrays, each containing 47 TB. Splitting the data up this way should substantially reduce the overhead cost to store this data, which is left out of our calculations.

Now we can derive the total precomputation cost of our table attacker, described

in Equation 8.4.

$$\text{electrical cost} + \text{storage cost} = \text{total cost}$$
$$\$40.2 \text{ million} + \$8.8 \text{ million} = \$49 \text{ million} \tag{8.4}$$

Including the storage cost gives us a cost ratio of 16. We keep this total cost ratio distinct from our security ratio (R) as the cost ratio will fluctuate based on hardware efficiency and electricity costs. The security ratio, R, is based on invocations of the hash function and will never change due to changes in hardware/electricity cost.

The online cost for each decryption with a table will be the equivalent of searching through $T * 6114 = t * l * 1.5 * 6114$ hashes, coming to around $N^{\frac{2}{3}} * 1.5 * 6114 = 4 * 10^{16}$ operations. Using the cost analysis from the original paper [58], we calculate the online cost of the table attacker in Equation 8.5.

$$\frac{4 * 10^{16} * 500\ Hashes}{1.02 * 10^{10}\ Hashes/Joule} * \frac{1\ kWh}{3,600,000\ Joules} * .0523\frac{\$}{kWh} \approx\ \$29 \tag{8.5}$$

This cost is smaller than the crumpling cost suggested [58], which was ≈ \$1000. If we combine this scheme with a crumpling scheme, we can adjust the cost of retrieving a crumple-encrypted message to offset the \$29 imposed per-message by this abrasion scheme. This low per-message cost gives our scheme Property F.

The client needs to compute Y hashes of 500 length 3 times. We calculate the cost of computing an abrasion tag in Equation 8.6.

$$\frac{6114 * 3 * 500\ Hashes}{5 * 10^{5}\ Hashes/Joule} * \frac{1\ kWh}{3,600,000\ Joules} * .0523\frac{\$}{kWh} \approx\ \$0.0000003 \tag{8.6}$$

The client must compute H_s for each distinct hash function to compute the abra-

sion hashes ah_i. This multiplication by 3 comes from the two extra H_s invocations needed to generate the keys to encrypt the secret shares. The reasoning for this double hashing is described in Section 7.4.

We use a weaker $Hashes/Joule$ ratio, associated with CPUs, to calculate client cost as clients will not likely use powerful hashing hardware like Antminers. A value of $5 * 10^5$ $Hashes/Joule$ was used in [58].

The frequency of abrasion tag computations can be adjusted as described in Section 6.3.2 to offset this cost and the storage size of abrasion tags. This storage size is computed in Equation 8.7.

$$Y * \left(\left\lceil log_2(N) * \frac{1\ byte}{8\ bits} \right\rceil + 32 \right)$$
$$6114 * (\lceil 63/8 \rceil + 32) = 244.5 \text{ KB} \tag{8.7}$$

The extra 32 bytes accounts for the secret share ciphertexts. Each of these takes 32 bytes including a MAC.

There are many trade-offs to consider in these calculations. Depending on how important some factors are, these parameters could be adjusted to achieve different costs. One major trade-off is the number of hashes that $H_s(\cdot)$ computes. Increasing this will decrease the storage required by the table attacker, but also increase the cost of computing an abrasion tag.

These parameters must be fixed for all clients. Changing Y, X, N or $H(\cdot)$ will force the table attacker to regenerate some or all of their table to maintain the same chance of success.

A legitimate attacker could also adjust l, m, and t to change the storage/online costs of their TMTO solution. They could also use another TMTO such as rainbow tables or distinguished points. We do not explore this in this work.

If this strategy were adopted by a legal system, law enforcement could build this TMTO strategy on-the-fly, meaning they spend somewhere between \$3 million and \$49 million to decrypt their first message while still building the TMTO tables. Their total cost will increase as they decrypt more and more abrasion tags, getting to \$49 million after they decrypt trillions of records.

In Table 8.1, we calculate the ratios and costs of schemes with other parameters. All rows use a λ of 2^{-45}. The parameter u defines the number of invocations of DoubleSHA256 used by H_s. The cost for the table attacker includes storage in this table. The first row displays the values found previously in this section. Row 2 shows what happens when all parameters are scaled up. Rows 3 and 4 show what happens when we redefine H_s. Rows 5 through 7 show what happens when we change the input space for preimages. Rows 8 through 11 show what happens when we modify both Y and X. In rows 12 and 13, we've reduced the work spent on each hash function ($w_{\text{legitimate}} = \frac{N*Y}{2}$) and the ideal X was reduced to give this table attacker $1 - \epsilon$ chance of success. Deviations from the first row are highlighted for set parameters (R is derived from other parameters).

8.3 Revisiting abrasion requirements

In this section, we list the properties that our main construction, from Section 6.3, achieves. We reference the locations in the thesis where we proved these properties.

Many of our claims rely on the fact that reversing the hash functions is the easiest way to break the scheme. This is proven in Section 7.4.

During the description in Section 6.3.2 we described how our construction uses a public key to satisfy Property C.

Because the weakest part of our encryption uses hash functions, we can easily

Table 8.1: Costs with other parameters

#	Parameters N	Y	X	u	R	Costs (millions of dollars) Table	Malicious	Storage	Costs (dollars) Online
1	2^{63}	6114	3035	500	13.2	\$49.0	\$3	\$8.8	\$29
2	2^{64}	7000	3499	700	12.8	\$144.9	\$10.1	\$16.1	\$73.1
3	2^{63}	6114	3035	100	13.2	\$16.9	\$0.6	\$8.8	\$5.7
4	2^{63}	6114	3035	1000	13.2	\$80.3	\$6	\$8.8	\$57
5	2^{64}	6114	3035	500	13.2	\$80.3	\$6	\$14	\$45
6	2^{60}	6114	3035	500	13.2	\$7.2	\$0.4	\$2.2	\$7.2
7	2^{67}	6114	3035	500	13.2	\$704.0	\$49.5	\$61.5	\$182.4
8	2^{63}	8000	4024	500	12.5	\$52.5	\$4.2	\$11.6	\$37.5
9	2^{63}	10000	5079	500	12.1	\$80.2	\$5.4	\$14.5	\$47.0
10	2^{63}	4000	1936	500	14.5	\$32.1	\$1.8	\$5.8	\$18.8
11	2^{63}	1000	419	500	25.9	\$8.0	\$0.3	\$1.5	\$4.7
12	2^{63}	6114	1885	500	18.8	\$25.7	\$1.1	\$5.6	\$18.1
13	2^{63}	10000	3035	500	17.1	\$39.6	\$1.8	\$8.8	\$28.4

calculate the costs of how to break it. This is similar to how the costs for crumpling were originally calculated [58]. This gives our construction Property E.

When treating hash functions as a random oracle model, there is no way to reverse a hash without iterating through its input space [26]. This allows any abrasion construction that is based on hash functions to achieve Property I as there's no way to generate a public key that makes the hashes easier to reverse. Random oracles are described in Section 7.2.

From our first simple hash-based abrasion scheme in Section 6.1 it should be clear how an adversary can reuse a TMTO as a "private key," satisfying Property D.

We described a security model in Section 5.2, created a security game and oracle for our scheme in Chapter 7, and calculated real world numbers in Chapter 8. Our main construction is the first abrasion scheme to provably guarantee security. We describe the resources of various attackers in Section 5.1.4 and thus can claim Property G and Property H. Also in Chapter 8, we show that our private key is very large, which

could help prevent its theft. This could be beneficial and meets Property K.

In Section 8.2, we calculated other costs of our scheme, proving that it is practical. We also calculated the per-message cost of our abrasion construction. This cost is much smaller than the crumpling cost suggested by Wright and Varia [58]. This means that, when our scheme is combined with a crumpling scheme, it achieves Property F.

As for ease of implementation, we integrated crumpled logging into Postfix in Section 4. This implementation is quite simple, and an abraded encryption implementation may not be much more complicated. Our construction also does not require any secret keys to be escrowed. These two qualities show that our construction would be simple to implement, as required by Property J.

Chapter 9

Conclusion

In this thesis we were able to achieve a practical abraded encryption scheme and evaluate the cost and security of it. We developed a framework to evaluate the security of any abrasion scheme. The main construction presented in this thesis is the first abrasion construction to be proven to guarantee any level of security. While some of the costs for storage and computation of our construction may seem undesirable, they are not overwhelming. We also created a proof of concept for crumpling.

9.1 Future work

We believe that probabilistic sharing (described in Section 6.3.9) can be used to reduce the cryptographic dependencies of this construction, thus making it easier to implement and audit. Future work will evaluate the security of our construction when used with probabilistic sharing.

Another improvement to the abrasion construction would be to modify the abrasion function described in Section 6.3.4 to mimic the creation of a TMTO. Introducing similarities between tag creation and table creation ensures that tables are more effective in reversing tags when compared to brute-force attacks. Specifically, we believe that if each abrasion hash were computed in a chain in the same way that TMTO chains were computed, our table attacker would have a significant advantage. This

improvement should reduce our ratio dramatically, but requires a new security oracle and game to fully analyze. Also, more research into TMTOs would be required to derive the legitimate attacker's success chance.

This abrasion construction could have applications in other fields. The novel properties of one-time work for multiple decryption could possibly be used in a proof-of-work blockchain or similar technology.

9.2 Crumpling and abrasion

The monetary cost of abrasion and crumpling is not negligible even with ideal constructions as we must force attackers to spend work to deter small attackers and rate limit law enforcement. Combining this solution with other exceptional access solutions such as AUDIT [36] or self-escrow [54] could help reduce this cost. The cost of doing nothing is arguably greater than any of these exceptional access solutions as uncontrolled weakening of security by law enforcement could have far more costly effects on society.

One potential problem with abrasion is the possible creation of criminal tag reversal as a service. A larger criminal could spend the initial cost to break the abrasion puzzle, then sell individual decryptions to smaller criminals to recoup their loss. While this would be a problem, abrasion would naturally centralize criminals that had the ability to do these decryptions. A centralized criminal organization is easier to attack than many smaller operations.

9.3 Safety and privacy

Our criminal investigations are progressively requiring more interaction with technology and the internet. With internet privacy being so important today, compromise

between privacy and safety is hard to develop and solutions to this problem can be very polarizing, politically. Attempting to introduce a compromise is often met with fierce opposition, which hampers progress towards a solution. Part of the motivation for this construction is to generate conversation about this compromise. We hope that this construction can be used to create or inspire the creation of more, practical, compromises between privacy and security in the future.

References

[1] "Cryptolocker victims to get files back for free," British Broadcasting Company, Aug. 2014. [Online]. Available: https://www.bbc.com/news/technology-28661463

[2] "Deep dive into crypto 'exceptional access' mandates: Effective or constitutional-pick one," Electronic Frontier Foundation, Aug. 2015. [Online]. Available: https://www.eff.org/deeplinks/2015/08/deep-dive-crypto-exceptional-access-mandates-effective-or-constitutional-pick-one

[3] "Fbi director says agents need access to encrypted data to preserve public safety," National Public Radio, Jul. 2015. [Online]. Available: https://www.npr.org/sections/thetwo-way/2015/07/08/421251662/fbi-director-says-agents-need-access-to-encrypted-data-to-preserve-public-safety

[4] "Carpenter v. united states," U.S. Supreme Court, Jun 2018. [Online]. Available: https://www.supremecourt.gov/opinions/17pdf/16-402_h315.pdf

[5] "Police use fitbit data to charge 90-year-old man in stepdaughter's killing," New York Times, Oct. 2018. [Online]. Available: https://www.nytimes.com/2018/10/03/us/fitbit-murder-arrest.html

[6] "Digital storage projections for 2019, part 1," Forbes, Apr. 2019. [Online]. Available: https://www.forbes.com/sites/tomcoughlin/2018/12/21/digital-storage-projections-for-2019-part-1/#2c615a534428

[7] "Encrypting stored email with postfix : Kacang bawang," Apr. 2019. [Online]. Available: http://kacangbawang.com/encrypting-stored-email-with-postfix/

[8] "Expense management, travel, invoice software, travel expense reporting - sap concur," Apr. 2019. [Online]. Available: https://www.concur.com/

[9] "Forget about backdoors, this is the data whatsapp actually hands to cops," Forbes, Apr. 2019. [Online]. Available: https://www.forbes.com/sites/thomasbrewster/2017/01/22/whatsapp-facebook-backdoor-government-data-request/

[10] "getrandom(2) - linux manual page," Apr. 2019. [Online]. Available: http://man7.org/linux/man-pages/man2/getrandom.2.html

[11] "Github - dsprenkels/sss: Library for the shamir secret sharing scheme," Apr. 2019. [Online]. Available: https://github.com/dsprenkels/sss

[12] "Github - jcushman/libgfshare," Apr. 2019. [Online]. Available: https://github.com/jcushman/libgfshare

[13] "Github - openssl/openssl: Tls/ssl and crypto library," Apr. 2019. [Online]. Available: https://github.com/openssl/openssl

[14] "How the assistance and access bill is going to hurt every australian." Hackernoon, Apr. 2019. [Online]. Available: https://hackernoon.com/how-the-assistance-and-access-bill-is-going-to-hurt-every-australian-7a5c935c797f

[15] "Hundreds of millions of facebook user records were exposed on amazon cloud server," CBS, Apr. 2019. [Online]. Available: https://www.cbsnews.com/news/millions-facebook-user-records-exposed-amazon-cloud-server/

[16] "Mail (mx) server survey," Apr. 2019. [Online]. Available: http://www.securityspace.com/s_survey/data/man.201809/mxsurvey.html

[17] "Mining hardware comparison." Mar. 2019. [Online]. Available: https://en.bitcoin.it/wiki/Mining_hardware_comparison

[18] "The postfix home page," Apr. 2019. [Online]. Available: http://www.postfix.org/

[19] "Safecurves: Introduction," Apr. 2019. [Online]. Available: http://safecurves.cr.yp.to/

[20] "scipy.integrate.quad - scipy v1.2.1 reference guide," Apr. 2019. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html

[21] A. Abdallah and M. Salleh, "Secret sharing scheme security and performance analysis," in 2015 International Conference on Computing, Control, Networking, Electronics and Embedded Systems Engineering (ICCNEEE), Sep. 2015, pp. 173–180.

[22] H. Abelson, R. Anderson, S. M. Bellovin, J. Benalob, M. Blaze, W. Diffie, J. Gilmore, P. G. Neumann, R. L. Rivest, J. I. Schiller, and B. Schneier, "The risks of key recovery, key escrow, and trusted third-party encryption," World Wide Web J., vol. 2, no. 3, pp. 241–257, Jun. 1997. [Online]. Available: http://dl.acm.org/citation.cfm?id=275079.275104

[23] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in 22nd ACM Conference on Computer and Communications Security, Oct. 2015.

[24] G. Avoine, P. Junod, and P. Oechslin, "Time-memory trade-offs: False alarm detection using checkpoints." 01 2005, pp. 183–196.

[25] E. Barkan, E. Biham, and A. Shamir, "Rigorous bounds on cryptanalytic time/memory tradeoffs," in Advances in Cryptology - CRYPTO 2006, C. Dwork, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–21.

[26] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in Proceedings of the 1st ACM Conference on Computer and Communications Security, ser. CCS '93. New York, NY, USA: ACM, 1993, pp. 62–73. [Online]. Available: http://doi.acm.org/10.1145/168588.168596

[27] S. M. Bellovin, M. Blaze, S. Clark, and S. Landau, "Going bright: Wiretapping without weakening communications infrastructure," IEEE Security Privacy, vol. 11, no. 1, pp. 62–72, Jan 2013.

[28] G. Blakley, "Safeguarding cryptographic keys," Managing Requirements Knowledge, International Workshop on, vol. 0, p. 313, 01 1979.

[29] J. Borst, B. Preneel, and J. Vandewalle, "On the time-memory tradeoff between exhaustive key search and table precomputation," in Symposium on Information

Theory in the Benelux. TECHNISCHE UNIVERSITEIT DELFT, 1998, pp. 111–118.

[30] V. Caproni, "Statement before the house judiciary committee, subcommittee on crime, terrorism, and homeland security," Feb. 2011.

[31] J. B. Comey, "Fbi budget request for fiscal year 2017," Jun. 2017. [Online]. Available: https://www.fbi.gov/news/testimony/fbi-budget-request-for-fiscal-year-2017

[32] T. Cook, "A message to our customers," Apr. 2019. [Online]. Available: https://www.apple.com/customer-letter/

[33] D. E. Denning, Cryptography and Data Security. Addison-Wesley, 1982.

[34] D. E. Denning and D. K. Branstad, "A taxonomy for key escrow encryption systems," Commun. ACM, vol. 39, no. 3, pp. 34–40, Mar. 1996. [Online]. Available: http://doi.acm.org/10.1145/227234.227239

[35] W. Diffie and M. Hellman, "New directions in cryptography," IEEE Trans. Inf. Theor., vol. 22, no. 6, pp. 644–654, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1109/TIT.1976.1055638

[36] J. Frankle, S. Park, D. Shaar, S. Goldwasser, and D. J. Weitzner, "Audit: Practical accountability of secret processes," Cryptology ePrint Archive, Report 2018/697, 2018, https://eprint.iacr.org/2018/697.

[37] A. Gabizon, "On the security of the bctv pinocchio zk-snark variant," Cryptology ePrint Archive, Report 2019/119, 2019, https://eprint.iacr.org/2019/119.

[38] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, ser. STOC '87. New York, NY, USA: ACM, 1987, pp. 218–229. [Online]. Available: http://doi.acm.org/10.1145/28395.28420

[39] P. Gutmann, "Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," RFC 7366, Sep. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7366.txt

[40] M. Hellman, "A cryptanalytic time-memory trade-off," IEEE Trans. Inf. Theor., vol. 26, no. 4, pp. 401–406, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1109/TIT.1980.1056220

[41] T. J. Holt, O. Smirnova, and Y. T. Chua, "Exploring and estimating the revenues and profits of participants in stolen data markets," Deviant Behavior, vol. 37, no. 4, pp. 353–367, 2016. [Online]. Available: https://doi.org/10.1080/01639625.2015.1026766

[42] J. Hong and S. Moon, "A comparison of cryptanalytic tradeoff algorithms," J. Cryptology, pp. 559–637, 2013.

[43] H. Krawczyk, "Secret sharing made short," in Advances in Cryptology — CRYPTO' 93, D. R. Stinson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 136–146.

[44] K. Kusuda and T. Matsumoto, "Optimization of time-memory trade-off cryptanalysis and its application to des, feal-32, and skipjack," pp. 35–47, 01 1996.

[45] A. McCabe, “Fbi budget request for fiscal year 2018,” Feb. 2016. [Online]. Available: https://www.fbi.gov/news/testimony/fbi-budget-request-for-fiscal-year-2018

[46] M. McGuire, “Into the web of profit,” Bromium, Apr. 2018. [Online]. Available: https://www.bromium.com/wp-content/uploads/2018/05/Into-the-Web-of-Profit_Bromium.pdf

[47] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf,” 2009.

[48] National Institute of Standards and Technology, FIPS PUB 46-3: Data Encryption Standard (DES), Oct. 1999, supersedes FIPS 46-2. [Online]. Available: http://www.itl.nist.gov/fipspubs/fip186-2.pdf

[49] NIST, “Escrowed encryption standard,” Feb. 1994.

[50] P. Oechslin, “Making a faster cryptanalytic time-memory trade-off,” in Advances in Cryptology - CRYPTO 2003, D. Boneh, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 617–630.

[51] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” Commun. ACM, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[52] R. L. Rivest, A. Shamir, and D. A. Wagner, “Time-lock puzzles and timed-release crypto,” Cambridge, MA, USA, Tech. Rep., 1996.

[53] R. J. Rosenstein, “Deputy attorney general rod j. rosenstein delivers remarks on encryption at the united states naval academy,” Oct. 2017. [Online].

Available: https://www.justice.gov/opa/speech/deputy-attorney-general-rod-j-rosenstein-delivers-remarks-encryption-united-states-naval

[54] S. Savage, "Lawful device access without mass surveillance risk: A technical design discussion," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1761–1774. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243758

[55] B. Schneier, "Did nsa put a secret backdoor in new encryption standard?" Oct. 2007. [Online]. Available: https://www.wired.com/2007/11/securitymatters-1115/

[56] A. Shamir, "How to share a secret," Commun. ACM, vol. 22, no. 11, pp. 612–613, Nov. 1979. [Online]. Available: http://doi.acm.org/10.1145/359168.359176

[57] Trustwave, "The value of data report," 2017. [Online]. Available: https://www2.trustwave.com/Value-of-Data-Report_LP.html

[58] C. V. Wright and M. Varia, "Crypto crumple zones: Enabling limited access without mass surveillance," 3rd IEEE European Symposium on Security and Privacy, 2018.

[59] ——, "A cryptographic airbag for metadata: Protecting business records against unlimited search and seizure," Proc. 8th USENIX Workshop on Free and Open Communications on the Internet, 2018.

[60] A. C. Yao, "Protocols for secure computations," in Proceedings of the 23rd Annual Symposium on Foundations of Computer Science, ser. SFCS '82.

Washington, DC, USA: IEEE Computer Society, 1982, pp. 160–164. [Online].
Available: http://dl.acm.org/citation.cfm?id=1382436.1382751

Appendix A

Hellman table success derivation

The success chance of a TMTO is the chance that any given output of the hash function is reversible by the tables. These tables are computed based on certain parameters, which affect the success chance. We review TMTOs and these parameters in Section 3.1.

In the original TMTO paper in 1980, Hellman finds a lower bound for the success chance of one of his TMTO tables through some simple reasoning and algebra [40]. We review Hellman's derivation here and extend it to find Equation A.2. This extension is not overly complex and has been used by many TMTO authors [42]. In our research, we did not find a long form derivation of this equation. This appendix provides that missing long form derivation. Kusuda et al. found through experimentation that this equation produces a lower bound on the success chance of TMTOs [44].

We can think of the success chance of a TMTO table as a function of the number of preimages that are "covered" by the table. If the preimage for a given hash value was never computed by the table creator during precomputation, there's no way to use the table to reverse that hash value.

Let's label this number of preimages covered by a table as the set $A_{i,j}$ where i is the index of the row and j is the index in the chain of that row. This is the set of every distinct preimage that has been computed, resulting in a corresponding hash. The

i and j parameters determine the row and index in the chain that the table creator has reached. After precomputation of a table is complete, their covered preimages will be $A_{m,t}$.

Our probability of success for our table $\Pr[S_{\text{table}}]$ is the probability that any given hash is in the table and can be shown as:

$$\Pr[S_{\text{table}}] = \frac{1}{N} * E[|A_{m,t}|]$$

Where $|A_{i,j}|$ is the number of elements in $A_{i,j}$ and N is the input space of the cryptographic function.

Our preimages are generated in a number of rows spanning a number of columns. This can be modeled mathematically like so:

$$\Pr[S_{\text{table}}] = \frac{1}{N} * E\left[\sum_{i=1}^{m}\sum_{j=0}^{t-1}\frac{\Pr[X_{i,j} \text{ is new}]}{N}\right]$$

By being "new" we mean the preimage in row i, column j ($X_{i,j}$) has not appeared in the table previously and thus increases the size of $A_{i,j}$.

We know that the probability $\Pr[X_{i,j} \text{ is new}]$ is at least equal to or higher than the probability that $X_{i,j}$ is new *and* every $X_{i,k}$ where $k < j$ is new.

$$\begin{aligned}\Pr[X_{i,j} \text{ is new}] \geq & \Pr[X_{i,0} \text{ is new}] \\ & * \Pr[X_{i,1} \text{ is new} | X_{i,0} \text{ is new}] \\ & * ... \\ & * \Pr[X_{i,j} \text{ is new} | X_{i,0} \text{ is new}, X_{i,1} \text{ is new}, ..., X_{i,j-1} \text{ is new}]\end{aligned}$$

This takes the chance of chain merges into account. We describe chain merges in Section 3.1. $X_{i,j}$ would be a duplicate if any $X_{i,k}, k < j$ were a duplicate. Multiplying our probability by the chance that each previous index in the chain was new represents this chance [44].

Substituting $A_{i,j}$ recursively, we can see that this equation is equivalent to following equation because $A_{i,0}$ is the number of new preimages at the start of row i.

$$
\begin{aligned}
\Pr[X_{i,j} \text{ is new}] \geq & \frac{N - |A_{i,0}|}{N} \\
& * \frac{N - |A_{i,0}| - 1}{N} \\
& * \ldots \\
& * \frac{N - |A_{i,0}| - j}{N}
\end{aligned}
$$

If we assume that every single preimage was new, we can substitute in $i * t$ for each of the $|A_{i,0}| - k, k \geq 0, k \leq j$, leaving us with:

$$\Pr[X_{i,j} \text{ is new}] \geq \left(\frac{N - i * t}{N}\right)^{j+1}$$

This allowed Hellman to derive the final equation.

$$\Pr[S_{\text{table}}] \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=1}^{t} \left(\frac{N - i * t}{N}\right)^{j} \tag{A.1}$$

Note here that this lower bound in Equation A.1 could've been tighter if $(i-1)*t-k$

were substituted in for each $|A_{i,0}| - k, k \geq 0, k \leq j$ instead of simply $i * t$ for each.

$$\Pr[S_{\text{table}}] \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=1}^{t} \prod_{k=0}^{j} \frac{N - (i-1) * t - k}{N}$$

Using Equation A.1 allows us to make some approximations that might not have been possible with a closer bound.

When using l tables, our hash chance of success for reversing the hash $\Pr[S_{\text{hash}}]$ becomes:

$$\Pr[S_{\text{hash}}] \geq 1 - (1 - \Pr[S_{\text{table}}])^{l}$$

We don't have to worry about merging chains across tables because they use different reduction functions. Each table's chance of success is independent if different starting points are used for each table. We will not run out of distinct starting points as long as $m * l \leq N$.

When m and t are large, this equation becomes very difficult to compute. Even using heavy parallelization on available GPUs, this equation would take months to compute for $N = 2^{80}$.

We approximate this chance of success using integrals, which uses far less CPU time.

In this approximation, a linear approximation to e^{x} is used.

$$e^{-x} \approx 1 - x$$

This approximation allows us to estimate the lower bound as:

$$\Pr[S_{\text{table}}] \approx \frac{1}{N} \sum_{i=1}^{m} \sum_{j=1}^{t} e^{\frac{-ijt}{N}}$$

We can estimate a sum with an integral from 0 to t:

$$\Pr[S_{\text{table}}] \approx \frac{1}{N}\left(\sum_{i=1}^{m}\left(\int_{j=0}^{t} e^{\frac{-ijt}{N}}\,di\right) - \frac{1}{N}\right)$$

$$\Pr[S_{\text{table}}] \approx \frac{1}{N}\left(\sum_{i=0}^{m}\int_{j=0}^{t} e^{\frac{-ijt}{N}}\,di\right) - \frac{m}{N^2}$$

The $\frac{m}{N^2}$ term is very close to 0 when $m \ll N$:

$$\Pr[S_{\text{table}}] \approx \frac{1}{N}\sum_{i=1}^{m}\int_{j=0}^{t} e^{\frac{-ijt}{N}}\,di$$

$$\Pr[S_{\text{table}}] \approx \frac{1}{N}\left(\sum_{i=1}^{m}\left(-\frac{N}{it} * e^{\frac{-it^2}{N}}\right) - \left(-\frac{N}{it} * e^{0}\right)\right)$$

$$\Pr[S_{\text{table}}] \approx \frac{1}{t}\left(\sum_{i=0}^{i=m}\frac{1 - e^{\frac{-it^2}{N}}}{\frac{it}{N}} * \frac{t}{N}\right) - \frac{1-1}{N}$$

Here we use another integral approximation of the sum:

$$\Pr[S_{\text{table}}] \approx \frac{1}{t}\int_{i=0}^{i=m}\frac{1 - e^{\frac{-it^2}{N}}}{\frac{it}{N}} * \frac{t}{N}di$$

$$\Pr[S_{\text{table}}] \approx \frac{1}{t}\int_{i=0}^{i=m}\frac{1 - e^{\frac{-it^2}{N}}}{i}di$$

Replacing $\frac{it^2}{N}$ with u ($u = \frac{it^2}{N}, du = \frac{t^2}{N}di$):

$$\Pr[S_{\text{table}}] \approx \frac{1}{t}\int_{u=0}^{u=\frac{mt^2}{N}}\frac{1 - e^{-u}}{\frac{u*N}{t^2}}\frac{N}{t^2}du$$

$$\Pr[S_{\text{table}}] \approx \frac{1}{t}\int_{u=0}^{u=\frac{mt^2}{N}}\frac{1 - e^{-u}}{u}du$$

If we use the $e^{-x} \approx 1 - x$ approximation here, we can create an equation for when multiple Hellman tables are used:

$$\Pr[S_{\text{TMTO}}] \approx 1 - (1 - \Pr[S_{\text{table}}])^l$$

$$\Pr[S_{\text{TMTO}}] \approx 1 - exp\left(\frac{l}{t}\int_{u=0}^{u=\frac{mt^2}{N}} \frac{1 - e^{-u}}{u} du\right) \tag{A.2}$$

In this thesis, we calculate this integral with the scipy function *integrate.quad* [20].