Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cascaded Bloom Filters in CRLite and their parameter selection

Bachelor Thesis

Philipp Engljähringer

May 6, 2022

Advisors: Prof. Dr. Kenny Paterson and Mia Filić

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

CRLite is an efficient fail-closed method to verifiy the revocation status of certificates used by TLS. It was introduced in 2017 at the IEEE Symposium on Security and Privacy and is currently deployed by Mozilla in the nightly version of Firefox. CRLite uses a probabilistic data structure called a Bloom filter cascade to construct a compact representation of the known revoked certificates. This data structure uses a hash function and a set of false positive probabilities. The purpose of this thesis is to introduce methods for the selection of a hash function and a set of false positive probabilities to improve certain properties of the Bloom filter cascade underlying CRLite.

To select an adequate hash function this thesis introduces a method to evaluate the performance of hash functions for their use in the Bloom filter cascade underlying CRLite. We proceed by introducing a parameter selection framework to select a set of false positive probabilities that minimizes certain properties of the cascade. This framework uses an optimization function that considers the size, the creation time and the time to perform set-membership queries with the cascade.

The experiments we conducted within this thesis lead us to recommend the cryptographic hash function `SHA3_256` and a set of 27 false positive probabilities to improve the performance of Mozilla's implementation of CRLite.

Acknowledgements

I would first like to express my profound gratitude to my thesis supervisors Mia Filić and Prof. Dr. Kenny Paterson and thank them for all their valuable guidance, mentoring and advice I have received throughout the process of developing and writing this thesis.

In addition, I am very grateful to my parents and my sister for their continuous support and their unwavering belief in me. Without their tremendous encouragement troughout the past few years I never would have gotten this far.

Finally, I want to thank my friends and my girlfriend for taking my mind off when I needed it, for the adventures we experienced together and for always reminding me of what is truly important in life.

Contents

Chapter 1

Introduction

Every day millions of users and corporations send tons of data over the internet. This data increasingly includes sensitive information such as credit card numbers, passwords and other private details that could cause harm if they ended up in the wrong hands. For this reason cyber security has become a major topic in the scientific community and a lot of effort went into the development of protocols and mechanisms to protect the users in the cyberspace.

Nowadays a great proportion of the web's traffic is secured through the Hypertext Transfer Protocol Secure (HTTPS) [26]. HTTPS uses the Transport Layer Security (TLS) [27] protocol to ensure authentication, confidentiality and integrity of web based communication channels. TLS itself relies on the Web's Public Key Infrastructure (PKI) in the form of digital certificates to perform two crucial functions, namely the encryption of data in transit and the authentication of the communicating parties.

One crucial facet of the Web's PKI is the ability to revoke certificates that are no longer trustworthy. Eventhough certificate revocation is of great importance for the security guarantees of HTTPS is often overlooked, thereby introducing new threats to users by giving them a false sense of security. This is due to the fact that all of the proposed methods for verifying the revocation status of certificates have issues reaching from massive overhead in website loading times to the introduction of threats to the users privacy [18].

To tackle the challenge of disseminating certificate revocation information without the introduction of massive overheads or privacy concerns James Larisch *et al* suggested a mechanism called CRLite [16], which is currently deployed by Mozilla. CRLite uses a probabilistic data structure called a Bloom filter cascade to construct a compact representation of the revoked certificates and then broadcasting this filter to all browsers. A Bloom filter

cascade can answer set-membership queries in constant time and therefore only adds a small overhead to website loading times. After downloading it the filter can be queried locally and therefore does not threaten the privacy of the useres by revealing their browsing behavior to third parties.

For the construction of a Bloom filter cascade one has to choose an adequate hash function and a set of false positive probabilities. Those choices will influence the size of the resulting cascade, the time it takes to create the cascade and the time needed to verify the revocation status of a certificate using the cascade. It is of great interest to minimize all the mentioned properties since they all have great impact on the introduced overheads and the added security guarantees of CRLite. The size will determine the needed bandwidth to push the filter to all browsers. The creation time will greatly influence the period in which one can publish a new filter and thereby the time window in which a compromized certificate can be used to cause harm. The time needed to verify the revocation status of a certificate with the filter will determine the overhead in web site loading times.

The choice of the hash function used for the construction of the filter influences all three properties we are interested in minimizing. Mozilla chooses the hash function for its CRLite implementation mainly focusing on the speed. This is in line with the choice recommended by the paper introducing CRLite.

For the set of false positive probabilities, Mozilla decided to follow the suggestion from the paper introducing CRLite. The authors recommend the use of two different false positive probabilities $0< p_1, p <1$. The first is used for the first layer of the cascade and the latter for all other layers. This strategy was recommended by James Larisch *et al* because their analysis showed that it produces a filter cascade with a size that is competitive with the theoretical lower bound.

The purpose of this thesis is to introduce methods for the selection of the hash function and a set of false positive probabilities for the Bloom filter cascade underlying CRLite. Those methods take all three properties -namely the size, the creation time and the time to verify a certificate- into account and can be extended to consider other properties if desired.

In Chapter 2 we introduce hash functions, Bloom filter and Bloom filter Cascades. In Chapter 3 we present the cornerstones on which HTTPS is built and explain the treats that can be introduced by omitting the verification of the revocation status of certificates. We then present the currently available methods for certificate revocation and explain the overheads and threats to the useres privacy that occur when using those methods. At the end of this chapter we present the fail-closed certificate revocation verification system called CRLite. In Chapter 4 we present a strategy to evaluate the performance of different hash functions in order to determine a suitable one

for the Bloom filter cascade underlying CRLite. In Chapter 5 we present a parameter selection framework for the selection of a set of false positive probabilities that allows for more than two false positive values and focuses on minimizing various properties of the cascade. We will also present and discuss the results we obtained by using this method to minimize the size, the creation time and the time to verify a certificate with the cascade.

Chapter 2

Background

2.1 Cryptographic hash functions

A Hash function is a computationally efficient function that maps some binary input string of variable length to a binary output string of fixed length. The output of a hash function is called the hash value or the digest. The idea behind hash functions is that the hash value is a compact representation of the input string. A cryptographic hash function [10], often also called one-way hash function, is a hash function that fullfills the three properties listed below to some degree. The recommended degree to which the properties need to be fulfilled evolves over time. There always might be applications that do not require the recommended degree of one or all three properties. However, this needs to be properly justified.

1) **Collision resistance:** It is computationally infeasable to find two different inputs x and y that result in the same hash value $hash(x) = hash(y)$.

2) **Preimage resistance:** Given a hash value h it is computationally infeasable to find an input string x such that $hash(x) = h$. This property is also often referred to as the one-way property.

3) **Second preimage resistance:** Given an input string x it is computationally infeasable to find a second input string y that will result in the same hash value $hash(x) = hash(y)$

The degree to which a hash function fulfills those properties is measured by the amount of computing power one has to spend in order to ”break” them with high probability. The collision resistance of a cryptographic hash function is expected to be half the length of the hash value. The expected preimage resistance is the length of the produced hash value. The second preimage resistance is also expected to be the equal to the length of the produced hash value, but for some hash functions this property also depends

on the input length. For example the hash function SHA_256 produces a hash value of 256bits and therefore has an expected collision resistance of 128bits, an expected preimage resistance of 256bits and an expected second preimage resistance of 256bits.

2.1.1 Building paradigms for cryptographic hash functions

Merkle-Damgard Construction

The Merkle-Damgard constrction method uses a collision resistant one-way compression function that can be applied to a fixed length input to construct a collistion resistant hash function that can be applied to inputs of arbitrary length. Ralph Merkle [20] and Ivan Damgard [9] independetly proved that this construction scheme is sound under the assumption that a used compression function is collision resistant and an appropriate padding scheme is used. First the padding scheme is applied to the input message and the binary encoding of the message length is appended. This is done to obtain a message with a length that is a multiple of some fixed number. The compression function is then iteratively applied to fixed sized chuncks of the padded message and the output value of the previous compression round. The first iteration of the compression function uses a fixed initialization vector and the output of the last iteration is the final hash value [28].

HAIFA Construction

The Hash Iterative Framework (HAIFA) is a modified version of the Merkle-Damgard construction which was proposed by Dunkelman and Biham [7]. It consists of the same steps as the Merkle-Damgard construction but it uses some additional input parameters in each iteration of the compression function. These parameters are a salt value and the number of bits hashed so far. This method preserves all important security properties and avoids some generic attacks of the Merkle-Damgard construction but suffers from degradation in efficiency [28].

Sponge Construction

The sponge construction [15] is a iterative method to obtain a hash function with variable length input and arbitrary output length. Key component is a transformation or permutation function that operates of fixed-length input. After the message is padded this construction continues in two phases:

1) Absorbing Phase: Fix sized chuncks of the padded input message are XORed into the internal state interleaved with applications of the underlying function. When all input chuncks are processed the squeezing phase begins.

2) Sqeezing Phase: Fixed sized chuncks of the internal state are iteratively returned interleaved with applications of the underlying function. The returned stream of bits serves as the hash value.

2.1.2 Secure Hash Algorithms:

The Secure Hash Algorithms are a group of standardized cryptographic hash functions. They are published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS). The first Secure Hash Algorithm was published in May 1993 [22] and is often reffered to as SHA0. It was developed by NIST in corporation with the NSA as part of the Digital Signature Algorithm [6]. SHA0 was withdrawn shortly after publication without an official reasoning. In 1995 NIST published a revised version of SHA0 [23] which introduces an additional rotate operation in the message expansion and is commonly referred to as SHA1 [6]. SHA0 and SHA1 produce a message digest of 160bits and are based on a Merkle-Damgard construction. SHA1 was widely deployed and viewed as the standard for collision resisitant hashing but it was later shown that one could find collisons via a birthday attack using 2^{80} evaluations of the function in expectance [6]. In 2005 Xiaoyun Wang *et al* presented a new technique to find collisions in SHA1 with 2^{69} evaluations of the function [30]. In 2002 NIST published the specifications of the SHA2 family [24].

SHA2 The SHA2 family consists of 4 different hash functions, namely SHA_224, SHA_256, SHA_384 and SHA_512 which produce a hash value equal to their respective suffixes. SHA_224 and SHA_384 are derived from SHA_256 and SHA_512 by truncating the final output. They are based on a Merkle-Damgard construction and use a Davies-Meyer compression function. Their cryptographic strenghths are summarized in Table 2.1.

SHA3 In 2006 NIST organized a hash function competition to create a new hash standard. The winner of this contest was annonced in 2015 and is now known as the SHA3 family [12]. The family consists of four cryptographic hash functions called SHA3_224, SHA3_256, SHA3_384 and SHA3_512. They each produce a hash value equal to their suffixes. They are complemented by two extendable-output functions (XOFs) called Shake128 and Shake256 which can produce an infinite output stream. Unlike their predecessors they are based on a sponge construction and internally use the KECCAK-f cryptographic permutation designed by Guido Bertoni, Joan Daemen, Michael Peeters and Gilles Van Assche [3]. Their cryptographic strengths are summarized in Table 2.1.

Function	Output Size	Collision	Preimage	2nd Preimage
SHA_1	160	<80	160	160-L(M)
SHA_224	224	112	224	min(224,256-L(M))
SHA_256	256	128	256	256-L(M)
SHA_384	384	192	384	384
SHA_512	512	256	512	512-L(M)
SHA3_224	224	112	224	224
SHA3_256	256	128	256	256
SHA3_384	384	192	384	384
SHA3_512	512	256	512	512
Shake128	d	min(d/2, 128)	≥min(d,128)	min(d,128)
Shake256	d	min(d/2, 256)	≥min(d,256)	min(d,256)

Table 2.1: Cryprographic strength of SHA hash functions. Numbers were taken from [12] Table 4.

2.1.3 Siphash

Siphash [4] is a family of add-rotate-xor based pseudorandom functions that were introduced in 2012 by Jean-Philippe Aumasson and Daniel J. Bernstein. It is optimzed for short inputs and it's target applications include network traffic authentication and hash-table lookups. Siphash uses a 128bit key and produces a 64bit output for a variable length input. During the compression and the finalization rounds Siphash iterates a simple round function called SipRound. This functions consists of four additions, four xors and six rotations, interleaved with xors of the intermediate message blocks. Siphash_c_d uses c iterations of the round function during the compression step and d iterations during the finalization step. The recommended versions of Siphash for the best performance are Siphash_2_4 and Siphash_4_8 for more conservative security properties. Siphash is cryptographically strong but since it does not provide collision resistance it is not cryptographically secure.

2.1.4 Blake

The initial Blake [5] hash function family was proposed by Jean-Philippe Aumasson *et al* and was submitted to the NIST hash function competition in 2008. It was one of the three finalists and uses a HAIFA construction. The family consists of four distinct hash functions called Blake-224, Blake-256, Blake-384 and Blake-512. They each produce a hash value equal to their suffixes. In 2012 the Blake2 family was introduced which consists of two hash functions called Blake2b and Blake2s which produce a hash value of 512bits and 256bits. They are complemented by two extendable-output function called Blake2bp and Blake2sp [1]. The cryptographic strength of

Function	Output Size	Collision	Preimage	2nd Preimage
Blake-256	256	128	256	256
Blake-512	512	256	512	512
Blake2s	256	128	256	256
Blake2b	512	256	512	512

Table 2.2: Cryprographic strength of Blake hash functions. Numbers were taken from [2].

the Blake functions are summarized in Table 2.2.

2.2 Bloom Filter

A Bloom Filter [8] is a probabilistic data structure which can be used to construct a compact representation for a set of elements. This representation allows queries for insertion and for set-membership with constant time. The price of compactness and constant time is the introduction of uncertainty. The way a Bloom filter is constructed allows for queries to have a certain false positive probability, meaning that the query returns the element is in the set even though it is not. A false negative on the other hand is not possible.

A Bloom Filter consists of a bitarray b of size m and a set of hash functions $\{h_1(), ..., h_k()\}$. Initially all m bits of the bitarray are set to 0. In order to insert an element e, we first calculate the k hash values of e $h_1(e), ..., h_k(e)$. We use those hash values as an index for the bitarray and set the corresponding bit-cells to 1. To answer a query for set-membership for an element e', we calculate the k hash values for e' and again use them as an index into the bitarray. If all the corresponding cells are set to 1, the query returns that e' is contained in the set. If at least one cell is set to 0, it returns that e' is not contained. A set-membership query for an element that was inserted into filter will always correctly return *true*. For an element that was not inserted, there remains a certain probability that each of its k hash values collides with those of some other inserted elements. In this cast the membership query would return that the elements is in the set eventhough it is not. This probability is called the false positive probability of the filter. Figure 2.3 gives a visualization of Bloom Filters and Figure 2.2 shows the pseudocode for the creation of a Bloom Filter used in Mozilla's CRLite implementation.

Normally, when using a Bloom Filter, the false positive probability p is set as a design constraint which represents the tradeoff between uncertainty

and size of the resulting filter. A certain false positive probability p can be achieved by setting the size m of the bitarray and the number of hash functions k accordingly. It was shown that for a given false positive probability p the size of the resulting Bloom filter can be minimized by setting

$$k = log_2(1/p) \tag{2.1}$$

$$m = (r * ln(1/p))/(ln(2))^2 \approx 1.44 * |R| * log_2(1/p) \tag{2.2}$$

as explained in [21]. The derivation of those formulas make the assumtions that m and r are large, that $log_2(1/p)$ is close to integral and that the used hash functions h_i are perfectly uniform. Now let us also make the assumption that all set-membership queries come from a finite known set U and we want to use a Bloom Filter to distinguish whether an element is in the set $R \subseteq U$ or in $S = U \setminus R$. The problem is if a set membership query returns that an element was inserted in the filter, we cannot be sure if it is a true positive or a false positive. But since the expected number of false positives is $E(|fp|) = p * |S|$ we can see that the set of true positives plus the set of false positives is in expectation strictly smaller than the initial set U for a probability $p < 1$:

$$E[|fp|] + |R| = p * |S| + |R| < |S| + |R| = |U|$$

This observation leads us to the idea that one could use a second Bloom Filter to distinguish between the true positives and the false positives. We can then use a third filter to distinguish the true and false positives of the second filter and so on. Since the sizes of the used sets keep decreasing in expectation at each layer, we will at some point reach a layer where there are no more false positives. And this insight brings us to the topic of Bloom Filter Cascades or Multi Level Bloom Filter.

a Insert element into a Bloom filter

```
1: procedure INSERT(b,elem,{h_1(),...,h_k()}))
2:     for h_i() in {h_1(),...,h_k()} do
3:         index ← h_i(elem) mod |b|
4:         b[index] = 1
5:     end for
6:     return b
```

b Set-membership query

```
1: procedure CONTAINS(b,elem,{h_1(),...,h_k()}))
2:     for h_i() in {h_1(),...,h_k()} do
3:         index ← h_i(elem) mod |b|
4:         if b[index] == 0 then
5:             return False
6:         end if
7:     end for
8:     return True
```

Figure 2.1: Pseudocode for the set-membership query and the insert operation for a Bloom filter. The **Input** parameters are b:= the bitarray of the Bloom filter, *elem*:=The element to insert or perform set-membership query, $\{h_1(), ..., h_k()\}$:= The set of hash functions to use. The **Output** of a) is the updated bitarray and b) returns a boolean value indicating if the element is in the Bloom filter or not.

Algorithm 1 Generate a Bloom filter

1: **procedure** GENERATE_BLOOM_FILTER($include\_set, p$)
2: $m \leftarrow \lceil 1.44 * |include\_set| * log_2(1/p) \rceil$
3: $k \leftarrow \lceil log_2(1/p) \rceil$
4: $bitarray = 0^m$
5: $\{h_1(), ..., h_k()\} \leftarrow$ CHOOSE_HASHES(k)
6: **for** elem in $include\_set$ **do**
7: $b \leftarrow$ INSERT($bitarray, elem, \{h_1(), ..., h_k\}, m$)
8: **end for**
9: **return** $bitarray$

Figure 2.2: Pseudocode for the generation of a Bloom filter. The **Input** parameter is the *include_set*:= The elements that should be inserted in the filter. The **Output** is a Bloom filter in which the elements provided as input were inserted.

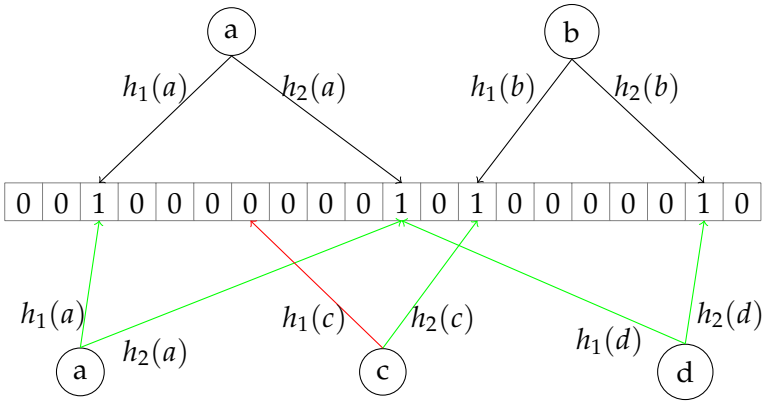


Figure 2.3: This Figure visualizes the insert and the lookup operation for a Bloom Filter. We use a bitarray of size 20 and 2 hash functions $\{h_1(), h_2()\}$. The elements $\{a, b\}$ (top) get inserted into the bitarray. A membership-query is performed for elements $\{a, c, d\}$ (bottom). The membership-queries result in a true positive for a, a true negativev for c and a false positive for d.

2.2.1 Cascaded Bloom Filter

The idea of using multiple Bloom Filter in a Cascade $BFC = [BF_1, BF_2, ...]$ can be used to construct a compact representation for a set of elements which

supports set-membership queries without the possibility of a false positive. However, this is only possible under the assuption that all set-membership queries come from a finite set U which is known at creation time.

Constructing a Bloom Filter Cascade: If we want to construct a Bloom Filter Cascade for a set $R \subseteq U$ and a finite known set U, we start by constructing the first filter BF_1. To do so, we pick a false positive rate p_1 and set the size m_1 and the number of hash functions k_1 of the filter according to the Formulas 2.2 and 2.1. After we inserted all elements of R into the bitarray, we proceed to do a set-membership query for all elements in $S = U \setminus R$. All queries with a positive return value represent a false positive and we therefore add the elements that triggered them into the false positive set fp_1. Now we want to store the elements of fp_1 in a second Bloom Filter BF_2 and we therefore initialize another Bloom Filter BF_2 with m_2 and k_2 according to p_2 and the Formulas for optimal k (2.1) and m (2.2). Aftwerwards we proceed to add all the elements of fp_1 to the bitarray. The purpose of the second filter is to be able to distinguish whether and element $e \in U$ for which the set membership query to the first filter has a positive return value, is indeed an element of R (true positive) or an element of S (false postive). Hence we want the set membership query to the second filter to have a positive return value for all elements in fp_1 and a negative return value for all elements in R. However there might be some elements from R that when queried to the second filter result in a positive return value.

We can then use a third Bloom Filter BF_3 to distinguish between the true and the false positives of the second layer and another to distinguish the true and false positives of the third filter and so on.

In essence, we use the i-th Bloom Filter BF_i to distinguish between the true and the false positives of the layer $(i-1)$. Hence we want to insert the false positives of BF_{i-1} and use the true positives of layer $(i-1)$ to search for false positives of layer i. We denote them as the include and exclude set for filter i:

$$include\_set_i = fp_{i-1}$$

$$exclude\_set_i = include\_set_{i-1}$$

This process is repeated until we reach a filter with no false positives. At that point we have eliminated the uncertainty of standard Bloom Filters and have nevertheless obtained a compact representation for R which allows for set-membership queries for all elements in $U = R \cup S$ without the possibilty of false positives or false negatives.

We know that if layer 1 contains e but layer 2 does not, e is in R. The way we constructed the cascade this can be be generalized to all layers. We always

continue to look up e in all layers in order until we reach the first one that does not contain e. Let us assume that the first layer that does not contain e is layer i. We can then answer the query in the following way:

$$\text{if} \quad i \text{ is } odd \Rightarrow e \quad is \quad in \quad S$$

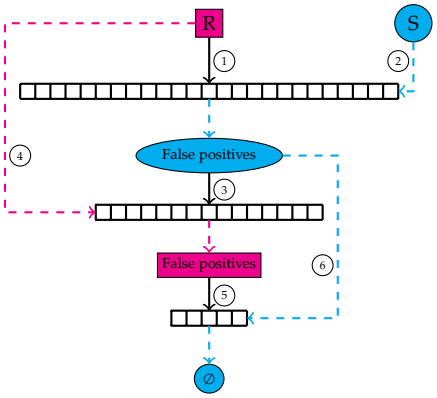
$$\text{if} \quad i \text{ is } even \Rightarrow e \quad is \quad in \quad R$$

If element e is included in all layers of the cascade then the answer depends on n, the number of layers of the cascade:

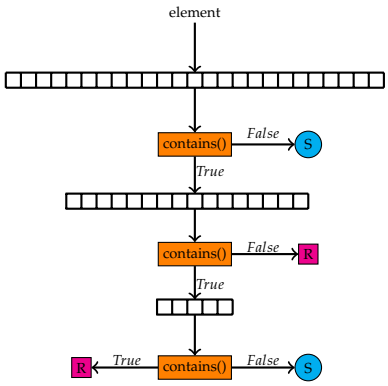
$$if \quad n \quad is \quad odd \Rightarrow e \quad is \quad in \quad R$$

$$if \quad n \quad is \quad even \Rightarrow e \quad is \quad in \quad S$$

Figure 2.4 shows a visualization of the creation process and the set-membership queries of Bloom filter cascades. Figure 2.5 shows the pseudocode for the creation process of a Bloom filter.



(a) Creation process of a Bloom Filter cascade. Solid arrows represent insert operations, dashed arrows represent set-membership queries to find false positives. The indizes show the order in which the steps are executed.



(b) Set-membership query for a BFC. We lookup the element in all filters in order to determine the set it belongs to.

Figure 2.4: Subfigure a) shows the creation process of a Bloom Filter cascade. Subfigure b) shows the set-membership query of a Bloom Filter cascade. The set R holds the elements that get inserted and S holds the elements that are not inserted but can be queried for set-membership. Figures were inspired by [16] Fig. 1 and 2.

Algorithm 2 Generate a Bloom filter cascade

```
1:  procedure GENERATE_BFC(include_set, exclude_set, {p_1, ...}))
2:      cascade ← []
3:      depth ← 1
4:      while |include_set| > 0 do
5:          cascade_depth ← GENERATE_BLOOM_FILTER(include_set, p_depth)
6:          fp_set ← ∅
7:          for elem in exclude_set do
8:              if CONTAINS(cascade_depth, elem) then
9:                  fp_set.add(elem)
10:             end if
11:         end for
12:         exclude_set ← include_set
13:         include_set ← fp_set
14:         depth ← depth + 1
15:     end while
16:     return cascade
```

Figure 2.5: Pseudocode for the generation of a Bloom filter cascade. The **Input** parameters are the *include_set* and the *exclude_set* for the first Bloom filter in the Cascade. The **Output** is a Bloom filter cascade that allows set-membership queries for all certificates in the provided input sets.

Chapter 3

Certificate revocation and CRLite

3.1 The Web's Public Key Infrastructure

Nowadays many websites and other internet applications require some securitiy features for the underlying communication channel. The Hypertext Transfer Protocol Secure (HTTPS) [26] uses Transport Layer Security (TLS) [27] coupled with the Web's Public Key Infrastructure (PKI) to create a communication channel which provides authentication, confidentiality and intergrity for 2 or more applications communicating over the web. The TLS protocol iteself mostly relies on X.509 certificates, which have to be issued and cryptographically signed by a trusted Certificate Authority (CA). The certificate thereby binds an identity (domain name) to a public key by the use of a digital signature. X.509 Certificates can be extended to also contain a Policy Verifier which indicates that the CA has performed additional steps to verify the identity of the issuer. Those certificates are called EV-certificates (Extended Validation) and are meant to provide greater assurance to clients that the issuer was properly verified. Each user has to chose a set of trusted root CA's. Those root CA's can issue intermediate CA's which in turn can issue more intermediate CA's and also leaf certificates. The owner of a leaf certificate cannot issue any certificates. When the user tries to establish a HTTPS connection with a web-site, the server sends a series of certificates to the client. This series has to form a chain of trust which starts in one of the root CA's trusted by the client and ends in the leaf certificate of the web-service you are trying to establish a connection with. The client verifies that chain of trust and if all certificates check out, he can be sure that a message signed with the private key of the leaf certificate comes indeed from the owner of the certificate. To verify the chain of trust sent to the client, he has to verify the signatures of the certificates, review the validity period, and check whether the root CA is in his set of trusted root CA's. The last step of this verification process is to check whether one of the certificates in the obtained chain of trust has been revoked [18].

3.1.1 Certificate revocation:

A certificate which is no longer trustable prior to its expiration date can be revoked by the issuing CA. The owner of the certificate can instruct the issuing CA to revoke the certificate at any point in time. A certificate can become untrustworthy for numerous reasons, such as a change in the useage of the certificate or if the owner of the certificate becomes untrustworthy. But the most important reason is if the private key of a certificate has been compromized. If those cases occur and the certificate does not get revoked, all the users will keep trusting the owner of the certificate until it reaches its expiration date. An attacker could use such a untrustworthy but non-revoked certificate to perform effective Man-in-the-Middle and phishing attacks against clients. If the certificate in question is an intermediate or a root certificate whose private key was compromized, the attacker could even forge valid vertificates for any domain he wishes. Therefore it is important that those certificates get revoked and that the client checks the revocation status of the certificates before establishing a connection. Currently there are two widely adoped methods to check for the revocation status of a X.509 certificate:

Certificate Revocation List (CRL): A CRL is basically a list of tuples of the form (serial number, revocation-timestamp, reason for revocation). These tuples are then collectively signed by the CA that publishes the CRL. To verify the revocation status of a certificate with this method, the user has to download the CRL file and then check whether the serial number of the certificate in question appears in the list. The main problem with this method is that the client has to periodically download the information about all revoked certificates, even if he is only interested in some. Since those revocation lists can grow to sizes of $>$ 70MB it will not only take a lot of bandwidth to periodically download the latest CRL, but can become a storage-problem on resource constrained devices like mobile phones.

Onlince Certificate Status Protocol (OCSP): OCSP was designed to avoid the overhead of Certificate Revocation Lists. With OCSP a client can generate a HTTP request to a CA, to query it for the revocation status of a single certificate's serial number. The CA then returns a signed response for the certificates current revocation status. This method brings a potential privacy risk as it reveals the browsing behavior of the client to the queried CA. In addition, the OCSP request will increase the loading time of a webservice by its roundtrip time. To avoid this problem the OCSP Stapling extension was introduced, which allows the server to cache the OCSP responses for the certificates in its chain of trust and include them in the TLS handshake with the client. Therefore the clients recieve the servers certificate and the OCSP response at the initialization of the connection and do not have to query a third party themselves. This extension also avoids the privacy concerns since

the third party would now be queried by the server and therefore could not infer who accessed the service.

Revocation checking in practice: We explained above that the threats that occur by not checking for the revocation status of certificates are immediate and that there are methods in place to do so.

In March 2020 George Huston conducted a test where he used Let's Encrypt to generate a valid certificate and then revoked it. He then used various browsers to connect to a site that uses this revoked certificate to observe their behaviour. On Windows 10 the browseres Chrome (version 80.0.3987.132) and Opera (version 67) did not detect the revocation and connected to the website. On Android 10 all tested browsers, namely Chrome (version 80.0.3987.132), Firefox (version 68.6.0) and Opera (version 56.1), failed to detect the revocation of the certificate [13].

Apart from the reasons we explained above (storage, privacy, latency) this is mostly due to the fact that all existing systems adopt a fail-open model. A fail-open model means that if they cannot determine the revocation status of a certificate, (e.g. if the browser cannot resolve the domain name of the CRL server or the OCSP server is down), they simply view the certificate as valid. Therefore an attacker can simply block all revocation status requests and thereby disable the whole revocation checking mechanism. A fail-closed solution would assume a certificate to be invalid if its revocation status can't be determined and would therefore be more secure, but browser vendors argue that implementing it that way with the currently available methods would lead an unacceptable level of failures for the client. To address these problems Mozilla has deployed a mechanism called CRLite which we will explain in the next section [16].

3.2 CRLite

CRLite [16] is a system for revocation checking that proactively pushes all certificate revocations to all browsers instead of relying on a pull model where clients download the revocation information on-demand. It was presented in 2017 at the IEEE Symposium on Security and Privacy by James Larisch *et al* [16]. The following section summarizes how CRLite works, what challenges it addresses and how the parameters for the Bloom filter cascade were chosen. For more details we refer to the original paper. CRLite consists of a server-side system and a client-side system.

Server-side: VanderSloot *et al* have shown that $> 99\%$ of all TLS certificates existing in the web can be obtained by using full IPv4 scans on port 443 and Google's CT logs [29]. The server-side uses those methods to periodically

aggregate information for all obtainable TLS certificates. It then extracts the OCSP and the CRL responders from those certificates, downloads all extracted CRL's and queries the respective OCSP responders to determine the revocation status of the certificates. It therefore divides the set of all obtained TLS certificates U into two mutual exclusive sets of valid certificates S and revoked certificates R. We explained in Section 2.2 that under those conditions one can construct a compact representation for those sets which allows for set-membership queries with a definite answer through a Bloom Filter Cascade. The set of revoked certificates will be inserted in the first Bloom filter and is denoted as R. The set of valid certificates is used to search for false positives in the first Bloom filter and will be denoted as S

Client-side: The client-side downloads the Bloom Filter Cascade constructed by the server-side and can then query it locally to determine the revocation status of the certificates it encounters.

This method of checking certificate revocation adresses six challenges:

1) **Efficiency:** During April 2022 CRLite included around 175Million certificates and the resulting size of the resulting Bloom filter cascade was about 6.8MB.
2) **Timeliness:** CRLite constructs a new filter 4 times per day.
3) **Fail-closed:** CRLite contains all revocations and the filter cascade is stored locally which eliminates the problem of unreachability of the responder and therefore allows the client to adopt a fail-closed model without an unreasonable amount of connection failures.
4) **Privacy:** After downloading the filter cascade clients can query it locally and therefore don't reveal their browsing behaviour to any third party.
5) **Deployability:** CRLite does not require any changes to the current TLS system nor any additional steps by the CA's and can therefore easily be integrated into any browser today.
6) **Auditability:** CRLite provides cryptographically signed logs which allow any client to audit/verify the filter it downloads.

False Positive Rates: For the construction of a Bloom Filter cascade, one has to chose a certain error rate for every filter in the cascade. Mozilla deployed the strategy using one false positive probability p_1 for the first layer and a second false positive probability p for all subsequent layers as suggested by the paper introducing CRLite. The authors suggested to use

$$p_1 = |R| * \frac{\sqrt{0.5}}{|S|} \tag{3.1}$$

in order to get an expected include-set size for layer 2 of size $|R| * \sqrt{p}$. Therefore the ratio between the expected include set size and the exclude set size for layer 2 is $\sqrt{p}$. For p they suggest a false positive probability of 0.5 in order to maintain the same ratio between the expected include set size and the exclude set size throughout all layers:

$$E[|include_i|]/|exclude_i| = E[|fp_{i-1}|]/|exclude_i| = \sqrt{p}$$

This strategy was propsed because their analysis has shown that it will result in a Filter Cascade with a size that is competetive with the theoretical lower bound. Other criteria like creation time and number of hashes were not considered for the choice of the false positive probabilities [16].

Hash functions and rounding: CRLite does not use a set of hash functions but only one specific hash function. To obtain different hash values on different layers, it uses a concatenation of the hash number $1, ..., k$ and the current layer as a seed. To obtain integer values for the size of the bitarray and the number of hash functions CRLite uses the ceil function.

Chapter 4

Hash function selection for CRLite

The choice of the hash function to use, for the creation of the filter, is of great importance. Not only can this choice introduce or avoid security risks, but it will influence all 3 major properties of the filter creation, we are trying to optimize. Namely the size, the creation time and the lookup time.

4.1 Current Implementation

The current implementation of CRLites uses Murmerhash3 for the construction of the filter. Murmerhash3 was chosen simply because it is designed for speed. Other hash functions were tested, but none were faster and apparently no other criteria, like cryptographic properties, were considered [16].

4.2 Important properties

In order to avoid security risks in the future, we will only consider cryptographic hash functions.

A great amount of the time needed to create the filter, or do a lookup in the filter, is spent with calculating hashes. Therefore the lookup time and the creation time are highly dependent on the speed of the used hash function, which is why Murmerhash3 was chosen in the first place. Hence we also have great interest in picking a fast hash function, and will therefore consider speed as the second criteria.

The last property we will consider is the uniformity of the hash functions, which will have high impact on the size of the Bloom Filter. It was shown that for a Bloom Filter, the optimal results in terms of size are achieved when the optimized Bloom filter looks like a random bit string, meaning each bit in the filter should be 0 or 1 with probability of 1/2. This distribution of

1s and 0s could be achieved by using a fully uniform hash function during the creation of the Bloom Filter [21]. In order to minimize the bandwidth needed to download a new filter, we want our cascade to be as small as possible, and hence we want our chosen hash function to be as uniform as possible.

4.3 Evaluating hash functions

Measuring the uniformity of a hash function is quite difficult. But we are not actually interested in the uniformity itself, but in its influence on the size. Hence instead of trying to quantify the uniformity of our considered hash functions, we are going to generate some random certificate sets and then use those hash functions to create a filter cascade. We then use the size of those cascades to compare the performance of the hash functions.

Measuring the speed of a hash function on a CRLite-certificate, could be done by simply using the average time it takes to compute the hash for one certificate. We could obtain this value by generating a random set of certificates and measuring the time it takes to compute the hashes for all elements in the set. We then simply have to divide by the size of the set to get the average hash time.

As explained above, the majority of the creation time and the lookup time is spent calculating the hashes of the certificates for every layer. Therefore the lookup time and the creation time for a filter, not only depend on the time it takes to compute the hashes, but also on the number of hashes we have to compute. To take this into account we will measure the time we spent calculating hashes during the creation of the cascade and use this value to compare the performance of the functions.

4.4 Generating random CRLite-certificates

Each certificate consists of an issuer's subject public key info (spki) hash, which represents the Certificate Authority that signed the certificate, and a certificate serial number. As of December 2021, Firefox has 580 Certificate Autorities enrolled in their CRLite system. For each certificate we generate, we will pick one of those 580 spki hashes at random.

The serial number of a certificate has to fulfill certain properties but can otherwise be freely chosen by the signing Certificate Authority.

" As per RFC 5280 §4.1.2.2, serial numbers MUST be unique, not greater than 20 bytes long non-negative integer and at least 1 bit must be enabled in first byte. If first byte is zero, this byte is truncated unless it is the only byte.

This means the first byte must be in range 0x0÷0x7f with a whole integer value up to 20 bytes. 0x00 0x00 0x01 serial number is truncated to 0x01. Various CA engines implement different serial number generation algorithms. There is no restriction to CA's on using sequential serial numbers, starting with 0x00, 0x01, 0x02, etc. (excluding 0x80÷0xff range for most significant byte) " [25]. In 2008 a vulnerability in the Internet Public Key Infrastructure used to issue digital certificates for secure websites, was found and exploited. This vulnerability took advantage of a weakness in the MD5 hash function and thereby enabled the attackers to obtain a rogue Certification Authority certificate which was trusted by all common web browsers. This attack can only be used against Certification Authorities that enable us to predict their future serial numbers by using sequentially generated serial numbers [11]. For this reason nowadays most Certification Authorities use cryptographically random serial numbers.

To chose the length of the serial numbers we want to use, we picked a reference dataset and observed that most of the serial numbers used were between 11 and 18 bytes long. Therefore for our certificates we will first generate a random number between 11 and 18 and then generate a random bytestring of that length to use as serial number.

Algorithm 3 Generating random CRLite certificates

1: **procedure** GENERATE_RANDOM_CERTIFICATE($spki\_hash\_set$)
2: $random\_spki\_hash \leftarrow$ PICK_RANDOM($spki\_hash\_set$)
3: $serial\_number\_len \leftarrow$ PICK_RANDOM($\{11, ..., 18\}$)
4: $random\_serial\_number \leftarrow$ GEN_RANDOM_BYTES($serial\_number\_len$)
5: **return** ($random\_spki\_hash, random\_serial\_number$)

Figure 4.1: Pseudocode for generating random CRLite certificates. The **Input** parameter is *spki_hash_set*:= A set of subject public key hashes to use. The **Output** is a randomly generated CRLite certificate.

4.5 Experimental setup and results

Before we can start our simulation we have to choose adequate sizes for the sets of revoked and valid certificates. For the set of revoked certificates we chose to use the number of revoked certificates that Mozilla included in their cascade during December 2021, which is around 5.200.000. The false positive probability on the first layer will ensure that the include set and the exclude set of the second layer have an expected ratio of $\sqrt{p}$. Therfore the size of the set of valid certificates will not influence the expected sizes for

the include and exclude sets for the higher layers and neither the expected number of layers the cascade has. To save time we therefore chose to not use the number of valid certificates that Mozilla included during december 2021, but only one tenth of it. This number comes down to 15.600.000. We will use the same set sizes for all experiments presented in this thesis.

Each simulation run consists of two steps:

1) Generate random certificate sets We begin by generating random sets of revoked and valid certificates. This can be done through repeated calls to the `GENERATE_RANDOM_CERTIFICATE()` function given in Figure 4.1. To use less main memory we are going to store the valid certificate set on the disk. For our choice of set sizes the include and exclude sets after the first layer are small enough such that we can continue using only main memory for the subsequent layer.

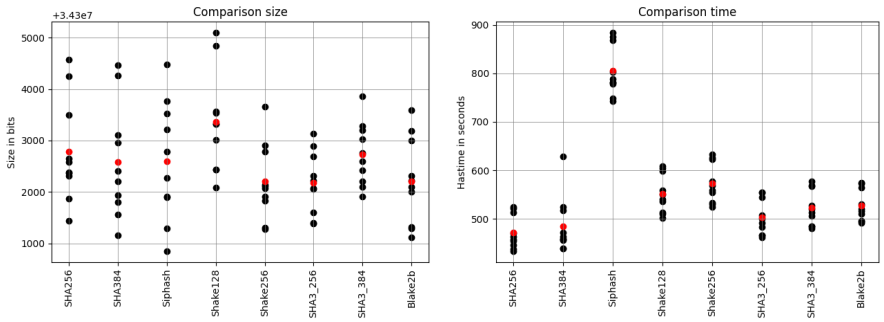
2) Creating Filter Cascades With the random sets generated, we can start to create a cascade on this input, using every hash function we want to study in turn. During the creation we will measure the time we spent caluclating the hashes needed for it's construction. When the creation of the cascade is finished, we return the size of the cascade and the time measurements for the hashes.

Algorithm 4 Performance evaluation of hash functions

```
1: procedure EVALUATE_PERFORMANCE(hash_set, num_revoked, num_valid)
2:     revoked_set ← GENERATE_RANDOM_CERT_SET(num_revoked)
3:     valid_file ← GENERATE_RANDOM_CERT_FILE(num_valid)
4:     p_1 ← num_revoked * sqrt(0.5) / num_valid
5:     fpp_set ← {p_1, 0.5, ..., 0.5}
6:     for hash_i in hash_set do
7:         size_i, total_hash_time_i ← GENERATE_BFC(revoked_set, valid_file, fpp_set, hash_i)
8:     end for
9:     return size, total_hash_time
```

Figure 4.2: Pseudocode to evaluate the performance of different hash functions for their use in the Bloom filter cascade underlying CRLite. The **Input** parameters are *hash_set*:=The set of hash functions to evaluate and *num_revoked*, *num_valid*:= number of revoked and valid certificates to use during construction of the BFC. The **Output** is the measurements of the size of the resulting cascade and the time needed to calculate the hashes during construction. We used the `GENERATE_BFC()` function presented in Figure 2.5 and added a fourth input parameter $hash_i$, which represents the hash funtion to use for the construction of the Bloom filters.

4.5.1 Result



(a) Comparison of size **(b)** Comparison of speed

Figure 4.3: Comparison of the measured properties of different hash functions.

We conducted 1000 test runs of our simulator with the hash functions `SHA_256`, `SHA_384`, `SHA3_256`, `SHA3_384`, `Shake128`, `Shake256`, `Siphash_2_4` and `Blake2b`.

We used the `Siphash_2_4` implementation of the python csiphash library. For all the other hash functions we used the implementation provided by the hashlib python library.

The sizes of the resulting cascades and the time needed to calculate the hashes during construction are plotted in Figure 4.3. Every black dot represents the average of 100 simulation runs and the red dots show the average over all 1000 simulation runs. Every simulation run was done with a new randomly generated set of certificates.

When examining Subfigure 4.3 a) we can see that on average `SHA3_256` resulted in the cascade with the least bits and it also had the least variance in terms of size. The second best hash function in terms of size of the resulting cascade was `Shake256` followed by `Blake2b`. In Subfigure 4.3 b) we can see that the fastest hash on average was `SHA_256` followed by `SHA_384` and `SHA3_256`. If one were to highly favorize the speed of the hash function we recommend the use of `SHA_256` since it was on average 30.579 seconds faster than `SHA3_256`. But the cascades constructed with `SHA3_256` had on average 611 bits less and we therefore recommend the use of `SHA3_256` over `SHA_256` for the use in the construction of the Bloom filter cascade underlying CRLite. We will continue to use `SHA3_256` for all following simulations in this thesis.

Chapter 5

Parameter selection for CRLite

In Subsection 2.2 we have shown that for a chosen error rate, one can calculate the ideal values for the size of the filter and the number of hash functions to use. The error probability is a design constraint of the filter cascade and there is no proven ideal value for it.

It was shown that the strategy of using one error probability p_1 for layer 1 and a second error probability p for all higher layers, will result in a filter cascade with a size that is competitive with the proven lower bound for a cascade. This strategy had slightly better results in terms of size than prior work which assumes a single error probability for all levels [16].

This sparks the idea of using multiple different error probabilities to further improve the properties of the resulting cascade. While prior work only chose their values to minimize the size of the cascade, we will also consider the time needed to create the cascade and the number of hashes needed to create the cascade. The number of hashes calculated during the creation of the cascade is always the same as the number of hashes one has to calculate to perform a set-membership query for every certificate used during creation. Since most of the time it takes to perform a set-membership query is spent with the calculation of the hashes, we will take this number as a measurement for the time it takes to perform a set membership-query with the cascade.

5.1 Naive Brute Force search

With infinite computing power and infinite memory, brute force searching the best parameters for all layers in a CRLite filter cascade would be no problem. One could do so by generating a random set of certificates and then create a cascade with it, for all possible combinations of error probabilities. After repeating this process a number of times, we pick the set of

error probabilities that on average resulted in the cascade which fullfilled our preferences towards the properties of the cascade the best. In the non-theoretical world the available amount of computing power and memory is always limited. Searching the interval [0.2,0.7] with a stepsize of 0.02 for 20 layers would require us to create 25^{20} cascades. Even if we could create a cascade in 1 microsecond, this would take at least $2.882 * 10^{14}$ years. Since this would clearly exceed the time frame of this thesis we will consider this approach infeasable.

5.2 Layer by Layer search

The approach described in the following section is inspired by the strategy suggested by James Laritsch *et al*. Instead of searching all possible combinations of parameters, we will try to optimize the false positive probabilities layer by layer. We will use the p_1 suggested in the paper introducing CRLite and will therefore start our parameter search at layer two. After we selected an adequate false positive probability for layer two we will continue our search for the probabilties for the higher layers one after another.

Optimization function f(): To pick an adequate false positive probability for each layer we introduce an optimization function that takes the size, the creation time and the number of hashes used during creation into account. This function can be extended to also consider other properties -like the number of layers- one wishes to minimize.

We first create a number of cascades with different sets of false positive probabilities and measure the number of hashes used, the creation time and the size of the resulting cascade. We then average the obtained measurements and store them in an avg_array, one per the property of interest.

To compare the cascades with different false positive probabilities we calculate the percentage increase from the minimal obtained value for all of our three measurements. We do so by first subtracting the minimial value from all measurements in the array and then dividing by it.

$$percent\_property_i(fpp) = \frac{avg\_property_i[fpp] - min(avg\_property_i)}{min(avg\_property_i)} \quad (5.1)$$

Putting all the obtained percentage arrays in one plot will show us the possible tradeoffs of the different false positive probabilities. We then proceed to calculate the weighted sum of those percentages.

$$\begin{aligned} f(fpp) =& w_1 * percent\_size(fpp) + \\ & w_2 * percent\_hashes(fpp) + \\ & w_3 * percent\_time(fpp) + \\ & \sum w_i * percent\_property_i(fpp) \end{aligned} \tag{5.2}$$

For our experiments we want to view the size, the number of hashes and the creation time as equally important and not consider other criteria. We therefore use $w_1 = w_2 = w_3 = 1$ and $w_i = 0$ for $i > 3$. We then select the false positive probability that resulted in the minimal value of our optimization function. To put emphasis on a property/properties one simply has to adjust the corresponding weight.

First Iteration: In the following description we are going to use the notation p_i^1 for the false positive probability we selected for layer i. For the false positive probability we are using on layer j during our optimization process we will use the notation p_j.

During the search for an adquate p_i^1 we are going to create the first $(i-1)$ layers of a cascade with the false positive probabilities we already selected for those layers. For every cascade we create we use a random set of valid and revoked certificates as described in Section 4.5.

$$p_l = p_l^1 \quad for \quad l = 2, ..., (i-1)$$

For each false positive probability we want to consider $fpp_{candidate}$ we copy the first $(i-1)$ layers and continue the creation of the cascade with the probabilities

$$p_h = fpp_{candidate} \quad for \quad h = i, ..., n$$

After we created a number of cascades we use the optimization function to select p_i^1 and continue searching for p_{i+1}^1. This process can be described by the pseudocode given in Figure 5.1. Once we chose a set of false positive probabilities for n layers, we will take them as a starting point to further improve the chosen set by reiterating all layers one after another.

Algorithm 5 First iteration

```
1: procedure FIRST_ITERATION(n, num_reps, set_sizes)
2:     for i = 2,...,n do
3:         for num_reps do
4:             cert_sets_1 ← GENERATE_RANDOM_CERT_SETS(set_sizes)
5:             start_cascade, cert_sets_i ← GENERATE_LOWER_LAYERS([p_1^1, ..., p_{i-1}^1], cert_sets_1)
6:             for fpp_candidate do
7:                 fpp_set ← [fpp_candidate, ..., fpp_candidate]
8:                 data[fpp_candidate] ← CONTINUE_CREATION(fpp_set, start_cascade, cert_sets_i)
9:             end for
10:        end for
11:        p_i^1 ← OPTIMIZATION_FUNCTION(data)
12:    end for
13:    return {p_2^1, ..., p_n^1}
```

Figure 5.1: Pseudocode for the first iteration of our framework. The **Input** parameters are n:= the number of layers to search, *num_reps*:=the number of simulation runs to conduct for the selection of p_i^1 and *set_sizes*:= the number of valid and revoked certificates to use. The **Output** is the set of n false positive probabilities that resulted in the minimum value of the optimization function.

j-th iteration: We will now use the notation p_i^j for the false positive probability we selected for layer i during the j-th iteration. When searching for p_i^j we will create the first $(i-1)$ layers of a cascade with the error probabilities we selected for those layers during the current iteration.

$$p_l = p_l^j \quad for \quad l = 2, ..., (i-1)$$

We will then continue the creation of the cascade with $p_i = fpp_{candidate}$ for every $fpp_{candidate}$ we want to inclue in our search. The difference to the first iteration is that we will use the false positive probabilities we chose during the preceeding iteration for all the higher layers

$$p_h = p_h^{j-1} \quad for \quad h = (i+1), ..., n$$

Algorithm 6 j-th iteration

```
1: procedure REITERATION(n, {p_2^{j-1}, ..., p_n^{j-1}}, num_reps, set_sizes)
2:     for i = 2,...,n do
3:         for num_reps do
4:             cert_sets_1 ← GENERATE_RANDOM_CERT_SETS(set_sizes)
5:             start_cascade, cert_set_i ← GENERATE_LOWER_LAYERS([p_1^j, ..., p_{i-1}^j], cert_sets)
6:             for fpp_candidate do
7:                 fpp_set ← [fpp_candidate, p_2^{j-1}, ..., p_n^{j-1}]
8:                 data[fpp_candidate] ← CONTINUE_CREATION(fpp_set, start_cascade, cert_set_i)
9:             end for
10:        end for
11:        p_i^j ← OPTIMIZATION_FUNCTION(data)
12:    end for
13:    return {p_2^j, ..., p_n^j}
```

Figure 5.2: Pseudocode for the subsequent iterations of our framework. The **Input** parameters are n:= the number of layers to search, $\{p_2^{j-1}, ..., p_n^{j-1}\}$:= the n false positive probabilities that were selected in the previous iteration, *num_reps*:=the number of simulation runs to conduct for the selection of p_i^j and *set_sizes*:= the number of valid and revoked certificates to use. The **Output** is the set of n false positive probabilities that resulted in the minimium value of the optimization function.

Performance evaluation: After each iteration we will evaluate the performance of the selected false positive probabilities through a slightly adaped version of the optimization function. We do so by creating a number of cascades with the newly selected and the "original" false positive probabilities suggested by James Larisch *et al*. For this comparison we will use the full sized data set, meaning we will use 5.200.000 revoked certificates and 156.000.000 valid certificates. We measure the number of hashes used, the creation time and the size of the created cascades. We then proceed to average the obtained measurements and calculate the percentage changes of the cascades with the selected false positive probabilities in comparison with the "original" false positive probabilities.

$$percent\_property_i(\{p_2^j, ..., p_n^j\}) = \frac{avg\_property_i[\{0.5, ...\}] - avg\_property_i[\{p_2^j, ..., p_n^j\}]}{avg\_property_i[\{0.5, ...\}]} \tag{5.3}$$

We then summarize the performance of our newly selected false positive probabilities in one value by calculating the weighted sum of the percentage changes.

$$perf(\{p_2^j, ..., p_n^j\}) = \sum w_i * percent\_property_i(\{p_2^j, ..., p_n^j\})$$

For this calculation one should use the same weights as in Formula 5.2 of the optimization function for the corresponding properties. We therefore use $w_i = 1$ for the size, the creation time and the number of hashes used during the creation. If the performance of the j-th iteration did not improve in comparison with the $(j-1)$-th iteration, we stop the reiteration process. We then return the set of false positive probabilities of the $(j-1)$-th iteration and recommend them for creation of the Bloom filter cascade underlying CRLite. The whole process of our *Layer_by_Layer* search can be implemented through the pseudocode in Figure 5.3.

Algorithm 7 Iterative parameter selection framework

1: **procedure** Layer_by_Layer_search(n, num_reps, set_sizes)
2: $\{p_2^1, ..., p_n^1\} \leftarrow$ First_iteration(n, num_reps, set_sizes)
3: $j \leftarrow 1$
4: **repeat**
5: $j \leftarrow j + 1$
6: $\{p_2^j, ..., p_n^j\} \leftarrow$ Reiteration($n, \{p_2^{j-1}, ..., p_n^{j-1}\}, num\_reps, set\_sizes$)
7: **until** $perf(\{p_2^j, ..., p_n^j\}) > perf(\{p_2^{j-1}, ..., p_n^{j-1}\})$
8: **return** $\{p_2^{j-1}, ..., p_n^{j-1}\}$

Figure 5.3: Pseudocode for the parameter selection framework. The **Input** parameters are n:= the number of layers to search, *num_reps* := the number of simulation runs to conduct for the selection of the false positive probability of one layer and *set_sizes*:= the number of valid and revoked certificates to use. The **Output** is the set of n false positive probabilities that had the best performance evaluation.

5.3 Implementation details

To utilize the resources available to us better we performed some optimizations to the code above. Those optimizations will be described in the following paragraphs.

Reduce read and write operations: When we search for p_i we observe that the *include_set* and the *exclude_set* for layer i (denoted $cert\_sets_i$ in Figure 5.1 and 5.2) are the same for all false positive probabilities we want to test. If we continue the creation of the cascade from layer i for every false positive probability in series, we have to repeatedly load those two sets into memory for every cascade we create. This effect remains for all certificates that we use on a certain layer in more than one cascade. To reduce the needed memory operations we want to create the layers of all cascades in parallel such that we have to load each certificate only once for every layer. We will maintain two 2-dimensional binary reference arrays to keep track of which certificates are still in the *include_set* / *exclude_set* of every cascade. At every layer we go through the *include_set* once and use the reference array to add the right elements to each cascade. We then go through the *exclude_set* once and use the second reference array to search for false positives. If a certificate is included in the *exclude_set* of a cascade but does not trigger a false positive, we set the corresponding entry in the reference array to $false$. We thereby updated the reference array from the exclude set to a reference array of the false positives. Since the *include_set* of layer i is the *exclude_set* of layer $(i+1)$ and the false positives of layer i are the *include_set* of layer $(i+1)$ we can now swap both sets and the corresponding reference arrays and continue on the next layer in the same manner.

To avoid continuously loading certificates that are no longer included in any cascade we will maintain a counter that indicates how many certificates of the sets are still in use. Whenever this counter reaches a certain threshold we delete the certificates that are no longer used from the set and update the corresponding reference array.

Use multiple intervals: For our experiments of the *Layer_by_Layer* search introduced in Section 5.2 we decided to test the false positive probabilities in the interval [0.2, 0.7] with a stepsize of 0.0002. This would require us to test 2500 false positive values for every p_i^j we want to select. Creating this many cascades in parallel would not only take too long for the time frame of this thesis but also take 17GB of memory to hold the cascades and 3GB to hold the reference arrays. We therefore decided to search this interval using 3 consecutive runs for the selection of every p_i^j. During the first run we will search the interval [0.2, 0.7] with a stepsize of 0.02. We will then analyze the data by hand and pick an adequate interval to search during the second run with a stepsize of 0.002. We will then pick an interval that contains the minimum of our optimization function to search during the third and final run with the desired stepsize of 0.0002. After the third run is completed we will set p_i^j to the false positive probability that resulted in the minimum of our optimization function.

Further implementation details The creation time is highly dependent on whether the *include_sets* and *exclude_sets* are stored on the disk or in main memory and on the underlying hardware that is used. For our implementation we decided to store them on the disk for the first few layers to reduce the needed main memory. Therefore the majority of the creation time of our implementation is used for the read and write operations of the certificates. To obtain more general results in terms of the creation time, we decided to exclude the loading times of the certificates in our creation time measurements. Our creation time measurements therefore consider the time it takes to add a new filter to a cascade, the time it takes to insert all elements in the *include_set* and the time it takes to search for false positives.

5.4 Final results

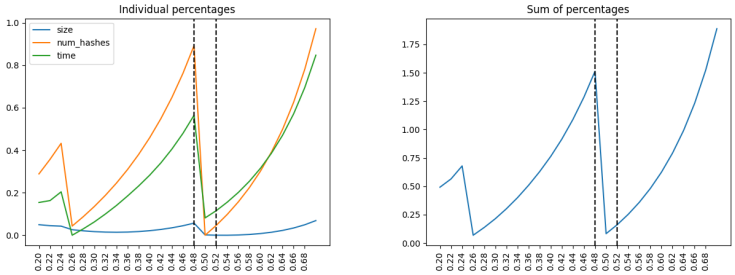
5.4.1 Experimental setup

All experiments in thesis thesis were carried out using Euler, a high-performance cluster service administered by the ETH Zürich. The implementation of the models presented were written in Python. For the Bloom filter cascade we used the implementation of J.C.Jones and Mark Goodwin [14] and utilized the structs and functions from the moz_crlite_lib provided on the Github repository of Mozilla [17].

5.4.2 First Iteration

With the process descibed in the Section 5.2 and the optimizations described in Section 5.3, we did the first iteration for 27 layers. We chose to not include the 28-th layer in our search to avoid the possibility of overfitting to the last layer. If one wanted to search for the most adequate probabilities for more than 27 layers, one simply has to increase the size of the set of revoked certificates. Presenting the results for all layers would take too much space and we therefore chose to show some representative data that suffices to show the development of the data throughout the first iteration and the choices we made. We chose to show the data we obtained for p_2^1 and p_{25}^1.

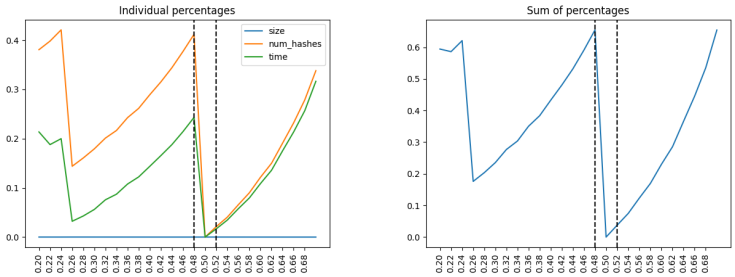
First run:



(a) Individual percentage changes

(b) Value of the optimization function

Figure 5.4: First iteration, Second layer, First simulation run



(a) Individual percentage changes

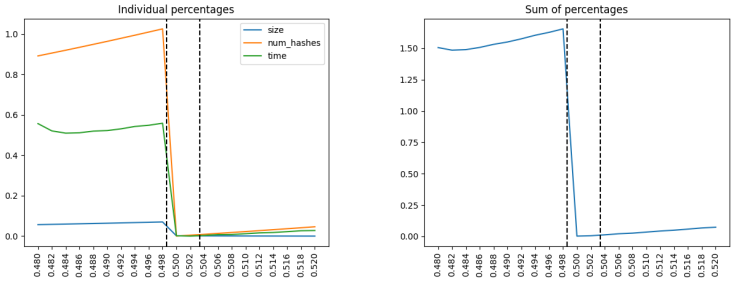
(b) Value of the optimization function

Figure 5.5: First iteration, 25-th layer, First simulation run

The Subfigures 5.4 a) and 5.5 a) show the percentage changes compared to the minimal obtained value for the size, the number of hashes and the creation time. We can clearly see that the influence of the false positive rate on the size decreases with increasing layers. This is due to the fact that the first few layers of a cascasde contribute the most to the the size of the full cascade. Meanwhile the creation time and the number of hashes can still be greatly influenced even in the last few layers. The Figures 5.4 b) and 5.5 b) show us the value of the optimization function introduced in Section 5.2. In Figure 5.4 b) we can find 2 false positive probabilities that would be suitable for further investigation and 5.5 b) shows that for the higher layers there seems to be only one false positive probability of interest. For the layers $2 - 8$ the optimization function reached its minimum at 0.26. We nevertheless chose to further investigate the interval around 0.5.

This choice was made because only one property (creation time) reached it's global minimum at 0.26 but the other two properties reached their global minimum at 0.5.

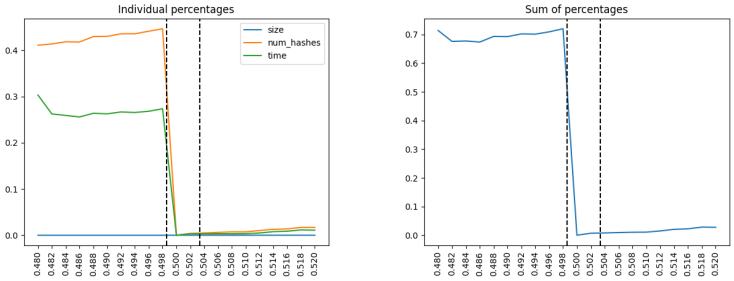
Second run:



(a) Individual percentage changes

(b) Value of the optimization function

Figure 5.6: First iteration, Second layer, Second simulation run



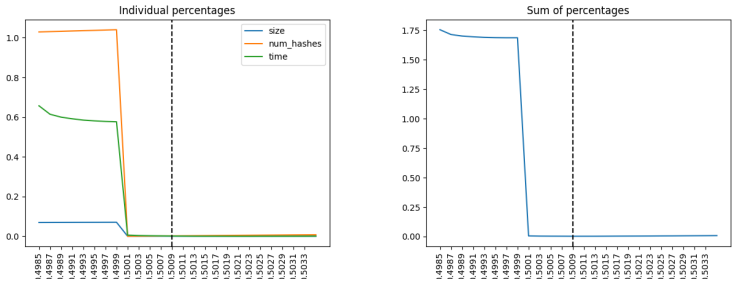
(a) Individual percentage changes

(b) Value of the optimization function

Figure 5.7: First iteration, 25-th layer, Second simulation run

Figure 5.6 b) and 5.7 b) show us the value of the optimization function of the second run for layer 2 and layer 25. Throughout the first iteration the minimum of the optimization function was always between 0.5 and 0.502. We therefore always chose the interval [0.4986, 0.5034] for our third and final run.

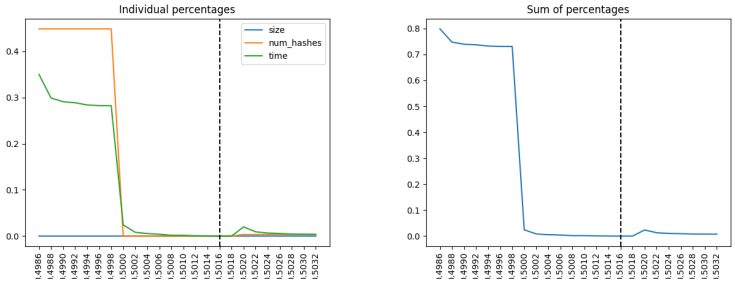
Third run:



(a) Individual percentage changes

(b) Value of the optimization function

Figure 5.8: First iteration, Second layer, Third simulation run



(a) Individual percentage changes

(b) Value of the optimization function

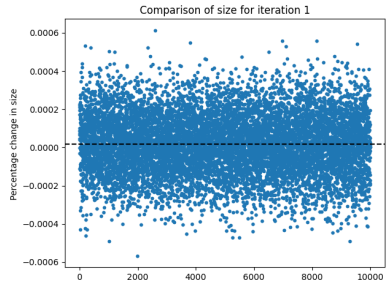
Figure 5.9: First iteration, 25-th layer, Third simulation run

After the third run had finished for a layer i, we set p_i^1 to the false positive probability that resulted in the smallest value of our optimization function. For p_2 we selected 0.5009 and for p_{25} we selected 0.5016. The false positive probabilities we selected for all layers during the first iteration are summarized in Table 5.1.

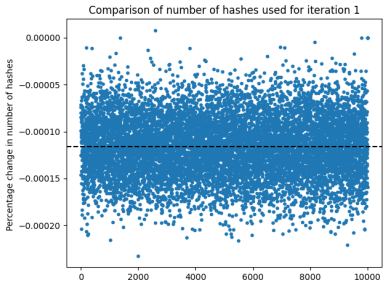
p_2^1	p_3^1	p_4^1	p_5^1	p_6^1	p_7^1	p_8^1	p_9^1	p_{10}^1
0.5009	0.501	0.5008	0.501	0.5006	0.5002	0.5002	0.5002	0.5002
p_{11}^1	p_{12}^1	p_{13}^1	p_{14}^1	p_{15}^1	p_{16}^1	p_{17}^1	p_{18}^1	p_{19}^1
0.5004	0.5002	0.5006	0.5008	0.5002	0.5012	0.5004	0.501	0.5002
p_{20}^1	p_{21}^1	p_{22}^1	p_{23}^1	p_{24}^1	p_{25}^1	p_{26}^1	p_{27}^1	
0.501	0.5004	0.5012	0.501	0.5016	0.5016	0.5016	0.503	

Table 5.1: Selected false positive probabilities during the first iteration.

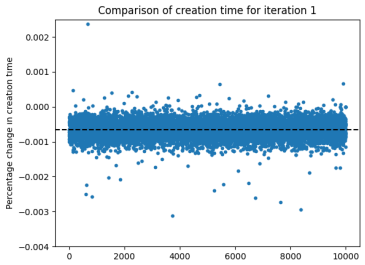
Performance evaluation:



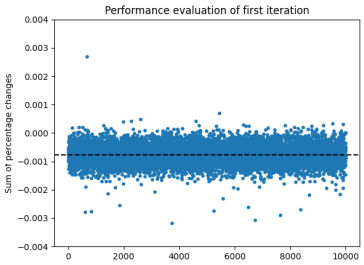
(a) Percentage change in size



(b) Percentage change in number of hashes



(c) Percentage change in creation time



(d) Value of the adapted optimization function

Figure 5.10: Summary of the percentage changes in size, number of hashes used, creation time and the value of the adapted optimization function of our selected false positive probabilities in comparison with the false positive probabilities suggested by James Larisch *et al*.

After we completed the first iteration we did 10000 performance evaluations for the false positive rates we chose during the first iteration as described in Section 5.2.

Each dot in the Figures 5.10 a), b), c) represents the percentage change in size, number of hashes used and the creation time of our choice of false positive probabilities in comparison with the false positive probabilities suggested by James Larisch *et al*. One can see that our choice of false positive probabilities improved the number of hashes used for creation by 0.01158% on average and the creation time by 0.06626%. We increased the size of the cascade on average by 0.001826% and therefore improved the value of the optimization function by 0.0761%

5.4.3 Subsequent Iterations

For the reiteration process we used the same steps and reasoning for selecting the intervals and final false positive probabilities as we described for the first iteration. When we search for the best p_i^j during the j-th iteration we use the $p_2^j, ..., p_{i-1}^j$ we chose during the current iteration to create the first $i-1$ layers. We then continue the creation with $p_i = fpp_{candidate}$ for every $fpp_{candidate}$ we want to include in our search. The only difference to the first iteration is that we don't use p_i for all the higher layers. Instead we use the values we chose during the preceeding iteration.

$p_h = p_h^{j-1}$ for $h = i+1, i+2, ...$

The Tables 5.2 and 5.3 show the false positive probabilities we selected during the subsequent iterations.

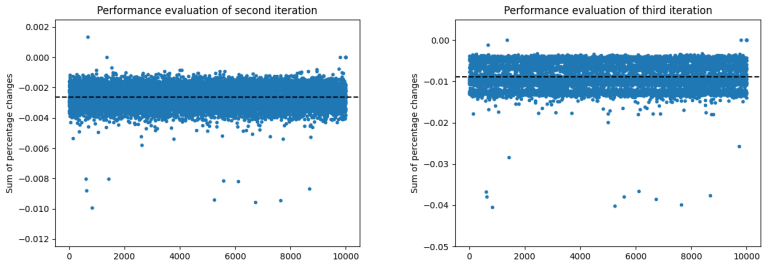
p_2^2	p_3^2	p_4^2	p_5^2	p_6^2	p_7^2	p_8^2	p_9^2	p_{10}^2
0.502	0.5028	0.5028	0.5016	0.5016	0.5022	0.503	0.5028	0.5028
p_{11}^2	p_{12}^2	p_{13}^2	p_{14}^2	p_{15}^2	p_{16}^2	p_{17}^2	p_{18}^2	p_{19}^2
0.503	0.5014	0.501	0.5008	0.5014	0.5012	0.5002	0.5014	0.5002
p_{20}^2	p_{21}^2	p_{22}^2	p_{23}^2	p_{24}^2	p_{25}^2	p_{26}^2	p_{27}^2	
0.503	0.5014	0.5008	0.5008	0.5016	0.5016	0.5034	0.509	

Table 5.2: Selected false positive probabilities during the second iteration.

p_2^3	p_3^3	p_4^3	p_5^3	p_6^3	p_7^3	p_8^3	p_9^3	p_{10}^3
0.503	0.5022	0.5016	0.5022	0.5022	0.5016	0.5024	0.5008	0.501
p_{11}^3	p_{12}^3	p_{13}^3	p_{14}^3	p_{15}^3	p_{16}^3	p_{17}^3	p_{18}^3	p_{19}^3
0.5012	0.5008	0.5014	0.5008	0.5014	0.501	0.5014	0.5008	0.5014
p_{20}^3	p_{21}^3	p_{22}^3	p_{23}^3	p_{24}^3	p_{25}^3	p_{26}^3	p_{27}^3	
0.5018	0.5004	0.5012	0.501	0.5014	0.5036	0.5034	0.5018	

Table 5.3: Selected false positive probabilities during the third iteration.

Performance evaluation of subsequent iterations:



(a) Second iteration **(b)** Third iteration

Figure 5.11: Value of the adapted optimization function for the second and third iteration.

	Size	Hashes	Time	perf()
2nd itr	0.0048827	−0.0332697	−0.2351988	−0.2635858
3rd itr	−0.0021632	−0.0335049	−0.8473088	−0.8829769

Table 5.4: Percentage changes and the value of the performance evaluation for the second and third iteration in comparison to the false positive probabilities suggested by James Larisch *et al*. Values are given in percent.

Table 5.4 shows that the first three iterations of our model resulted in lower values of the performance evaluation in comparison with the preceeding iteration. The third iteration had improved all three considered properties in comparison with the false positive probabilities suggested by James Larisch *et al*. The false positive probabilities we selected in the fourth iteration did not perform better than the set we chose during the third iteration and we

therefore stopped the reiteration process.

Discussion of results

During the first iteration of our model we selected a set of false positive probabilities that were all higher than the false positive probabilities suggested by James Larisch *et al* and are currently used by Mozilla. In our performance evaluation the set we thereby selected had decreased the creation time and the number of hashes used during creation but increased the size. We observed that this iteration selected the lowest false positive probabilities of all iterations for the layers below the 14-th.

The second iteration increased all of the selected false positive probabilities up to the 14-th layer. For the remaining layers (14-27) this iteration increased 6 of the selected false positive probabilities, kept 5 at the same value and decreased 3 without a clear pattern. The set we thereby selected further reduced the creation time and the number of hashes used during creation but it had also further increased the size of the resulting cascade in comparison with the first iteration.

The third iteration incremented 9 of the false positive probabilities selected during the second iteration, decremented 14 and kept 3 at the same value. There was no noticeable pattern in the changes of the selection. The set we thereby selected further decreased the creation time and the number of hashes used in comparison with the second iteration. Most interestingly this set of false postive probabilities had additionally decreased the average size of the resulting cascade in comparison with the false positive probabilities suggested by Larisch *et al*.

After the first simulation run of the first iteration we had found two false positive probabilities that resulted in a local minima of our optimization function. These values were 0.26 and 0.5. We assume that the minima occured around these two values because $log_2(1/p)$ is here the closest to an interger value. The more the selected values deviate from an interger value the more rounding is involved in setting the number of hash functions and the number of bits used in every Bloom filter in the cascade. The attentive reader might observe that similar holds for the value 0.125 which was not presented in this thesis. Before we performed our experiments we conducted three iterations of our model on a reduced test set. The data we obtained showed that there was indeed a third local minimum in the considered properties around the value 0.125. We chose to omit this region in our final test since none of the considered properties had a global minimum and we therefore concluded that this value is not a viable candidate for optimizing the performance of the Bloom filter cascade in any way.

Chapter 6

Conclusion

At the beginning of this thesis we presented an efficient fail-closed method for verifying the revocation status of certificates called CRLite. It was developed by James Larisch *et al* and is currently deployed by Mozilla. When implementing this method one has to select an adequate hash function and a set of false positive probabilities for the creation of the underlying Bloom filter cascade. The choice of those parameters and the hash function has great impact on the overheads introduced and the reliability of the system. In this thesis we presented formal methods to select adequate parameters in order to improve the performance of CRLite.

Hash function selection In Chapter 4 we have introduced a method to evaluate the performance of hash functions. This method quantifies two properties of hash functions when using them for the construction of the Bloom filter cascade underlying CRLite.

We conducted 1000 test runs to compare 8 cryptographic hash functions. Mozilla chose to use the non-cryptographic Murmurhash3 in their implementation of CRLite. This choice was made because of its speed and their back-up plan is to use the cryptographic hash function `SHA_256`. The results we obtained supports the choice of their back-up hash function since it was the fastest one in our test suite. But our model also considers the uniformity of hash functions via the size of the resulting cascade. The unified results we obtained lead us to recommend the use of `SHA3_256` over `SHA_256` for the use in CRLite. `SHA3_256` was the third fastest hash function in our test suite and we come to the conclusion that it is more suitable to replace Murmerhash3 because it had the best and most consistent performance in terms of the size of the resulting cascade.

Parameter selection In Chapter 5 of this thesis we introduced an iterative parameter selection framework. The framework can be used to select a set

of false postive probabilities for the construction of the Bloom filter cascade underlying CRLite. Previous work selected the false positive probabilities only under consideration of the size of the resulting cascade. The purpose of our framework is to select false positive probabilities that improve the performance of the Bloom filter cascade, not only in terms of size but also under consideration of the creation time and the number of hashes used during creation. The optimization function we use to select adequate false positive probabilities treated the creation time, the number of hash functions used during creation and the size of the resulting cascade as equally important. The function can be adjusted towards different preferences towards those three properties and also to take additional properties into account.

In this thesis we have applied the presented framework to the Bloom filter cascade underlying CRLite. We thereby selected a set of 27 false positive probabilities and conducted 10000 performance evaluations of this false positive set. The results we obtained showed that the parameters we chose on average improved all three properties we focused on when comparing with the parameters suggested by James Larisch *et al* and currently used by Mozilla. This result proves that the framework we presented is a valid method to further reduce the overheads introduced by the use of CRLite for certificate revocation verification.

We suggest the following areas for further reasearch:

1) How does adjusting the optimization function towards different preferences change the false positive probabilities that the presented framework selects?

2) Could the properties of the cascade be improved by using different rounding techniques for the calculation of the size and the number of hash functions used for every Bloom filter?

3) Can the parameter selection framework we presented also be used to optimize the parameters of modified data structures e.g. cascaded cuckoo filers [19]?

Bibliography

[1] Blake2 - fast secure hashing. https://www.blake2.net/.

[2] Elena Andreeva, Bart Mennink, Bart Preneel, and Marjan Škrobot. Security Analysis and Comparison of the SHA-3 Finalists BLAKE, Grøstl, JH, Keccak, and Skein. In *AFRICACRYPT*, 2012.

[3] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche. Keccak. https://keccak.team/keccak.html.

[4] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A Fast Short-Input PRF. In *INDOCRYPT*, 2012.

[5] Jean-Philippe Aumasson, Willi Meier, Raphael Phan, and Luca Henzen. *The Hash Function BLAKE*. Springer Publishing Company, Incorporated, 2014.

[6] Eli Biham, Rafi Chen, and Antoine Joux. Cryptanalysis of SHA-0 and Reduced SHA-1. *Journal of Cryptology*, 28:110–160, 2014.

[7] Eli Biham and Orr Dunkelman. A Framework for Iterative Hash Functions - HAIFA. Cryptology ePrint Archive, Report 2007/278, 2007. https://ia.cr/2007/278.

[8] Burton H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.

[9] Ivan Bjerre Damgård. A Design Principle for Hash Functions. In Gilles Brassard, editor, *CRYPTO*, 1990.

[10] Quynh Dang. Recommendation for Applications Using Approved Hash Algorithms, 2012. https://csrc.nist.gov/publications/detail/sp/800-107/rev-1/final.

[11] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, Benne de Weger. MD5 considered harmful today. *https://www.win.tue.nl/hashclash/rogue-ca/*, 2008.

[12] Morris Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2015. https://csrc.nist.gov/publications/detail/fips/202/final.

[13] Geoff Huston. Revocation: is there a better way to secure certificates. https://blog.apnic.net/2020/03/16/revocation-is-there-a-better-way-to-secure-certificates/.

[14] J.C.Jones and Mark Goodwin. mozilla/filter-cascade. https://github.com/mozilla/filter-cascade.

[15] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, Ronny Van Keer. The sponge and duplex constructions. https://keccak.team/sponge_duplex.html.

[16] James Larisch, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers. 2017.

[17] J.C.Jones, John Schanck, Patheldt, Dana Keeler, Luke Crouch, Sven Marnach, Mathieu Leplatre. mozilla/crlite. https://github.com/mozilla/crlite.

[18] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. An End-to-End Measurement of Certificate Revocation in the Web's PKI. In *Proceedings of the 2015 Internet Measurement Conference*, 2015.

[19] Sai Medury, Amani Altarawneh, and Anthony Skjellum. Design and Evaluation of Cascading Cuckoo Filters for Zero-False-Positive Membership Services. In *Computing and Communication Workshop and Conference (CCWC)*, 2021.

[20] Ralph C. Merkle. One Way Hash Functions and DES. In Gilles Brassard, editor, *CRYPTO*, 1990.

[21] M Mitzenmacher and Eli Upfal. *Probability and Computing*. 2005.

[22] National Institute of Standards and Technology. Secure Hash Standard, 1993. https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/NIST.FIPS.180.pdf.

[23] National Institute of Standards and Technology. Secure Hash Standard, 1995. https://csrc.nist.gov/publications/detail/fips/180/1/archive/1995-04-17.

[24] National Institute of Standards and Technology. Secure Hash Standard, 2002. https://csrc.nist.gov/publications/detail/fips/180/2/archive/2002-08-01.

[25] Vadims Podāns. ADCS certificate serial number generation algorithms – a comprehensive guide. *https://www.pkisolutions.com/adcs-certificate-serial-number-generation-algorithms-a-comrehensive-guide/*, 2020.

[26] Eric Rescorla. HTTP Over TLS. RFC 2818, 2000. https://www.rfc-editor.org/info/rfc2818.

[27] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018. https://www.rfc-editor.org/info/rfc8446.

[28] Harshvardhan Tiwari. Merkle-Damgård Construction Method and Alternatives: A Review. *Journal of Information and Organizational Sciences*, 41:283–304, 2017.

[29] Benjamin VanderSloot, Johanna Amann, Matthew Bernhard, Zakir Durumeric, Michael Bailey, and J. Alex Halderman. Towards a Complete View of the Certificate Ecosystem. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, page 543–549, 2016.

[30] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. CRYPTO, 2005.