

Replay Attack Detection Using Bloom Filters

Master thesis by Lukas Werthmanns
Date of submission: September 4, 2024

1. Review: Prof. Dr. Björn Scheuermann
2. Review: Jan Götte
Darmstadt



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Electrical Engineering and
Information Technology
Department

Fachbereich Informatik
(Zweitmitgliedschaft)

Fachgebiet
Kommunikationsnetze
(KOM)

Lukas Jakob Werthmanns

Course of study: M.Sc. Informatik

Master Thesis

Topic: Replay Attack Detection Using Bloom Filters

Submitted: 4.11.2024

Betreuer: Jan Götte

Prof. Dr. Björn Scheuermann

Fachbereich Kommunikationsnetze

Rundeturmstr. 10

64283 Darmstadt

Veröffentlicht unter CC-BY 4.0 International

<https://creativecommons.org/licenses/by/4.0>

Contents

1	Introduction	6
1.1	Overview	6
1.2	VPN	6
1.3	The PSP Protocol	7
1.4	Replay Attacks	9
1.4.1	Countermeasures	10
1.4.2	PrePlay Attacks	11
1.5	Duplicate Detection	12
1.6	Standard Bloom Filter	13
1.6.1	History	14
1.6.2	Mechanisms	15
1.6.3	Mathematic evaluation	16
1.7	Bloom Filter Variation	18
1.7.1	Scalable Bloom Filter	18
1.7.2	Counting Bloom Filter	18
1.7.3	Rolling/Aging Bloom Filter	19
1.7.4	Stable Bloom Filter	20
2	Methods	24
2.1	Problem Description	24
2.2	Evaluation Methods	25
2.3	Environmental Parameters	25
2.3.1	Scenario 1	25
2.3.2	Scenario 2	27
2.3.3	Scenario 3	28
2.3.4	Acceptable False Positive Rate (α)	28
2.4	Bloom Filter Selection	30
3	Findings	31
3.1	Bloom Filter Evaluation	31
3.1.1	Bloom Filter Mathematical Evaluation	31
3.2	Standard Bloom Filter - Simulation	32
3.3	Stable Bloom Filter Evaluation	37
3.3.1	False Positives	38
3.3.2	False Negatives	41



3.3.3 Sliding window cutoff 43

4 Discussion 45

5 Conclusion 47

Abstract

Even with widespread encryption and traffic signing in modern networks, certain challenges persist. Replay attacks, in particular, pose significant risks by causing unintended consequences in application protocols, and can be very difficult to detect without modifying the underlying protocols. Relying solely on the application layer to identify duplicate messages may be insufficient. This work explored the use of Bloom filters as a potential space- and time-efficient solution for detecting duplicate messages in network traffic. However, after a thorough analysis of the key aspects and parameters of various Bloom filter implementations, it was determined that this approach is neither practical nor scalable within the memory constraints of current networking hardware.

1 Introduction

In our landscape of computer networks, we rely heavily on the efficiency and security of large scale data-centers. Among the ongoing challenge of balancing those two often contradicting goals in a field of ever-changing requirements, Google released its PSP Protocol (short PSP) to ensure encryption in transit. In their design for scalability, they focus on hardware implementability and minimal memory requirements through a highly reduced per-connection state. The design is, however, distinctly lacking replay detection, leaving its implementation to the respective application. This work addresses whether the Bloom-Filter with its highly space-efficient membership testing capability and hardware implementability could be used to add support for duplicate detection.

1.1 Overview

The following section will introduce the various concepts relevant to the problem of replay detection in the PSP protocol. Besides introducing the PSP protocol, an overview of replay attacks will show their inner workings and the real-world implications a successful replay attack can have. The reader should understand all technologies involved, the dangers of replay attacks, and the problems associated with attempts to prevent them globally in network traffic. At last, the Bloom filter, both in its standard version as well as its relevant derivatives, will be explained in order to understand their use cases, respective qualities, and trade-offs that will help in understanding the further attempts of implementation and evaluation using them in packet filter for duplicate detection.

1.2 VPN

A Virtual Private Network (VPN) is a network architecture designed to extend a private network across one or more other networks that can be untrusted or less secure. By doing so, VPNs allow for secure communication over networks, like the public internet, that are not under the control of participants of the VPN. VPNs are often used between organizations and individuals who require secure access to private network resources from remote locations but are also used between data centers that require a secure way of communication. This technology is able to provide a layer of abstraction and isolate the private network from the underlying network. They do so by using tunneling protocols that work by encapsulating existing packets that should not be sent directly over

the underlying network. These tunneling protocols are able to provide features like encryptions and traffic signing between the communication parties. [23]

Their important role in IT Security also makes VPN an attractive target for criminals. If a malicious actor can get access to the private network they are able to access resources that are not reachable over the underlying network directly. As VPN are often used between companies and individuals in a work from home scenario a hijacked session could grant the attacker access to company resources and a potential entrydoor for gaining further access inside the company network. This could be carried out in form of a Man-in-the-middle attack that exploits weaknesses in a key exchange algorithm that does not provide a way for the parties to verify each others identity. Other attacks might include DNS Hijacking by redirecting traffic through a malicious server or brute force attacks to find weak credentials. [6]

1.3 The PSP Protocol

Cryptographic encryption is one of the fundamental techniques used to secure data by converting it into a format that can only be read by those with the proper decryption key. This ensures that data remains unintelligible to unauthorized parties even if the communication takes place over an otherwise open channel. In modern large-scale data centers, encryption is critical to ensuring the security and privacy of transmitted information. This type of encryption protects data as it travels between different physical locations to prevent unauthorized access or tampering with the encrypted data. A critical requirement for it to work is that the surrounding protocol ensures a secure exchange of keys, as well as both communication partners, knowing they are communicating with each other and not a third party posing as a man-in-the-middle [38] PSP does not dictate a key exchange algorithm itself. [25] For the rest of this work, we assume that a secure key exchange has been performed and that no man-in-the-middle attack is possible.

The PSP protocol has been part of Google's decade-long effort to grant encryption in transit for almost all of their connections. [21] [5] The strict isolation requirements between their connections introduce the necessity to encrypt each connection separately across their millions of connections. These enormous demands at Google's scale come at a significant computing overhead of 0.7% of Google's entire processing power which lead to the development of PSP. [5] The PSP Protocol was designed to meet the requirements of large-scale data centers by off-loading cryptographic tasks to the Network Interface Cards (NIC) hardware. The goal of developing PSP was to encapsulate and encrypt network traffic sent via existing and established networking protocols like TCP [56] and UDP [57]. This makes it possible to send previously unencrypted traffic as the payload of a PSP packet that can guarantee the contents to be encrypted and authenticated between communication partners. Throughout PSP development, a huge focus was put on keeping the state saved in hardware to an absolute minimum, as storing even relatively small amounts of encryption data for millions of connections will pose a significant demand for memory. These qualities differ from already established solutions for traffic encryption, such as TLS, which is tightly coupled to TCP and is, therefore, not able to work independently of the underlying protocol. Another possible solution, the protocol suite IPsec [49], is able to work independently of the underlying protocol but requires

storing the entire encryption state in hardware that does not scale well enough to support the millions of connections required by Google. [5]

PSP has been designed using the concept of Security Associations (SA) from IPsec, which represents a set of unidirectional traffic. Packets from one SA are always encrypted and authenticated using the same cryptographic parameters, which are either stored in hardware on the Network Interface Card (NIC) or derived from the packet itself when using stateless mode.

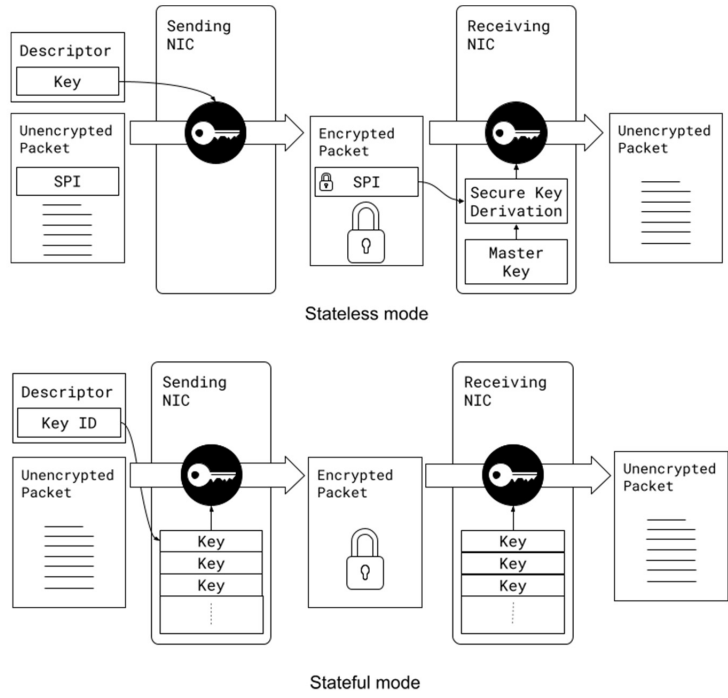


Figure 1.1: PSP stateful and stateless mode [14]

All PSP Traffic consists of UDP-encapsulated PSP packets with a custom header and trailer. The crypt offset parameter defines which part of the packet should be encrypted using a key that must be unique for the respective SA. The encryption key is derived using the currently active master key, while the last active key is still kept for the decryption of existing connections until a set rotation period ends. This period marks the end of the SA lifetime and is typically set to 24 hours. After that period, the previously active key will become inactive and will only be used to decrypt older incoming packets. At the end of the second period, the key will be discarded and replaced by the previously active key. This is also called a double rotation.

Relying on hardware implementation for performance and security reasons, the NIC is responsible for generating the Security Parameter Index (SPI), which is an index tag that is also used in SA key derivation. Additionally, the NIC is required to implement a 64-bit timestamp counter that increments in 1-picosecond units. The 32-bit SPI and the 64-bit timestamp counter are combined into the 96-bit Initialization Vector (IV), which must be strictly increased with a 64-bit wraparound.

PSP supports 4 Operational modes using the cryptographic algorithms AES-GCM [48] with encryption and authentication and AES-GMAC [29] with authentication only, both in 128- and 256-bit modes.

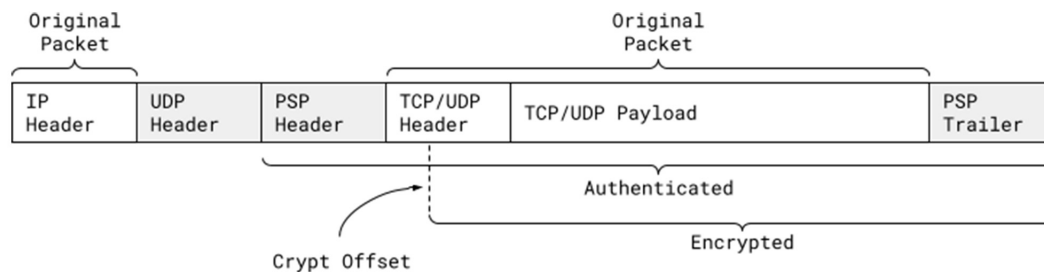


Figure 1.2: PSP Packet encapsulation [13]

The Architecture specification states that PSP currently has no support for duplicate detection. With its aim to keep the stored state to a minimum, the PSP protocol design makes this trade-off that adds simplicity to the protocol design and frees up memory that would have been used to store the required information about previously encountered packets. One major downside of having no support for duplicate detection is that PSP does not support the detection of replay attacks. [5]

1.4 Replay Attacks

After establishing some fundamental knowledge about the PSP protocol and its goals in terms of encryption and performance, we also understand why PSP is vulnerable to replay Attacks. We can go a step further and look at how replay attacks work and what the risks being vulnerable against them might introduce. Replay attacks are cryptoanalytic attacks on communication protocols. [20]

To better illustrate the replay attack, we are going to use the placeholder names Alice, Bob, and Eve, first introduced in the article “aA method for obtaining digital signatures and public-key cryptosystems” by Rivest, Shamir, and Adleman. Alice and Bob are names used for two communication partners, A and B, with Alice trying to send a message to Bob. Eve, short for eavesdropper, is a malicious intruder trying to cause harm. [47] [50] Given that both communication parties, Alice and Bob, communicate exclusively through encrypted and authenticated traffic, their communication might be recorded by the malicious party, Eve. This could be achieved by Eve through recording vulnerable wireless traffic, having placed a recording device somewhere along the route their traffic

takes, tapping a wire physically, having access to a compromised router or simply by owning parts of the communication hardware in use. Eve has no possibility of reading or editing the network traffic thanks to encryption and authentication being in place but with no other methods of protection in place a replayed message from Eve would seem indistinguishable to Bob from the original packet, sent by Alice.

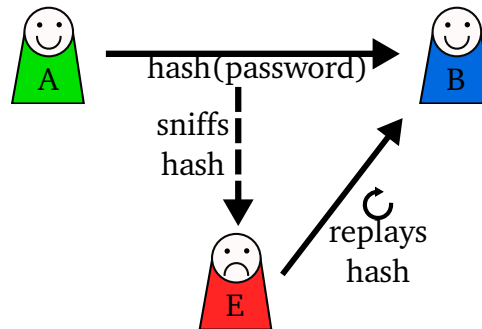


Figure 1.3: Replay attack Illustrated. Source: Wikimedia Commons, Author: László Körtvélyesi, License: CC BY-SA 4.0.

If the protocol in use receives the same packet twice, unwanted behavior on Bob's end could be triggered. In a simple example replayed messages could be used to re-use log in credentials or perform the same transaction multiple times or disrupting operations by sending shut down signals. Detailed knowledge about the protocol in use as well as the recorded traffic is required to be able to trigger specific actions. A much less elaborate form of attack might just use recorded traffic in DoS attacks by sending masses of authentic looking traffic with the goal of simply overwhelming network resources or intrusion detection systems. This form of attack poses the particular risk of being used as a false flag operation in order to frame Alice of sending the traffic herself. Detecting the true origin of the used traffic could be especially complicated when network resources are yielded, and log data can not be fully recovered.

1.4.1 Countermeasures

There are different ways to address the vulnerability against replay attacks, which are often similar in nature. For example, session IDs, also called session tokens, can be used. [37] In this case, Bob sends Alice a nonce, which Alice uses to compute the used Session Token to be included in future messages. Bob performs the same calculation, e.g., hashing the value, and can therefore compare Alices Session Tokens with his. He will only accept messages that include the token matching the currently used one. When Eve tries to replay captured traffic, her session will be different from Alices and cause a mismatch of session tokens, rendering her attack futile.

Another similar approach might include one-time passwords or nonces to be sent along with Bob's Message. In the latter case, he must also include a Message Authentication Code that Alice has to check.

The use of nonces in the different processes requires a random selection process, which is unpredictable to outside observers who otherwise might use predicted future values for successful attacks.

Another possibility is timestamping, in which Bob periodically broadcasts his system time. [22] Alice then tries to include her best estimate of Bob's time, which will only be accepted by Bob if it is in an acceptable tolerance range. This saves the need for (pseudo-) random number generation or time used to ask for a nonce to include. However, it is possible that in replay attacks that are very quickly carried out, the acceptance limit could still be met.

Replay attacks have been used successfully in the past to target different and important parts of IT infrastructure. The following examples aim to show their relevance and the need to protect against systems, ideally the pre-application layer.

The high-profile case of the Stuxnet worm uses replayed process input to avoid detection, among a plethora of other techniques. [43] It was used in 2009, targeting Iranian nuclear centrifuges in an attack of previously unknown sophistication. [40]

In older Kerberos Versions <5 replayed traffic could be used to obtain a Ticket Granting Service (TGS) ticket. [53]

For a more recent example of replay attacks being used, the cyber criminal threat group LAPSUS\$ used replayed session tokens to trigger multi-factor authentication simple-approval prompts, expecting a portion of legitimate users to grant access. [44] [42]

Implementing no method of detecting duplicates simplifies the PSP protocol significantly and saves a lot of resources that would have been allocated to achieve this goal. However, having no way of detecting duplicates forces the respective applications to deal with the possibility of replay attacks. This could be achieved by implementing their own way of duplicate detection at the application layer by using the techniques mentioned above, building the used protocol in a way that does not allow for such an attack, or simply accepting the risk. Leaving the decision to the applications down the line might introduce the risk of lacking replay protection being exploited, possibly leading to severe consequences. [34] In addition, the overhead of application layer protocol design and implementation can be significant and redundant depending on the respective application. Finally, the possibility of programmer error in implementing this security feature is increased with every application that has to add its own protection mechanism against replay attacks. Detecting replay attacks as early as possible would be ideal for applications, freeing application layer protocol design from the responsibility of including a mechanism for duplicate detection. Therefore, supporting duplicate detection and granting protection against replay attacks would be useful features for PSP and could be useful for every application using the protocol.

1.4.2 PrePlay Attacks

In addition to the replay attacks and their countermeasures mentioned above, a slight variation of the attack should be mentioned, which can lead to severe consequences. An attack form called a pre-play attack not only records the network traffic to be replayed but also keeps it from reaching

its destination entirely. [4] This attack requires more control over the flow of traffic but creates a situation where a packet seems to be lost in the network. Such a behaviour is expected in most computer networks and usually handled by either retransmitting the packet in protocols like TCP or ignoring the dropped packet for use-cases like video voice over IP using UDP. Without a method of invalidating the old packet, it could be used as a packet that looks indistinguishable from other novel packets arriving at their destination when finally sent out by the malicious party carrying out an attack. It has to be kept in mind that in cases of retransmissions, depending on the protocol, timestamps, checksums, and nonces being transmitted might not be the same as in the stolen packet. If that is the case, the methods mentioned above can not prevent such an attack as they assume that every replayed packet has been previously encountered.

Let us keep this kind of attack in mind for later and focus on the possibility of duplicate detection in networks to catch actual replay attacks.

1.5 Duplicate Detection

The field of duplicate detection has been an important area of study in data management, network monitoring, web crawling, and indexing. Various different techniques have been developed to deal with the goals of scalability and accuracy. A major portion of the effort of duplicate detection has stemmed from the desire to make databases more storage-efficient by keeping as little duplicate data as possible. This field is called deduplication and deals with the process of finding and eliminating duplicate portions of data in databases.

A notable difference between compression and data deduplication in comparison is that compression takes a Byte-level approach to find and eliminate redundancies while deduplication focuses on eliminating redundancies at the level of files or chunks of files, which is shown to scale much better for large-scale storage systems. Deduplication Techniques usually center around the use of hash functions in order to calculate fingerprints of files or chunks of files, the latter being either fixed in size or variably sized based on the respective data being handled. In a typical workflow for chunk-level deduplication, the fingerprinted chunks are then indexed for further storage management, operating with unique chunks to perform the various tasks needed to work with the data, including support for file deletion without losing unwanted data and garbage management. [66] Among the different techniques and tweaks used is content-aware deduplication, which takes knowledge about the data into concern. In that context, it is also possible to use specialized fingerprinting methods to give specific parts of data more weight in the comparison. For some databases, the problem of almost exact matches can be solved by using hash algorithms like SimHash that are able to produce the same hash value on only slightly different inputs. [51]

One major concern when dealing with very large amounts of fingerprints is the increased likelihood of hash collisions, described by the phenomenon called the birthday paradox. [8] In case the hash function output length, in combination with the number of fingerprints generated, will produce a likelihood for a hash collision resulting in data corruption being deemed to be high, other methods of

verification have to be put in place in order to point out possible differences in data. Those methods could consist of additional hashing or even byte-by-byte comparison granting definitive results.

One important aspect of deduplication is the desire to grant real-time or near real-time deduplication upon arrival of data newly added to the database, which is also used in the process of finding new data inside data streams. Hash Tables can provide an average and amortized lookup, insertion, and deletion time of $O(1)$. This is only the case as not many items are hashed into the same key. [16] If n elements are hashed into the same key, the worst-case complexity of $O(n)$ is reached, either reducing performance significantly or making an expensive rebalancing necessary that maps the existing keys into a larger memory area in order to drive the collision rate down but takes up further resources during the rebalancing process. Due to their relative memory efficiency and low time complexity, Hash tables can be used in database indexing, providing the ability to detect duplicates during the handling of hash collisions.

Another solution for speeding up the lookup time for key-value storage utilizes the data structure of a Bloom filter that will be described in much greater detail in the section below. The Bloom filter is tasked with answering set membership queries before the key-value storage is queried. Since the Bloom filter can return false positives, never false negatives, a negative answer from the Bloom filter can be returned directly since the database does not contain the queried element. Upon a positive answer from the Bloom filter, the database has to be queried in order to retrieve either the element in the case of a true positive or a negative answer in case the Bloom filter produced a false positive, which can happen with a small likelihood. The Bloom filter only takes up a limited amount of space and causes a negligible amount of overhead compared with the costly database lookups, as they can answer queries in constant lookup time. This approach of duplicate detection seems highly interesting for our case of detecting duplicates in network traffic. Not only are the fast lookup and insertion times for new elements in Bloom filters generally attractive properties, but Bloom filters are space efficient, produce only a low amount of false positives, and can be efficiently implemented in software and hardware. [18] [33] Another property that the Bloom filter supports by default is that we only need to consider exact matches since all traffic we are taking into account is encrypted using AES-GCM and signed with AES-GMAC therefore, changing the content of an existing message is not possible without breaking the cryptographic methods used and encountering the same non-replayed data twice is highly unlikely. [25]

From a first look, the Bloom filter seems like an interesting candidate for solving the problem of duplicate detection in network traffic. Going forward, we are going to take a closer look at the Bloom filter properties as well as the requirements our solution needs to fit.

1.6 Standard Bloom Filter

Bloom filters are a space-efficient probabilistic data structure first conceived by Bloom in 1970, designed to test whether an element is a member of a set. [Bloom1970] One of its core characteristics is that false positive matches are possible, but false negatives are not, meaning that while a query might incorrectly report that an element is in the set if it claims an element is not present, it is

guaranteed to be correct. This makes Bloom filters especially useful for applications where it is acceptable to have a certain probability of error in exchange for significantly reduced memory usage and high query performance.

1.6.1 History

The original paper gives an example of a hyphenation algorithm, deciding at which point a word at the end of a line can be broken into two words. [Bloom1970] The described algorithm considers two distinct cases where 90% of words follow simple hyphenation rules while the remaining 10% follow irregular hyphenation patterns that need to be accessed using expensive disk lookups to retrieve the rules stored in a dictionary that would be too large to fit in memory at once. By only saving the information about which item lookups are necessary The Bloom filter aims to eliminate unnecessary disk access by storing information about whether a word is part of the set of irregular hyphenation rules. In this example, following the given distribution, the majority of words encountered will be quickly identified as non-members of the set, in which case regular hyphenation rules can be applied, and disk access will be obsolete. Words belonging to the set will always be accurately identified as members of the set, making an expensive disk lookup necessary to retrieve the respective hyphenation rule for the word. With a low probability, a non-member could be falsely identified as a member of the set in case hash collisions occur, leading to an unnecessary disk lookup that returns the information that the word is not a member of the set. Since false positives only occur with a small probability, they are an acceptable trade-off for the fast lookup times in the majority of cases, and memory usage is greatly reduced by keeping only the Bloom filter in memory. This example serves as a demonstration of the utility of a Bloom filter in scenarios where data exceeds the available memory, but knowledge of set membership can be leveraged, greatly reducing expensive disk lookups.

The same concept from the original paper example has been found to be applicable to improving query performance and cache retrieval times. [2] [15] [46]

Also, other areas of Database Management have seen improved performance thanks to Bloom filters like Join Operations [41] [35] and Coupling between Tables. [12]

A variety of features in IT security also utilize the Bloom filter, which is acceptable for trade-offs of a low false positive rate. Examples include wireless communication, firewalling, and even replay detection in the LOFT protocol, judging the freshness of a packet in order to diminish the overhead of freshness checks, which could be used as a bottleneck in Denial-of-Service attacks where a Network is flooded with traffic by an attacker, rendering it less usable or even unusable for legitimate users. [24] [30]

Among other high-speed packet filters, the Linux kernel uses the Bloom filter to match IP and Port Tuples. [55] In this particular case, even wild cards are supported by zeroing the respective field, extending their utility even further.

As shown since their introduction in 1970, Bloom filters have been an ongoing topic of interest for many different fields, bringing improvements in a variety of use cases. In the following section,

we are going to take a close look at how the technology behind the Bloom filter works and what modifications and improvements have been made to enable such widespread adoption.

1.6.2 Mechanisms

The original concept proposed by Bloom in the original paper goes as follows. Essential to a Bloom filter are two elements, a bit array of length m and k , and different hash functions. In the beginning, the bit array is initialized with zeroes and gradually filled up with information about the set membership of elements inserted into the Bloom filter. As an element is added to the Bloom filter, it is hashed k times by the k hash functions, the output of which produces a value between 0 and $m-1$. Each output value generated by the hash functions, therefore, corresponds to a position in the bit array that is used as an index. For insertion, all indexes produced by the hash functions are set from 0 to 1. In Bloom's originally proposed idea, the bits in the array are never reset to 0, and the deletion of elements is not supported. When checking the set membership for an element, it is also hashed by the same hash functions, producing indexes that can be checked.

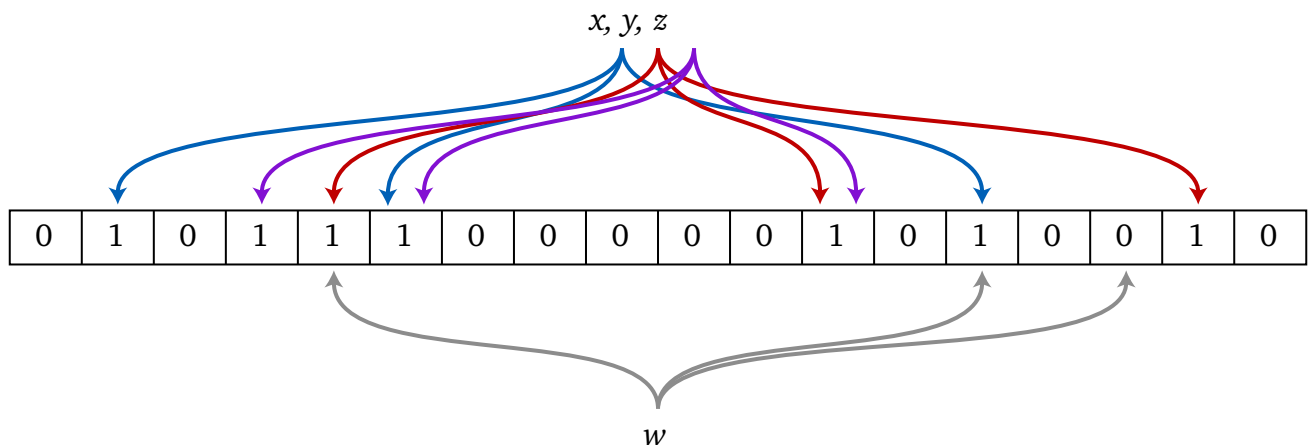


Figure 1.4: Insertion and Querying of an element via $k = 3$ hash functions [14]

If any of the positions are set to 0, the element was certainly never added to the Bloom filter, as the same hash value would otherwise have been encountered and set to 1 in the insertion process. In case all checked positions based on the calculated hashes for the queried item are set to one, two scenarios are possible. Either they have been set during the process of adding the element to the Bloom filter, or all hash functions have produced a hash collision leading to a false positive. This is the reason the Bloom filter is able to produce false positives. Bloom filters allow for some elements to be falsely identified as members of the set as a trade-off for its greatly reduced hash area size. Putting it differently, increasing the hash area size, i.e., the bit array, reduces the probability of encountering false positives.

This clever yet simple algorithm spawns both interesting and desirable behavior that marks the driving mechanism behind the widespread and diverse utility we have seen above. We are now looking at the math behind it, revealing and understanding how the different aspects of the Bloom filter play together.

1.6.3 Mathematic evaluation

The likelihood of a new element being falsely classified as a new member of the set can be calculated as follows: With $m \in N$ being the number of bits in the bit array and k being the number of hash functions used for each new entry, we can calculate the probability that a certain bit has not been set by any of the k hash functions:

$$\left(1 - \frac{1}{m}\right)^k.$$

by substituting

$$\lim_{m \rightarrow \infty} \left(1 - \frac{1}{m}\right)^m = \frac{1}{e}$$

we can transform the equation for a large m .

$$\left(1 - \frac{1}{m}\right)^k = \left(\left(1 - \frac{1}{m}\right)^m\right)^{k/m} \approx e^{-k/m}.$$

The probability that a certain bit after n insertions is set to 1 is

$$1 - \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-kn/m}.$$

The probability of all k positions being already set to 1 for a non-member of the set resulting in a false positive would therefore be

$$\alpha = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

This is only true if we assume that the hash functions produce perfectly uniform results and that all bits in the array are set independently of each other. [26]

Upon filling up the Bloom filter with new items, i.e., increasing n , the rate of false positives increases along the ratio of 1's to 0's in the bit array. Increasing m while n stays the same, the probability of false positives decreases. With enough available space, the maximum likelihood for a false positive along a Bloom filter lifetime (in the following called alpha) can stay extremely low or even near zero.

Besides its low rate of false positives the Bloom filter are also valued for their memory-efficient way of storing only the minimal required information about an item instead of storing other representations of data items in the form of hash tables, binary search trees or even arrays or lists containing all data themselves. After calculating the probability of a false positive, we want to look at the Bloom filters and other variables to establish an understanding of their relationships.

For a given m and n , the optimal amount of hash functions is

$$k = \frac{m}{n} \ln 2.$$

This can be inserted in the formula for false positive probability calculated above.

$$\alpha = \left(1 - e^{-\left(\frac{m}{n} \ln 2\right) \frac{n}{m}}\right)^{\frac{m}{n} \ln 2} = \left(\frac{1}{2}\right)^{\frac{m}{n} \ln 2}$$

and simplified as

$$\ln(\alpha) = -\frac{m}{n} \ln(2)^2.$$

resulting in the needed m in order to keep n items in the Bloom filter at a fixed error rate α

$$m = -\frac{n \ln(\alpha)}{\ln(2)^2}$$

as well as the optimal ratio of bits per element given m , n and α

$$\frac{m}{n} = -\frac{\ln(\alpha)}{\ln(2)^2} \approx -2.08 \ln(\alpha)$$

and the optimal value for k given α

$$k = -\frac{\ln(\alpha)}{\ln(2)}.$$

For a fixed α , the length of the bit array m is, therefore, proportionate to the number of items to be stored in the Bloom filter n .

[52]

As stated above,

$$1 - \frac{1}{m} \approx e^{-\frac{1}{m}}$$

is only an approximation for $m \rightarrow \text{inf}$. In addition to that, we assume that every bit in the Bloom filter that is set to 1 is independent of every other bit that is set to 1. Of most concern is the assumption that

$$k = \frac{m}{n} \ln 2$$

is fortuitously integral.

At last, we can give an estimate about the number of items n^* that have been placed in a Bloom filter based on the variables m , k , and the number of 1's being set

$$n^* = -\frac{m}{k} \ln \left[1 - \frac{X}{m}\right].$$

[54]

As for time complexity, the Bloom filter is a simple and efficient algorithm that requires only k lookup- or writing operations for checking and inserting elements, keeping the time complexity in constant time of $O(k)$ and marks the second attractive feature of the Bloom filter. This performance is achievable by utilizing the bit-arrays constant time lookups and insertions but also introduces the restriction that the array is fixed in size and can not be restructure to fit changing demands.

1.7 Bloom Filter Variation

After exploring the Bloom filter in its originally conceived version, we should now have a good understanding of its underlying algorithm, the desirable qualities derived from it, as well as the trade-offs that come with it. This gives us a good base for looking at some of the plethora of further developments that build on the original design that address some of its shortcomings, adding new features and adapting it to new scenarios. The following section aims to give an overview of some of the Bloom filter variations in existence while diving deeper into the more relevant examples and aspects for the goal of detecting duplicates in streaming data.

1.7.1 Scalable Bloom Filter

As we have seen, the false positive probability is directly dependent on bit-array size m and the number of stored items in the BF n . Upon filling up with more and more items, the probability of false positives increases. This behavior is normal and should be factored in when choosing the Bloom filter size when utilizing it in a specific scenario in order to not surpass the acceptable amount of false positives. However, the maximum n that has to be supported by the Bloom filter might not always be a known size, depending on the use case. Especially when multiple Bloom filters share a fixed pool of memory, the available space should ideally be distributed based on the individual Bloom filters' needs. The scalable Bloom filter addresses this problem by starting out with a regular Bloom filter of fixed size and setting a maximum false positive rate. If the rate is reached, another Bloom filter is added to contain the future items until its false positive rate is also reached, repeating the process. Queries for new items have to be answered by all Bloom filters in use. This simple adaptation of the original design allows for much more flexibility in memory allocation and makes it possible to control the false-positive rate effectively. Important parameters are the memory size m and maximum false positive rate, as they influence the amount of new BF spawned. It has to be noted that the lookup complexity increases with every new BF, which is queried in sequence, often starting with the oldest BF first and moving to the next newer BF when returning a negative result. This is a trade-off for only having a slight overhead when adding a new BF to the scalable Bloom filter that simply consists of creating a single empty Bloom filter of fixed size instead of performing an expensive remapping of all previous items into a new single larger Bloom filter. [1]

1.7.2 Counting Bloom Filter

One of the major shortcomings of the original design is its lack of deleting single elements. Using the standard Bloom filter, only the entire set can be deleted at once by resetting the entire bit-array to its original state, which is filled with nothing but zeroes. The trivial approach of resetting only the k indexes produced by the hash function for a to-be-deleted element is flawed. Even with enough space hash collisions between single hash functions upon insertion of new elements can and will occur and has no negative implications as long as the indexes are only set to 1 and never reset as is the case in the original design. If we would reset a single index back to zero we could therefore

affect more than the desired element introducing false negatives to the Bloom filter with a high probability.

The counting Bloom Filter (CBF) addresses this use case by replacing the bits in the bit array with counters. With the insertion of a new element, the counter at every index produced by the hash functions is not simply toggled from 0 to 1 but incremented by 1. When checking the membership of an element, a value greater than zero is interpreted as an indicator of membership. If all indexes checked are greater than zero, a positive result is returned. The calculation for false positives is similar to that of the standard Bloom filter, with the difference that we are looking at the potential for multiple hash functions to increment the counters instead of simply having bits set to one. Using the binomial distribution b , we can model the counting Bloom filter as follows. An m bit array is filled with $k * n$ values for all of the n elements inserted into it, which are hashed via k hash functions. The probability that a given counter in a CBF is one can be calculated as:

$$b(l, kn, \frac{1}{m}) = \binom{kn}{l} \left(\frac{1}{m}\right)^l \left(1 - \frac{1}{m}\right)^{kn-l}.$$

The probability of k counters being greater or equal to 1 can be calculated as:

$$p_{fp}(k, n, m) = \left(1 - \sum_{l < 1} b(l, kn, \frac{1}{m})\right)^k,$$

which can be used to calculate the probability for a false positive.

The hash functions are again assumed to be perfectly random. [55]

The ability to delete items from the filter works the same way as the insertion of a new item, except that the counters are decremented instead of incremented. Interestingly, when applied in practical networking scenarios, situations can arise that delete items that were never inserted in the CBF, leading to the possibility of false negatives. As studied by Guo et al., the occurrence of false negatives can have much more severe consequences than false positives, depending on the networking scenario. [27]

The implications for our application scenario will be discussed at a later stage when looking at concrete examples.

The counting Bloom filters support for deletion comes with the question which items should be deleted. As the information stored by Bloom filters is minimal there is no trivial way of determining which items are contained that could be deleted if the item is not present anymore. One answer to this is setting the counter to a maximum for an element lifetime upon insertion and decrementing them with every at every new arriving element. This is called a decaying Bloom filter. [11]

1.7.3 Rolling/Aging Bloom Filter

A relevant problem for providing replay detection is handling a possibly infinite amount of streaming data and performing membership querying on it. Adding a very large amount of elements to a Bloom filter can not be achieved by simply choosing an infinite amount of memory for a standard Bloom

filter or even adding more and more memory to a scalable Bloom filter. Because increasing the factor m is unfeasible and the maximum false positive rate should stay fixed, it is necessary to focus on the single remaining controllable element n , which is the number of messages to be kept in the filter. This can be achieved by finding a way to keep the relevant Items in the Bloom filter and cleaning up the irrelevant Items to free up capacity in the bit array. Counting Bloom filter support the deletion of items but are themselves unable to store any information about the nature of the stored items that could help in deciding whether they should be kept or not.

When looking at streaming data one often critical metric in deciding whether a given datum is still relevant is its age. Generalizing one can often assume that the older an Item in streaming data is the less relevant it is going to be for the present. This assumption does not have to hold true by any means and relies heavily on the use case, but in scenarios where membership-testing in old data is not of particular value the aging- or rolling Bloom filter provide a good and easily implementable way of keeping the amount of tested data in limits. In this quite naïve approach, at least two Bloom filters are filled up successively in a way similar to the scalable Bloom filter. After an amount of time passed or a threshold of items is reached the first Bloom filter is cleared of the now-obsolete data it holds and now begins to fill it up with new entries from the data stream. In this way, stale data is discarded, and the system requires only a fixed amount of memory if an upper bound for n is set instead of a time limit that controls when data will be discarded. This approach should illustrate the goals and challenges of dealing with streaming data and serve as a baseline for the more elaborate systems that will be explored in the next parts. [67]

1.7.4 Stable Bloom Filter

Tackling the problem of dealing with streaming data in a different approach, the stable Bloom filter has been described by. [17] The Stable Bloom Filter (SBF) provides guaranteed stable performance regardless of the amount of input data by continuously evicting old information from its bit array. Its approach is to focus on the ratio of zeroes to ones in the bit array of the Bloom Filter, as it is crucial for calculating the false positive probability, as we have seen in the calculation above. When dealing with streaming data, the amount n of elements to fit in the bit array of size m can easily lead to a state where the Bloom filter fills up to a point where it is almost entirely filled with ones, and the probability for false positives nears 1. The SBF extends the Bloom filters' functionality by keeping the number of stored elements at an almost constant value and, therefore, an almost fixed false positive rate. It operates by using a d bit counter in its array with a maximum value of $Max = 2^d - 1$. These fields are called cells in the SBF and replace the bits in the BFs bit-array. Like in the CBF, SBF cells are initialized with the counter set to zero, and non-zero comparisons are used in the querying process. Upon insertion of new elements, a query is performed to determine whether the element is already part of the set. After that, a number of P Cells are randomly chosen and decremented by 1 before inserting the new element by setting the counter of the cells according to Max .

This change in the algorithm leads to some interesting and desirable properties. Deng & Rafiei introduce the stable property to describe the SBFs convergence to a fixed fraction of zero to ones after several iterations.

Given an SBF with m cells, decrementing every cell with probability p at every iteration and set to max with probability κ at every iteration (note that this value has been renamed from the original paper to avoid collisions with the previously used letter k which will continue to mean the number of hash functions). With a large N and SBF_N being the value of the call after N iterations, the probability that the cell becomes zero after N iterations is constant.

$$\lim_{N \rightarrow \infty} Pr(SBF_N = 0)$$

The assumption is that the underlying input data is evenly distributed and does not change over time. This phenomenon is called the stable property. [17]

The stable point is defined as the limit of the expected fraction of zeroes to ones in an SBF with the number of iterations going to infinity. An SBF with the same conditions as above reaches stability with a fraction of zeros not greater than

$$\left(\frac{1}{1 + \frac{1}{P(1/k - 1/m)}} \right)^{Max}$$

with k being the number of cells being set to Max and P the number of cells decremented by 1 within each iteration.

Reaching stability implies a fixed and predictable false positive Rate not greater than the false positive stable point (FPS),

$$FPS = \left(1 - \left(\frac{1}{1 + \frac{1}{P(\frac{1}{k} - \frac{1}{m})}} \right)^{Max} \right)^k$$

We can note that in the equation above, the impact of $1/m$ is negligible compared to $1/k$, assuming $m \gg k$. Therefore, the amount of space in the SBF has only very little impact compared to P , which is the number of cells decremented in each iteration. This can be understood intuitively as clearing more cells each iteration leads to a faster clearing speed which should result in more zeroes and a lower false positive rate. Increasing Max increases FPS by introducing higher counters to be lowered to zero through decrementing iteratively.

In addition to the false positives discussed, the SBF introduces false negatives (FN) as a direct consequence of its design. A false negative occurs when an element that is part of the set is falsely reported as distinct. They occur when one or more cells that lie at the indexes of the element's hash values hold the value of zero even though the element is still part of the set. SBFs produce FNs when decrementing random counters at every turn leads to decrementing a counter to zero within the desired or expected lifetime of an element. The probability of a false negative occurring when an element x_i arrives that has last been added to the set with element $x_{i-\delta_i}$ and hashed to the cells $SBF[C_{i_l}]$ through $SBF[C_{i_k}]$ using the k hash functions can be given using the probability that Cell $C_{ij}(j = 1...k)$ is decremented to zero

$$PR0(\delta_i, k_{ij}) = \sum_{t=Max}^{\delta_i-1} [Pr(SBF_{\delta_i} = 0|A_t)Pr(A_t)] + Pr(SBF_{\delta_i} = 0|\bar{A}_{\delta_i})Pr(\bar{A}_{\delta_i}) \quad (1.1)$$

where

$$\begin{aligned} Pr(SBF_{\delta_i} = 0|A_l) &= \sum_{j=Max}^l \binom{l}{j} p^j (1-p)^{l-j} \\ Pr(A_l) &= (1 - k_{ij})^l k_{ij} \\ Pr(SBF_{\delta_i} = 0|\bar{A}_{\delta_j}) &= \sum_{j=Max}^{\delta_i} \binom{\delta_i}{j} p^j (1-p)^{\delta_i-j} \\ Pr(\bar{A}_{\delta_i}) &= (1 - k_{ij})^{\delta_i} \end{aligned}$$

with A_l denoting the event that the most recent time a cell has been set to Max has occurred at iteration $N - l$ and k_{ij} giving the probability of cell C_{ij} being set to Max .

The probability of x_i being classified as a FN upon arrival is

$$Pr(FN_i) = 1 - \prod_{j=1}^k (1 - PR0(\delta_i, k_{ij})).$$

It is of particularly important note that a $\delta_i < Max$ has a probability of 0 to be classified as an FN within the time of arrival δ_i . This is obvious as the counter could not possibly be decremented often enough even if decremented with every newly inserted element. Choosing the number of bits per cell d and thus the corresponding Max value gives us a minimum lifetime of an element in the SBF as it must be kept at least for $Max - 1$ newly inserted elements.

Apart from Max , setting the other parameters k and P can be performed using the calculations shown above. It is necessary that a desired maximum FP rate FPS are given by the user.

$$P = \frac{1}{\left(\frac{1}{(1-FPS^{1/k})^{1/Max}} - 1\right)(1/k - 1/m)}.$$

The amount of memory m is a negligible part of the equation as it is dominated by the much larger k .

Deng and Raffiei show in their paper that the value of k can be set regardless of m and is dominated by Max and FPS and usually falls under a value < 10 when minimizing FNR . Setting Max is highly dependent on available memory and should be set according to the amount of bits per cell as, $d = 2^{Max}$ to not waste space. The authors conclude that prioritizing values according to the respective use case and trying different alternative values according to the formulas is a practical way of setting values according to one's specific need. [17]

Time complexity is dependent on k and P . With k and Max constant, the time complexity required to process a data item is $O(1)$.

With its convergence to a fixed FPR, controllable FNR, and capability to handle unlimited streaming data, the stable Bloom filter combines many interesting qualities, extending the original BF design.

2 Methods

2.1 Problem Description

The problem of adding duplicate detection to the PSP protocol has many different facets that must be considered. An attacker who tries to perform a replay attack already has significant control over the network traffic, which allows for capturing traffic and tampering with the flow of packets. If not for the encryption and signing of packets, they could easily cause significant damage to the receiver, but even with those security measures in place, the possibility of a replay attack remains. As described in the introduction replayed traffic can cause various adverse effects on application layer protocols that have no protection against such a form of attack. It can be used both for targeted applications to trigger a specific vulnerability in the protocol or as part of a Denial of Service (DoS) attack that has the purpose of degrading performance or even rendering a network unusable. Because replayed traffic seems like legit traffic it is especially hard to detect and ironically especially signing traffic can even increase the potential harm done by such an attack by giving the attacker the ability to send packets that seem to originate from one specific sender that originally signed them. Those false flag attacks can have severe consequences not only for the receiving side's integrity but also for the reputation of the packet's original sender, who could be falsely accused of being the originator of the attack.

Another important aspect is that we try to add replay protection to an already existing protocol that is designed for large-scale data centers. PSP does not have sequence numbers that could help with identifying packets and already has made its design choices that we have to work with. One important aspect of PSP is that it has been designed to provide almost stateless performance, which can save on NIC hardware requirements. Since duplicate protection inevitably requires keeping track of the encountered traffic in some form, we have to try to deal with only low amounts of very costly NIC memory. In addition, we will operate on streaming data that does not have a fixed data set size but can fluctuate quite significantly. Dealing with a possibly very large amount of messages and only limited capacity will be one of the most important challenges for this work.

These factors all play together to form the starting point for this work, in which we try to find a solution that considers all important aspects. This can only be achieved by thoroughly identifying and evaluating all important aspects of this challenge.

2.2 Evaluation Methods

In order to evaluate the possibility of utilizing the Bloom filter for detection against replay attacks we are going to first describe the problems to gain a better view of the challenges a possible solution has to deal with. This will consist of gaining an overview of the environments in which the replay detection could take place. This includes sources from the official PSP documentation and examples from existing networking hardware that could be utilized in such an environment as a reference. Furthermore, we will define scenarios that should reflect different networking use cases and their respective limitations in order to narrow down the solution space of our work. These scenarios will include all important factors that are dictated by the networking environment, including the number of simultaneous connections, the amount of incoming packets, the available memory space that can be allocated for replay detection and the highest amount of false positives tolerated. The solution of the problems we intend to solve should orient itself on well defined parameters that will function as a base for all further developments and give us a way of comparison for the different results.

Since there are many different Bloom filter variations, we will have to choose the ones we wish to evaluate further. This choice will be part of the discussion and also include arguments against some of the other variations that will not be examined in this work beyond the introduction. We will then take these scenarios into consideration for the selected Bloom filters that are evaluated, taking their qualities and restrictions into consideration. The evaluation will be based on their mathematical background, which has already been described in the introduction but now includes a more practical approach of trying to fit their parameters to practical scenarios. Various parameters are going to be examined in detail with the goal of identifying their respective role and importance for the goal of replay detection as well as finding good values to gain maximum performance. Taking not only the already established formulas into account simulations will be written that help in gaining an understanding for the Bloom filters behavior in the scenarios we defined. Along the way, we will attempt to consider every aspect that plays an important role in both the practical applicability of this approach to large-scale data centers and our goal of providing security for these environments. Trying to identify the possible attack vectors of an attackers will also play a large role in this work in order to provide a broad view on possible solutions that only increase security. Finally, we will conclude our results and provide an outlook on important points of concern that could play an important role in future research or the practical application of our findings.

2.3 Environmental Parameters

2.3.1 Scenario 1

To evaluate different solutions to bring replay detection to the PSP protocol using Bloom filters, a quantitative description of the problem is needed. As networking environments and their respective requirements can differ quite a lot, we will describe three different scenarios in which Bloom filters could be utilized to grant replay detection. Scenario 1 attempts to capture the conditions outlined in the original PSP architecture specifications published by Google. [25] It will base all values for the

defined parameters on the specifications released by Google while trying to make the least amount of assumptions.

First we will focus on the maximum amount of packets a Bloom filter needs to handle in a given time frame. As we have seen, PSP borrows a lot of its concepts from IPsec including the Security Associations (SA) and their predetermined lifetime. During the SA lifetime, a cryptographic key is marked as active and used until the first of two conditions is met. Either a given amount of time that is typically set to 24h passed or the 31 bit SPI space is exhausted. While active, the Master Key is utilized to encrypt and decrypt traffic. After that period, it is kept for another entire SA life cycle to decrypt incoming traffic that still uses the old key.

We want to establish an upper bound for traffic that could theoretically occur within an SA lifetime. This number will be unrealistically high and should serve as an illustration of what dimensions could theoretically be handled by a Bloom filter especially when taking future networking developments into account.

We will assume an 800Gbit/s connection that is sending the smallest possible packets one-way using up all available bandwidth across the 10 Million connections stated by Google as their PSP architecture specifications. [5]

The smallest possible packet size will consist of the following components. At least 20 Bytes for the IPv4 Header, the PSP Header of 16 Bytes, not including Virtualization Cookies, a UDP packet as the PSP packets payload with a Header of 8 Bytes, no UDP Payload, and the PSP Trailer of 16 Byte to a total of 60 Bytes. If sent over Ethernet, the content would have to be padded to the minimum frame size of 64 Bytes. With this minimal packet size we can calculate the amount of packets sent in 24h over a 800Gbit/s link as follows.

$$\frac{24h * 60\frac{m}{h} * 60\frac{s}{m} * 800\frac{Gbit}{s}}{64\frac{B}{Packet} * 8b} \approx 1,81 * 10^{13} \text{Packets}$$

This number can be considered as a true upper bound as there is no way of fitting even more packets into a state-of-the-art 800 Gigabit connection. Including no payload data, however, is not a realistic assumption. In order to provide more realistic conditions we are going to assume standard simple IMIX distribution for payloads for future calculations. IMIX traffic profiles consist of distributions of packet sizes expected in different networking scenarios and are used to simulate network traffic in testing conditions by hardware vendors. Many IMIX profiles exist with slight differences and are used for use cases such as IPv4 or IPv6, TCP, or IPsec. The IMIX we are going to look at stems from an IPsec scenario and assumes a distribution of 90 Bytes for 58,67% of Packets, 92 Bytes for 2%, 594 for 23,66% and 1418 Bytes for 15,67 %, which is also the maximum packet size to avoid fragmentation of packets. We are primarily interested in the average packet size which is 418 Bytes with our distribution, on which we will add the 40 Byte PSP header and trailer including the Virtualization Cookie to a total of 458Bytes.

Within the standard 24h SA lifetime, we can calculate the packets the same way we did before.

$$\frac{24h * 60\frac{m}{h} * 60\frac{s}{m} * 800\frac{Gbit}{s}}{458\frac{B}{Packet} * 8b} \approx 53 * 10^{12} \text{Packets}$$

For both values, The amount of packets has to be doubled to cover the entirety of 2 SA lifetimes in which traffic can be handled by one key to a total of $2.7 * 10^{14}$ minimal Packets and $3.56 * 10^{13}$ IMIX distributed Packets. Google mentions supporting 10 Million simultaneous connections as their goal, which would lead to $2.7 * 10^7$ minimal Packets and $3.56 * 10^6$ IMIX Packets per connection. [5]

As for memory we will try to establish a lower bound of available capacity using publishings from Google that ideally already can be used to fulfill the requirements for our duplicate detection goals. The unidirectional SA based nature of PSP requires to keep track of individual connections that would have to be handled by data structures keeping track of individual connections, e.g. one standard Bloom filter per connection. For example handling all connections in one single large Bloom Filter would not be feasible because of the need to reset packet hashes for individual connections, at least in a standard Bloom Filter variant.

Google does not mention available memory capacity that could be used for duplicate detection but does note that 256 Bytes per stored cryptographic IPsec state per connection would not support their scaling requirements of 10 Million connections. Saving those 256 Bytes per connection was one of their reasons to decide against IPsec as it would provide “a represent a significant percentage of a NIC’s DRAM capacity”. [5] Ideally our solution would take up only a small fraction of this NIC memory and still be able to process enough traffic in order grant security across the entire SA lifetime. As for a lower bound, we are going to look at 128Bytes to be used for duplicate detection per connection to give us an idea of what is possible with such a low amount of memory.

2.3.2 Scenario 2

For a less strict scenario we are trying to create conditions that are going to cover a more broad range of networking environments. The lower requirements should ideally represent realistic conditions and at the same time enable us to use more of the Bloom filters potential for duplicate detection.

We are going to look at a Network Card akin to the AMD Virtex UltraScale+ HBM FPGAs VU57P that keeps 10000 connections simultaneously its 100Gbit/s port. [3]

Assuming a 100Gbit/s connection that sends IMIX distributed packets both ways with 50% of traffic going into each direction and the SA lifetime being set to 12h we map out conditions which should still hold up in most networking environments. These parameters result in

$$\frac{12h * 60 \frac{m}{h} * 60 \frac{s}{m} * 50 \frac{\text{Gbit}}{s}}{458 \frac{B}{\text{Packet}} * 8b} \approx 5.9 * 10^{11} \text{Packets}, \text{ or } 1.18 * 10^{12} \text{Packets for a double rotation and } 1.18 * 10^8$$
 packets per connection.

The mentioned card has 270 Mbit RAM which would leave us with an amount of 27Kb for each of the 10000 connections.

2.3.3 Scenario 3

Going even further with relaxing the requirements in Scenario 3 we are looking at a Network Card akin to the AMD Kintex UltraScale+ FPGA Board KU19P that keeps only 10 connections that, although the mentioned NIC supports up to 100Gbit/s connections only utilize a combined 10Gbit/s across all connections which 50% IMIX traffic are sent in each direction. [3] The SA Lifetime will be set to 1h which would increase the overhead of creating new SAs significantly but could be seen as still realistic with only 10 connections. This scenario should serve as a special case example as only a few networking environments could make use of such a low amount of connections while justifying the cost of expensive NIC hardware.

$\frac{1h * 60 \frac{m}{h} * 60 \frac{s}{m} * 5 \frac{Gbit}{s}}{458 \frac{B}{Packet} * 8b} \approx 4.91 * 10^9 \text{Packets}, \approx 9.83 * 10^9$ per double rotation and $9.83 * 10^8$ per double rotation per connection

The card's 60Mbit memory across 10 connections leaves a generous amount of 6Mbit per connection.

2.3.4 Acceptable False Positive Rate (α)

As described in the introduction, a false positive occurs when an element not previously encountered is reported as duplicate. Setting performance requirements for our accepted false-positive rate α is a good way to start narrowing down the other parameters, as the false-positive rate will ultimately limit the usefulness of the duplicate detection mechanism. When utilizing a BF, it will inevitably produce an increasing amount of false positives upon filling up. This creates an important effect inside the network, from the receivers perspective duplicate packets as well as false positives will be discarded upon arrival and appear as dropped packets. Discarded packets that were correctly marked as duplicates are the desired effect of replay detection and dropped packets from other causes inside the network such as congestion errors, hardware issues or even interference are outside of the replay prevention mechanisms control. Additionally, introduced packet drops caused by false positives in the duplicate detection mechanism, however, can have varying implications for different tasks. For UDP, dropped packets due to false positives might lead to a loss of quality in streaming video or audio data, as there is no retransmission in UDP. [31] The acceptable limit of dropped packets has to be determined for each respective use case.

In TCP, however, it is possible to calculate the respective impact of lost packets on the throughput rate. TCPs congestion control mechanism interprets lost packets as a sign of network congestion and will adjust the transmission rate following it's additive increase and multiplicative decrease (AIMD) approach. [9]

Upon introducing further dropped packets with the application of Bloom Filter false positives the congestion avoidance algorithm will negatively impact the transmission rate.

The impact of lost traffic for TCP however can be calculated by looking at its congestion avoidance algorithm using the Mathis Formula.

$$Throughput \leq \frac{mss}{(rtt * \sqrt{p})}$$

[39]

Knowing the round trip time and the networks probability for packet loss p we can establish an upper bound for possible throughput. For demonstration purposes we can distinguish 3 different scenarios for different network connections with the different latencies of 150ms, 50ms and 1ms. The latter 1ms would be realistic latency inside a data center in which PSP is applied, while the 150ms and 50ms could be achieved for connections between data centers depending on their location and distance. [7] These numbers should give an idea of the possible trade-offs that varying amounts of packet loss can cause.

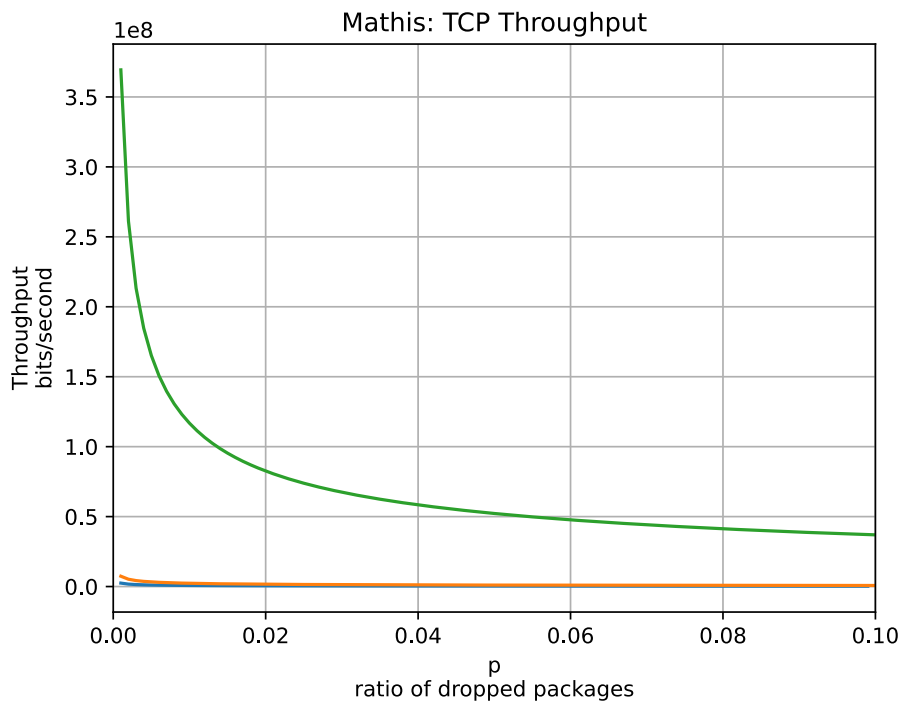


Figure 2.1: TCP bandwidth with increasing loss rate[65]

It is to be noted that the Mathis formula does not apply when packet loss is so rare that the TCP window regularly becomes fully extended but given that the increase of packet loss will either be high enough to make this restriction of the formula irrelevant or will be low enough to not have an effect on the networking environment the formula can be used in our case.

The Mathis formula shows that the impact of increased packet loss will hit low-latency connections significantly higher. Generally a network should ideally stay below 1% packet loss in order not limit a fast and low-latency connection too much. When defining an upper bound for additionally introduced packet loss we have to take into account that it will be added on top of already existing packet loss inside the network. The highest tolerable loss rate can vary between individual applications and cases based on network specs and underlying protocols but for the rest of this work we will assume that 0.01% additional loss rate introduced by the Bloom Filter will be tolerable for most scenarios as we stay way below generally still accepted 1%. [10] It should be kept in mind that in scenarios

where more packet loss is tolerable this value could be increased to 0.1% or even 1% in order to gain better Bloom Filter performance in terms of memory usage.

2.4 Bloom Filter Selection

Now that we have quantitatively established the different networking environments, we can utilize them as a basis for the rest of this work. They will serve as a baseline for calculations and simulations with the goal of bringing Bloom filter based protection against replay attacks to these described networks.

As we have seen we have a lot of options to choose from when it comes to the question which Bloom filter variation we could utilize in this goal. This section will provide an overview of the Bloom filter variations mentioned in the introduction. We will briefly point out their respective qualities and take note of their advantages and disadvantages concerning their ability to provide replay detection.

First of all, we would like to test the standard Bloom filter for our scenario to see what the Bloom filter, in its most basic form, is able to accomplish. Our aim is also to gain a better understanding of the mechanisms at play, especially when it comes to the Bloom filter behavior in the lower memory scenarios we mentioned above. Also, the standard bloom filter is going to answer a lot of questions about other bloom filter variations that are closely related to it. After testing the standard Bloom filter, we could easily extrapolate our findings to variations like the rolling Bloom filter.

After we established an understanding of the standard bloom filter, the stable bloom filter seems like a very promising choice for further examination. Not only is it among the Bloom filter variations that come with a decaying mechanism to handle streaming data, but its convergence towards a fixed false positive rate provides a good way of ensuring that we can safely stay below the maximum false positive rate we set in concern for TCP. As the stable bloom filter also shows similarities to the counting or decaying bloom filter while extending both their functionality by a promising deletion mechanism it should be an optimal choice for our scenario.

By picking those two examples, we cover a broad area of available bloom filters that could be utilized in our scenario.

3 Findings

3.1 Bloom Filter Evaluation

After establishing the standards for different scenarios in which a Bloom filter could be applied to provide protection against replay attacks, we will evaluate different solution attempts. In this process we will hold different Bloom filter variants against the defined scenario based target and see whether they are able to provide an adequate solution. Additionally, we will see which adjustments can be made to tweak performance and what trade-offs are possible.

3.1.1 Bloom Filter Mathematical Evaluation

The standard Bloom filter, described in detail above, is a good starting point for providing a baseline for other variants. Other qualities include that the standard Bloom filter is well understood and tweaking parameters can be performed by using the already existing equations.

We are now going through each scenario successively, trying to find constellations of parameters that satisfy the given requirements as well as discussing areas of interest and potential for trade-offs and tweaking.

Standard Bloom Filter - Scenarios

As described above, we are trying to use only 128 Bytes or 1024 Bits for the entire BF and an $\alpha = 0.0001$ in this scenario. With the usual size for the Bloom filters bit array being much larger, we can already expect that fitting the desired $3.56 * 10^6$ IMIX Packets into such a filter is not going to succeed.

Using the formula for the optimal bit-per-element ratio derived above yields the number of required bits per element,

$$-2.08 \ln(0.0001) = 19.16$$

This ratio shows that 1024 can hold only approximately $\frac{1024}{19.16} = 53$ elements. Even allowing for an α of 0.01% would only allow for $\frac{1024}{-2.08 \ln(0.01)} \approx 107$ Elements. In order to fit the entirety of

$3.56 * 10^6$ Elements into a standard BF we would require a total of $3.56 * 10^6 * -2.08 \ln(0.01) \approx 3.41 * 10^7 \text{ bit} = 4.26 \text{ MB}$, which would mean that only 1173 of the desired 10 Million connections surpass the 5GB DRAM per NIC that were the reason for Google to invent the PSP protocol in the first place. [psp_arch_spec]

These not very encouraging results already hint at what we can expect from the other two scenarios with more available memory.

In scenario 2 we gave ourselves 27Kbit which, at an error rate of 0.01% could handle $\frac{27*1000}{-2.08 \ln(0.0001)} \approx 1409$ Entries or $\frac{27*1000}{-2.08 \ln(0.001)} \approx 1879$ Entries at an error rate of 0.1%.

Scenario 3 allocates a generous amount of 6Mbit for duplicate detection per connection which could handle $\frac{6*10^6}{-2.08 \ln(0.0001)} \approx 3.1 * 10^5$ or $\frac{6*10^6}{-2.08 \ln(0.001)} \approx 4.18 * 10^5$ Entries with false positive probabilities of 0.01% and 0.001%. This is still only a fraction of the desired $9.83 * 10^8$ packets we would ideally like to handle in one large standard Bloom filter, but it comes quite a bit closer than the other two examples. Handling the desired amount of packets in one Bloom filter would require up to $18.8 * 10^9$ bit or 2.36GB, which is also clearly out of scope for NIC hardware.

As we can see, the Bloom filter, although memory efficient, still requires too much memory to handle any realistic networking scenario in terms of expected packets and available NIC memory if we try to handle every packet in one single large Bloom filter.

In addition to setting the desired maximum amount of false positives, another relevant problem arises. Bloom filters are probabilistic data structures. The occurrence of hash collisions that lead to false positives is directly dependent on the size of the bit array and the number of ones set in it, which means that they are generally predictable when working with large amounts of memory. As an easy to understand example we can see that in our 1024 bit array in scenario 1 a second false positive would already lead to surpassing the threshold of 0.01% false positive rate. [45] If we think about applying Bloom filter in scenarios where memory is limited by the NIC to a very small amount further investigation into the distribution of false positives in low amounts of memory is required. In the following we are going to look at a simulated Bloom filter to answer some questions relevant for practical applicability.

3.2 Standard Bloom Filter - Simulation

Simulating a Bloom filter gives us the ability to answer deeper questions about its behavior when dealing with small amounts of memory. The goal of this endeavor is to perform test runs of a Bloom filter from initialization until reaching a desired maximum false positive rate. During that process, we can gather quantitative data about fill state, false positive rate, and distribution that we can then inspect further.

The simulation for this work was written as a python script. [58] Being a quite straight forward algorithm the Bloom filter constitutes of a list initialized with zeroes, and k hash functions that are simulated using the python sha256() method from the popular hashlib library. in a loop that

appends the numbers of $0..k - 1$ to the message leading to k different and reproducible hash values. [19] The messages that constitute the entries or packets that are to be placed inside the Bloom filter are pre-generated to produce 64 random Bytes. Due to the birthday paradox, the approach of ad-hoc generation of messages using the `os.urandom()` method has been discarded, and care has been taken to ensure that no duplicates exist in the messages utilized. [8] As an important additional step for every newly created Bloom filter object, a random 64bit seed is chosen using `urandom` that is utilized as a bias for the generated messages. Otherwise, the same messages would be read from the file and paired with the same $0..k$ hash seeds, ultimately introducing a large bias to the simulated packets. We expect to deal with encrypted traffic in real world scenarios, therefore simulating incoming messages as random data should be an appropriate assumption. When dealing with real traffic however it is important to make sure that the hashed part of the message consists of unbiased data as having hashes of equal data will inevitably lead to hash collisions and if a specific type of packet will always produce the same hashes the Bloom filter will throw those packets out reliably. This means that the replay detection part of a network becomes a huge liability, which should be avoided.

A loop reads the prepared messages from a file in a line by line fashion and tests for membership by calculating their hashes and checking the respective locations on the bit array. The approach of having no true duplicates in the dataset ensures that every encountered duplicate must be a false positive, granting an accurate measurement of false positives. Additionally the script has been tested to ensure that true positives can in fact be detected as a test of functionality. If a message is found to be new it is then inserted into the Bloom filter and the loop continues. In case the message is marked as duplicate, the false positive counter is increased, and a check against the desired maximum false positive rate across the number of all previously encountered messages is performed. This concept of initializing the Bloom filter and filling it with simulated traffic data in a loop marks the base for multiple different scripts that have been developed to test different areas of the Bloom filter.

The first script we are looking at tests whether the implementation produces the expected results in terms of false positives. A Bloom filter is initialized with fixed values for m and n as well as k and filled up until it contains n messages. The false positive rate was originally calculated as $\frac{\text{\# of duplicates}}{\text{\# of simulated messages}}$. This metric should provide insights that are useful in a real world scenario as we would be primarily interested in knowing how many of originally sent packets actually arrive. Upon a test run of the script, however it became obvious that this metric does not fully encompass the demands for a practical application as $m = 10000$, $n = 8000$ and $k = 8$ resulted in a false positive rate of 0.578125 which is only approximately half of what we would expect given the formula calculated

above $\alpha = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx (1 - e^{-kn/m})^k$ that would produce an $\alpha \approx 0.99$. The reason behind

this difference is that while we calculate the false positive probability as an average above all packets the formula gives us the probability for an item that is newly inserted into a BF of given properties to be marked as a false positive. This discrepancy is especially important given our findings on the impact of high false positive rates on TCP. An example with $m = 100000$, $n = 9500$ and $k = 8$, chosen for illustrative reasons can give us a better view on the situation. While the average false positive rate across all packets was ≈ 0.00074 in a test run, which might still be desirable as we have argued above, the false positive rate increases at an exponential rate when reaching the last packets to a

final value of ≈ 0.0064 which can be calculated using the formula for FP probability from above, over 8.5 times the average. A modification in the code to evaluate this phenomenon gives us the probability in a sliding window with the size of 100 and shows that the average false positives in the last 100 messages were 0.02. Spikes like this towards the end of the Bloom filters capacity are certainly to be avoided as reaching 2% additional package loss rate can significant impact on a high speed TCP connection as shown above.

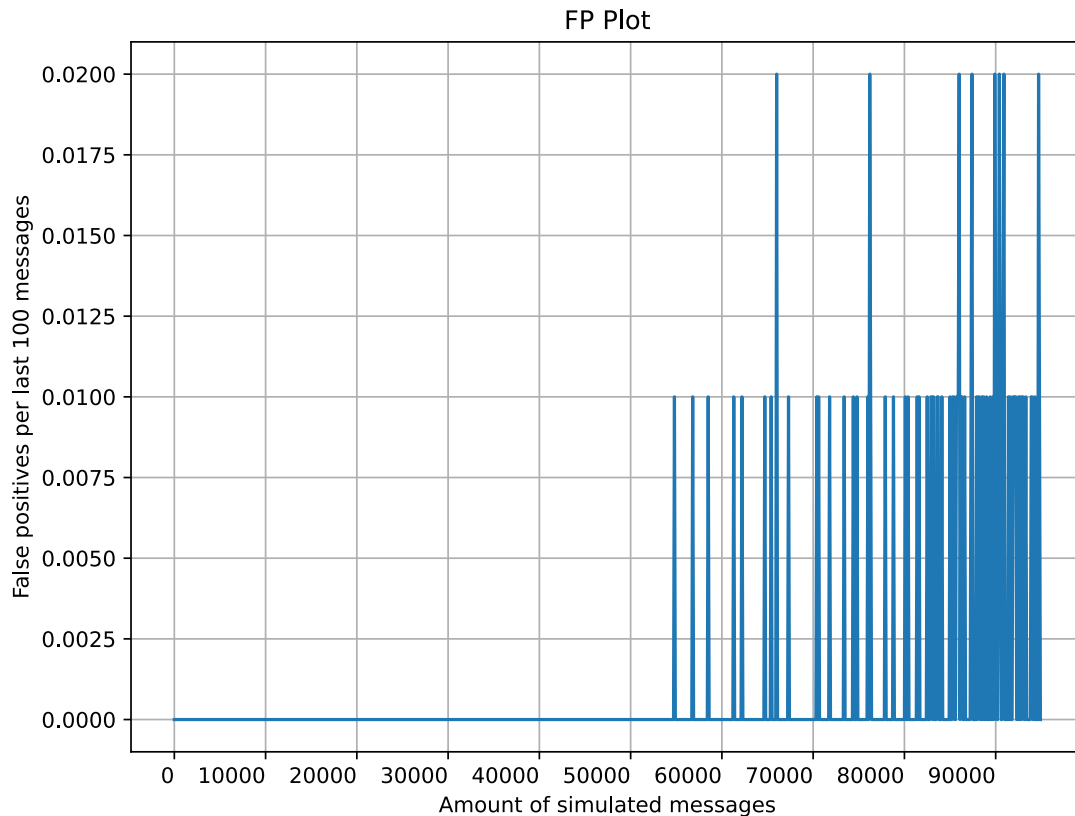


Figure 3.1: Development of during the process of filling up a BF rate [59]

This graph shows the distribution of false positives over batches of 100 consecutive messages. $m = 1000000$, $n = 95000$ During the first half, false positives are very rare but increase significantly in frequency when storing the second half of messages in the Bloom filter, eventually surpassing the acceptable amount of false positives.

As we do not have a way of quantitatively measuring the false positive probability of each new individual element during runs of the script we will use the average false positives in the sliding window of last x elements like we did above for future FP measurements. This is a lot closer to the probability of the latest element being classified as a false positive compared to the average FP rate we calculated earlier but is not an exact measurements. Comparing the 0.02 over the last 100 elements measured in the example above to the calculated value for the last elements FP probability

of 0.006 shows that the amount of false positives can fluctuate when looking at only a small number of elements. With $m = 10000000$, $n = 950000$, the FP probability for the last element remains unchanged when calculated using the formula. Setting the sliding window set to 10000 results in values falling much closer to the expected value of 0.006.

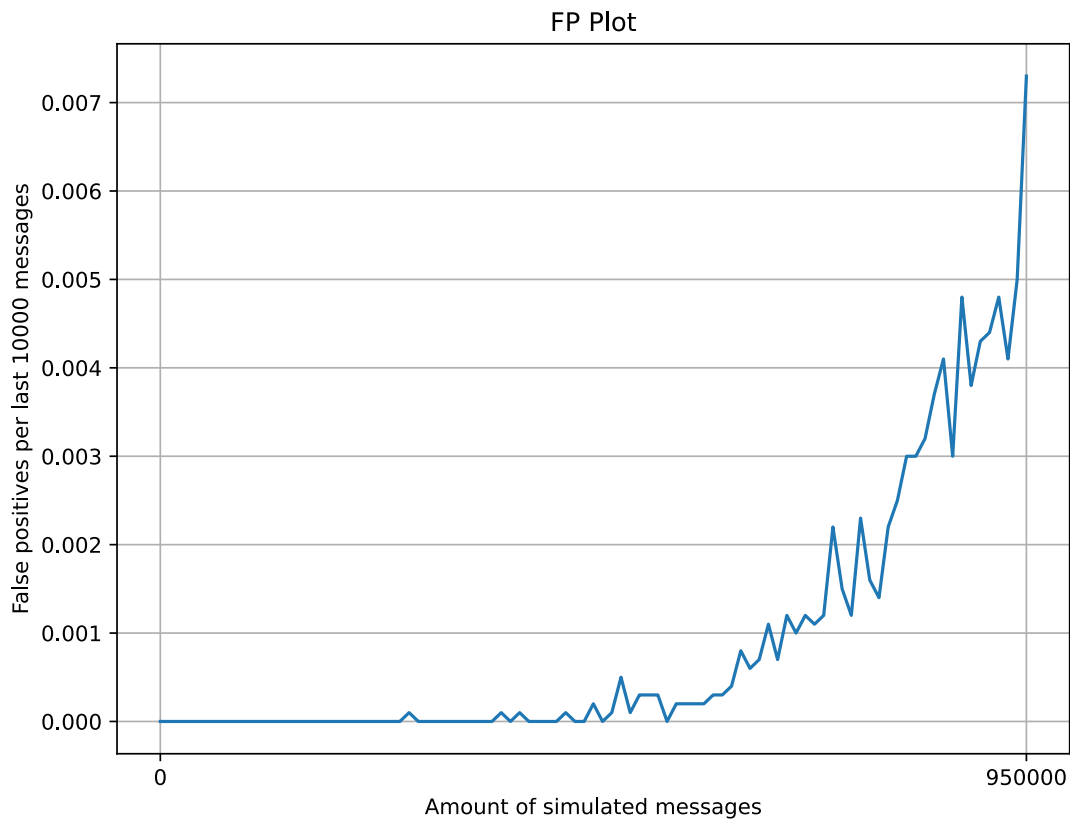


Figure 3.2: Development of during the process of filling up a BF rate [60]

Smoothing out the false positive rate should not distract from the fact that false positives do not occur evenly spread out. This effect has to be kept in mind as a local increase in false positives can and will effect performance negatively. One way to counteract this phenomenon could be to chose an accepted false positive while keeping in mind that it could be exceeded by a factor of up to 10. With a value on the lower end of our accepted range of 0.01% we should have enough tolerance during FP peaks and profit from an exceptionally low FP rate in the rest of time.

Taking our insights on false positive rates to another case we can now attempt to answer the important question of how much memory we should allocate for a given amount of messages. Utilizing the formula $\frac{m}{n} = -2.08 \ln(\alpha)$ we already saw that at a FP rate of 0.0001 we would need 19.16 Bits per element at an optimal $k = -\frac{\ln(\epsilon)}{\ln(2)}$, which will be rounded to the nearest integer. We saw that for 2 of our 3 scenarios the calculated values were too far from acceptable performance with scenario

3 being the only one that could be considered interesting. For use-cases where a high amount of memory is available, the amount of messages is much lower or both the Bloom filter could still be an interesting solution for duplicate detection. For those cases, we want to find out how the Bloom filter behaves with lower amounts of memory through a simulation. Our approach will be very similar to the FP simulation, with the only difference being that we set m and an α , which will constitute the maximum accepted fraction of FP in a sliding window. When filling up the Bloom filter and eventually reaching the FP threshold within the sliding window the script will break and provide information about the amount of messages stored. For a better illustration of the dependency between m and n the script will perform several iterations with different values for m always increasing by a factor of 10.

As we can see in the graph, the relationship between m and n stored in the BF is very linear, as we would have expected given the BF calculations. With an $m = 10^9$ we are able to fit $n \approx 7.7 * 10^7$ messages in the BF before reaching the threshold FP rate of 0.01 in the last 1000 messages. This performance measurement results in an approximate 12.96 Bits per message in contrast to the expected 9.58 Bits per message that we expect from the calculation. The inferior performance is easily explained by the stricter cut-off rule that terminates the loop upon reaching the maximum FP rate within the last 10000 messages in contrast to the calculated probability for the next inserted element being a FP.

Graph showing the dependency between different values for m and their supported amount of messages n a BF can hold until reaching a $FPR \geq 0.01$ within the last 100 messages.

After confirming similar performance in comparison to the expected results regarding the BF calculations we can take a look at the behavior of a Bloom filter at lower amounts of memory. We compare the results of 100 measurements at $m = 10^i$ with $i = \{2, 3, 4\}$ and find that they all show considerable statistical variance. In a bad run, a Bloom filter could reach the maximum tolerated FPR at half its capacity from an average run.

Boxplots of different smaller values for m and their supported amount of messages n a BF can hold until reaching a $FPR \geq 0.01$ within the last 1000 messages.

This is also an important finding to be considered for application in real world scenarios, as there are large fluctuations in capacity given only limited available memory. A Bloom filter reaching its maximum capacity when taking the FPR as a hard limit, as we did in this example, would mean that its lifetime is reached much faster than expected initially. Handling a full Bloom Bloom filter might differ between application scenarios but could make clearing out the BF necessary, which would mean a shorter lifetime of the data stored in it. In scenarios that allow it more memory could be allocated for a new BF as a substitute or, alternatively a higher FPR could be simply accepted until a certain amount of messages is reached. The latter strategy does not necessarily mean that the FPR after reaching the initial threshold has to be as high as or even higher than before but as we saw in the example above FPR tend to escalate quickly after a certain point and terminating early is the only way of guaranteeing that future added messages do not suffer from a higher-than accepted FPR.

These insights suggest that without sufficient memory performance, the standard Bloom filter is not able to perform well enough to handle large amounts of traffic. In environments where only a low

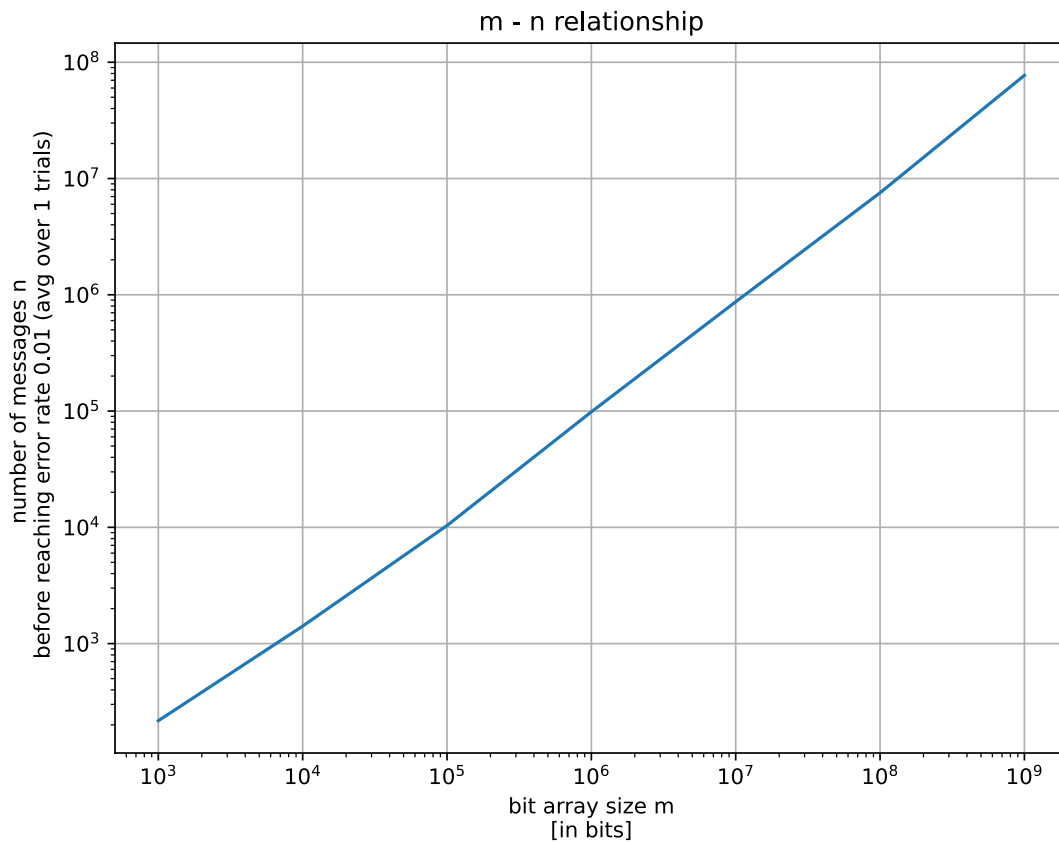


Figure 3.3: Relationship between m and n[63]

amount of memory can be dedicated to duplicate detection, the standard Bloom filter suffers from low capacity and struggles with fluctuating false positive rates. Its low predictability under such conditions suggests that it is not cut out for the task, and even if one is inclined to utilize a large amount of memory, the goal of fitting the entire traffic from SA lifetimes (a double rotation) into a single BF might still be too much depending on traffic requirements. This would make rotating BFs with limited lifetime necessary, which introduces false negatives due to their limited lifetime.

When we are accepting of the fact that false negatives might be a necessary evil for our goal to try to bring duplicate detection even to low memory environments we should take a look at other Bloom filter variations that are suited better to the demands of high streaming data environments.

3.3 Stable Bloom Filter Evaluation

The Stable Bloom Filter has been designed with exactly this goal in mind. In contrast to the standard BF it is able to deal with an unending input stream by using counters instead of bits of which

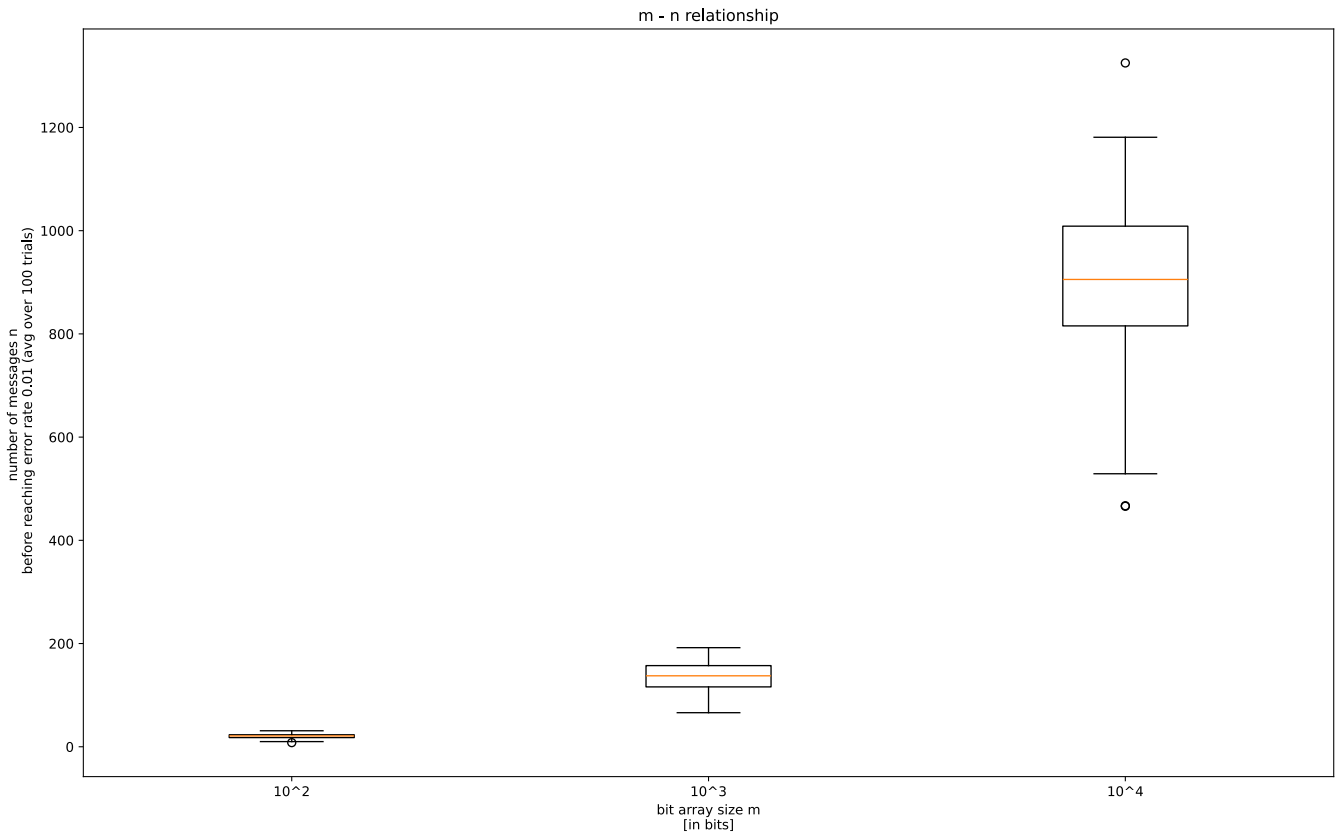


Figure 3.4: Relationship between m and n[64]

with each new element some are chosen at random to be decremented in order to clear out stale data. This works in a way that grants a convergence towards a fixed FPR that does not pass $FPS = (1 - (\frac{1}{1 + \frac{1}{P(\frac{1}{k} - \frac{1}{m})}})^{Max})^k$ at any given point, even before reaching stability. As we have seen in the calculations above for FPS the value of m can largely be ignored making the value of FPS mostly dependent on Max , the maximum values the SBFs cells are set to, k which denotes the amount of Cells set to Max and P , the amount of cells decremented per iteration.

3.3.1 False Positives

For our case these concepts should play together very well by providing a ceiling that the FPR does not pass at any given point. We should simply choose the parameters of Max , P and k in a way that result in a desirable FPS . The parameter P marks the amount of cell counters to be decreased and therefore plays an important role in driving the amount of non-zero values down, to avoid a state where too many cells are filled to utilize the SBF at the desired FP rate. Max , on the other hand

specifies the value the cells are going to be set with each newly inserted item. Those parameters are going to work against each other, keeping the amount of zero and non-zero cells in balance which determines the FPR . In addition to that k works as a multiplier for Max as there are always k cells being set for each added item which also impacts the balance of Max and P .

In order to do that a simple script utilizing the known Formula for FPS has been written that simply produces a graph for different values of P at a fixed $k = 8$.

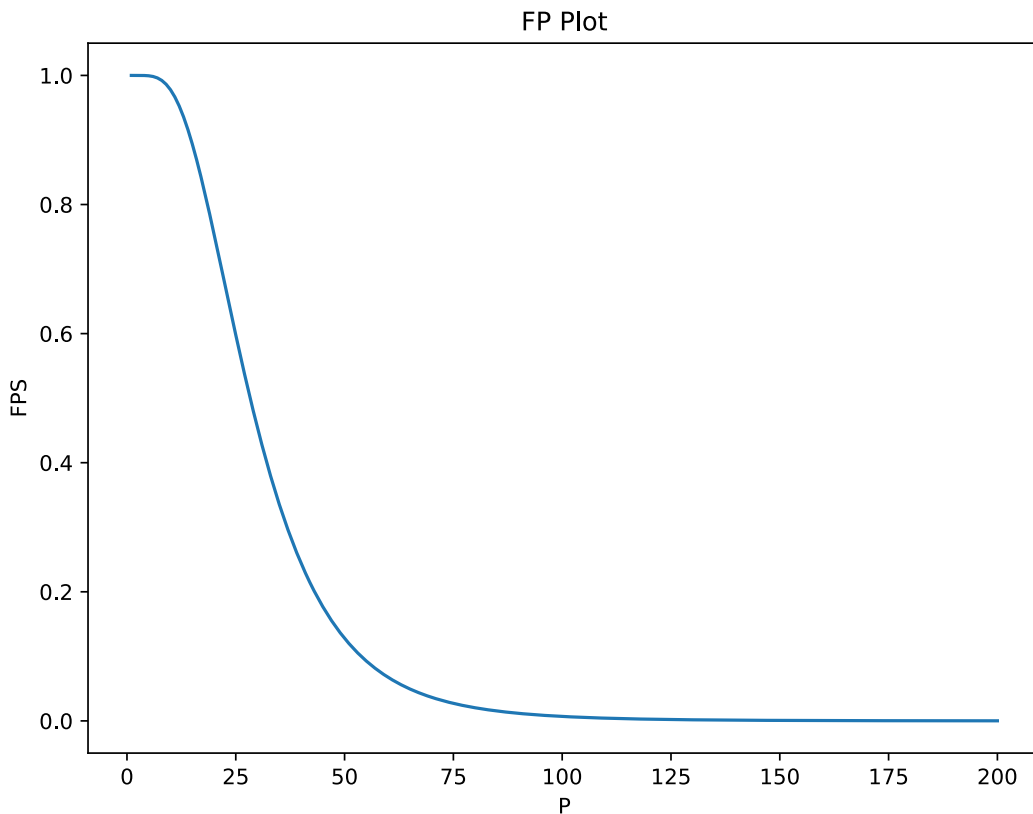


Figure 3.5: Graph showing the relationship of P and FPS with $m = 1M$, $k = 8$ and $Max = 10$. [61]

The SBF reaches an $FPS \approx 0.001$ with $k = 8$ and $Max = 10$ at $P \approx 143$ and $FPS \approx 0.001$ at $P > 200$. As the value of m does not have a large impact on FPS as was calculated above. Even for the low value of $m = 1000$, we reach FPS at the same values for P . Thus, we can expect the values for P to hold up even when deploying the SBF with low memory.

When setting $P = 143$ and keeping m at $1M$, we compare values for k to see their impact on FPS .

It is not a surprise to see FPS initially decline at low values for k as we have picked the other parameters to minimize FPS with k set to the fixed value of 8 in our example above. The minimum that is in fact reached at 8 slowly rises with an increasing k reaching an approximate linear dependencies for values beyond $k = 50$.

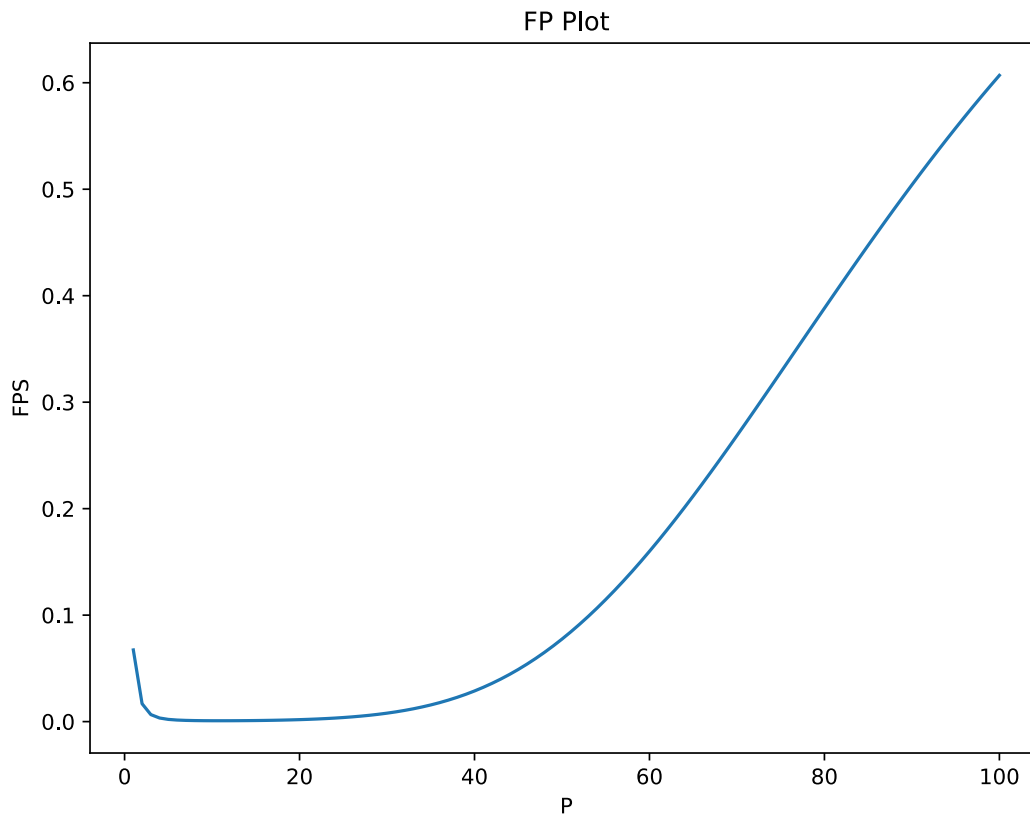


Figure 3.6: Graph showing the relationship of k and FPS with $m = 1M$, $P = 143$ and $Max = 10$. [62]

Expectedly, out of all parameters, Max has by far the largest impact on the choice of parameter P . With all other parameters staying the same an increase of $Max = 100$ requires a $P \approx 1500$ to reach $FPR > 0.001$ and $P = 2000$ to reach $FPR = 0.0001$

This sharp increase for the parameter P does not mark a problem as it can be set as a parameter regardless of hardware restrictions but it should support our desired false positive rates and has to be set with regard to Max which will be a very important parameter for our application as we will see. As we will continue with setting the SBF parameters for dealing with false negatives, the value of $k = 8$ will be continued to be used as a default as we can see that most values for k should perform approximately the same unless chosen very low or beyond a value of 20 which would impact performance. Not tampering with k also gives us the freedom of focusing on other parameters that require more of our attention and have a much larger impact on performance as exploring them has the potential of revealing more interesting and less obvious relationships.

3.3.2 False Negatives

Introducing and dealing with false negatives is one of the necessary drawbacks of the SBF. Enabling us with the ability to handle unrestricted amounts of messages means that data has to be discarded at some point, causing false negatives as a consequence when an Item that has previously been added to the set leaves the set before encountering it again. On a technical level, any item that has one of its cells set to 0 can cause a possible false negative when the same cell is tested at a point in time after the counter reaches zero. While false positives can be controlled by simply setting the parameters of P , k , and Max accordingly, which results in a value for FPS that will not be surpassed by false positives, the handling of false negatives could be a lot more delicate.

As mentioned above all parameters relevant for the setting of FPS will also be relevant for finding the according FNR . In their work, Deng et al. [17] go to great lengths to determine the ratio of false negatives.

Their calculations, which we already saw above, revolve around finding the probability of a newly added item x_i and its nearest predecessor $x_{i-\delta_i}$. If the nearest predecessor has one of its cells decremented to zero, the newly tested item will be marked as a FN. The FN probability is largely dependent on the age of the element and Max . If an item occurred more iterations than the value of Max its likelihood of having at least one of its counters reaching zero will increase. The most relevant factors here are Max and the number of iterations since the insertion of the last occurrence as a cell can only be decremented once every iteration. Having calculated this probability Deng et al go on to construct more practical example around it that work with setting a duplicate frequency in order to calculate the probability of FN occurring under more specific circumstances.

While these calculations sound interesting and relevant to our case as we want to controll the amount of false negatives we should take a step back and remember what our goal is and what these probabilistic calculations can provide. Considering that we make assumptions about duplicate frequency, it would mean that we make assumptions about an attacker. If for example 0.1% Duplicates are to be assumed we could calculate how messages we would miss if an attacker was to perform a replay attack for every 999 messages send through our network! Taking the attacker's perspective reveals the obvious flaw in this assumption. They could simply increase the amount of true negatives, i.e. messages that are replay attacks until a single one of them comes through. Accepting any amount of FN can be exploited and following the principles of good IT security, we should assume that it will.

Taking these considerations into account makes one concept from Deng et al.'s FN calculations especially interesting. When $\delta_i < Max$, the probability of a single cell being decremented to zero within δ_i iterations becomes zero, meaning that the FN rate is 0. Because with every iteration, only one decrement happens, choosing a Max value means choosing a minimum lifetime for each newly set item in the form of a minimum number of iterations until one of its cells can reach zero.

The idea of setting a very large Max to control the minimum lifetime of an element has a couple of implications. First of all, we must choose what lifetime for a message would be appropriate. This could be achieved fittingly by utilizing the values from our scenarios, i.e., scenario 1 $3.56 * 10^6$, scenario 2 $1.18 * 10^8$ and scenario 3 $9.83 * 10^8$. Since setting the Max value does mean that every cell need the according amount of bits to represent this value we will round up the needed amount

of bits in order to not waste capacity in the cell. This means for the respective scenarios that 23, 38 and 31 bits would be needed per cell which would mean $m * bits$ times the amount of space in our SBF. With scenario 1, this would mean that only 44 cells would be supported. Scenario 2 and 3 with 27Kbit and 6Mbit would allow for 710 and ≈ 193000 Cells. Contrasting these numbers of cells with the exorbitant amount of messages will obviously lead to a FPS of 1, guaranteeing that the number of false positives will be smaller or equal to one. This should not come at a surprise, the maximum amount of P is bound to m since we can not decrement more cells than the SBF contains and after all SBF are not magic and should not surpass the standard Bloom filter in terms of capacity. The SBF's advantages come by reducing the streaming data to a sliding window that can be managed with acceptable FP and FN rates.

Putting a pin in the problem of preventing replay attacks that stem from messages sent at a time that is not covered by the sliding window anymore, we can calculate how many messages can be managed by the respective amounts of cells and try to solve this problem separately. This can be done by transforming the given FPS equation to the value of Max and setting an according value for P to maximize Max .

$$Max = \frac{\log(1 - FPS^{1/k})}{\log\left(\frac{1}{1 + \frac{1}{P(\frac{1}{k} - \frac{1}{m})}}\right)}$$

As P can not be larger than Max , we can now see that for any given FPS , m , and k , $P = m$ maximizes the equation. This means that decrementing every cell in each iteration leads to the largest amount of messages we can store in the SBF while keeping the other variables fixed. We have therefore reached the conclusion that for duplicate detection that does not allow for false negatives, the ideal SBF, providing the best conditions, is a special case SBF with $P = m$, which is also known as the decaying Bloom filter we briefly mentioned earlier.

Quantitatively evaluating our three scenarios requires one further step. We have shown the ideal value for P of $P = m$ for a fixed m , but we still need to find the value for m in our available memory. The amount of bits for each scenario calculated above still relates to the idea that every message during an SA double rotation should fit in the SBF. As we moved away from this idea we can now allocate less bits that support a smaller value for Max but better utilize available memory by having more cells over all. The ideal amount of bits to allocate per cell could be calculated mathematically, but in this work, a script has been utilized that brute forces the solution by iterating over all possible bit values, deriving the number of possible cells for the available memory, and setting $m = P$ accordingly. [pbc] Through this approach we arrive at the respective values for each scenario of only 13 supported messages for scenario 1 with 4 bits per cell and $k = 13$, 175 supported messages for scenario 2 with 8 bits per cell and $k = 13$ and 20862 supported messages for scenario 3 with 15 bits per cell and $k = 13$. A set $FPS = 0.0001$ has been utilized for all scenarios to be in line with our requirements. Allowing for $FPS = 0.001$ does not have any meaningfully impact scenario 1 and 2 but increases Max to 27820 in scenario 3 with 15 bits per cell and $k = 10$.

Having these values, one major obstacle still remains. Having only 5.5k messages inside the sliding window is just not nearly enough to provide meaningful coverage. For example, the 5Gbit/s connection from scenario 3 could send out the 4601 packets that fit in the SBF from scenario 3 in $\frac{20862*458*8bit}{5Gbit/s} \approx 0.015s$. Having a sliding window that covers only the last 15ms when sending out packets back to back is not going to provide practical coverage against replay detection.

For our last part on the SBF, we are going to take a reverse approach and calculate how much memory would be required to provide a sliding window large enough to serve as a practical line of defense against replay attacks and how we should handle packets that fall outside of this sliding window.

3.3.3 Sliding window cutoff

In our desire to provide protection against replay attacks we have shown that the SBF can provide duplicate detection for a limited amount of messages inside a sliding window. We have shown that this window will be too small for any practical application at our projected available memory but in addition to that another, previously ignored factor has to be taken into account that limits the usefulness of the SBF. Messages that leaves this sliding window will not be part of the set anymore and therefore can not be detected as a duplicate when it reoccurs later in time. This would have the severe implication that an attacker would only have to wait until *Max* messages have passed after capturing a packet so that the sliding window does not cover their captured message anymore. Alternatively an attacker could even cause an increase in traffic themselves in order to fill up the SBF with new items if they have at least *Max* old captured messages that are not part of the SBFs sliding window. If the replay protection only works within a very limited time frame it would be practically useless but if we can find a way of dealing with packets arriving that are not covered by the sliding window we could effectively grand replay protection for the entire Key lifetime and even a double rotation. Since we showed that adding more memory is no feasible way of tackling this problem a solution could be to simply discard every packet that is too old to be covered by the SBF. Every message that is more than *Max* messages behind our latest added one will be treated as if it was a duplicate and thrown away. An important aspect of this approach is that if our sliding window is too small, out of order traffic would routinely be dropped and could cause a significant increase of false positives beyond our set *FPS*.

Since the PSP protocol does not include sequence numbers it is also not possible to find out the exact amount of packets between two packets without counting them exactly. We could, however, utilize one of its header fields to get an estimate. The PSP architecture specifications describe the process for the IV generation, which must be guaranteed to be unique within an SA lifetime. IVs are created by the NIC by combining a 64 bit timestamp from a picosecond counter that the NIC is required to implement as well as the 32 bit SPI. Another interesting aspect about the IV is that it can be passed to the operating system, making it an option to be considered when thinking about a userspace implementation. [psp_arch_spec] Using this information about packets to our advantage we can simply set a time in which incoming packets should still be accepted for testing against the SBF. Upon inserting a new packet in the SBF its timestamp is saved overwriting the previously latest

timestamp, which would only require 64 bit more memory. On arrival, a packet's timestamp can then simply be checked against the saved timestamp. In the case that it is too old it can be discarded without testing for membership in the SBF as it would have been dropped out of the sliding window by the time of its arrival anyway. It is very important to pick the sliding window size generously enough to not interfere with traffic that is delayed by normal effects inside the network like routing delays or network congestion. Packets that arrive within the chosen time limit are to be tested for membership in the SBF window. Ideally this approach does not increase the amount of false positives as all packets that are discarded for timing reasons are not relevant anymore as their content was already retransmitted or they were actually replayed with malicious intent.

First, we would like to establish a baseline for traffic that could be expected to arrive within a time limit. If we set a sliding window for the size of 10 seconds in which traffic is going to be checked against the data inside the SBF to detect duplicates, we would require at most $Max = \frac{5Gbit/s * 10s}{458 * 8bit * 10} \approx 1.36 * 10^6$ for covering a continuous transmission of IMIX packets. Using the by now well known FPS formula from above we can simply set $k = 8$ to arrive at $m = 2.8610^7$. Since this value is the number of required cells, it has to be multiplied with the number of bits per cell $\log_2(1.36 * 10^6) = 21$ for a total of $\approx 6 * 10^8 bits$ or $75MB$ that allows for up to $\approx 1.5 * 10^6$ messages in the sliding window.

However, this approach might be naïve. Picking a time limit should be based on the highest amount of traffic that could arrive within it to match our calculated Max . Otherwise, an attacker could try to flood the network with traffic to fill up the SBF with more messages than anticipated, causing a higher FPR for a denial or degradation of service attack. Performing the same steps again with the minimal 64Byte packet we established in scenario 1 for the full 10Gbit/s link from scenario 3 we model a scenario where an attacker is able to fill up the entire bandwidth with minimal packets which should serve more as an illustration than a real attack against the SBF that would at this point not be the bottleneck for the DoS attack anymore as no real traffic would arrive anyway. In this scenario we calculate $Max = \frac{10Gbit/s * 10s}{64 * 8bit * 10} \approx 1.95 * 10^7$ for an $m \approx 4.10 * 10^8$ with $25bit$ cells and arrive at $1.28GB$ that is able to handle $21 * 10^6$ Messages. It is unlikely that this amount of NIC memory would be dedicated to protecting one simplex connection against replay attacks and even in a userspace implementation over $1GB$ of RAM is not likely going to be spared for such a task, especially given the alternative options.

Looking at the lower end $75MB$ of memory for ensuring absolutely no false negatives in the given time frame could be a viable option depending on the use-case. The requirements for these individual scenarios and what constitutes acceptable performance and expendable amount of memory can of course not be generalized in this work. With these findings it should be possible to make a much more educated decision when deciding whether a Bloom filter is going to be an adequate tool for providing replay detection.

4 Discussion

With these somewhat sobering results we can say that the original goal of utilizing a Bloom filter to bring replay detection to the PSP protocol does not have been achieved. For special networking environments that provide the necessary resources, Bloom filter-based replay detection was shown to be achievable. The usefulness of such an endeavor is, of course, up for decision with respect to the individual use-case. Especially the possibility to utilize a 100MB SBF in a user- or kernel-space implementation utilizing the much less restricted RAM of the server itself instead of the expensive and limited NIC Ram could be of interest. In such an implementation even the IV of PSP could be utilized as the NIC is required to provide a way of passing it to the receiver of the message as mentioned in the standard. [25] The possibility of a hardware implementation might also be of interest in cases where specialized hardware is built, for example, through the use of an FPGA board. [36] [28] For such a use case, the required memory could also be acquired as needed to cover the demand of the respective environment for the application.

Picking hash functions in software or hardware should prioritize hash functions that produce uniform outputs to avoid unnecessary false positives. Speed is also of concern for this choice, as having slow hash functions will slow down the otherwise very fast Bloom filter. Ideally, the hashing process can happen in parallel and produces perfectly uniform outputs. Since the Bloom filter does bring redundancy through using not only one but k hash functions, shortcomings in uniformity can be somewhat compensated by the Bloom filter.

In PSP, the encrypted payload or the IV field in the header could be utilized for hashing via the k hash functions. An important aspect of this is that the hashed part of a packet should be unbiased and uniform. Biases in the input data could also lead to the hash functions producing an increased rate of false positives resulting in a higher packet loss rate. As any duplicate in the hashed data will inevitably lead to the packet being dropped extra care should be taken to avoid this case. In the case that some class of packets has the problem of always having the exact same data hashed only the first of them would arrive with all other, within the sliding window, being dropped, possibly even including their retransmissions.

While this work laid a huge focus on the PSP protocol the Bloom filter is not dependent on the protocol it could provide replay detection for all sorts of networking protocols TCP and UDP directly without the PSP encapsulation but also other new protocols that could serve as direct PSP replacements like Wrapped ESP. [32] One interesting feature of Wrapped ESP, that at this point is still a work in progress, is the presence of sequence numbers that PSP does not provide. With packets being numbered their relative position in the data stream could easily be calculated which would help greatly in terms of calculating the sliding window size through the *Max* parameter. Also, the

cut-off for packets that fall out of the sliding window could be simply set to the delta between the last accepted packet number and Max instead of having to set a time limit and work through the expected or maximum packet rate that could occur within a set time frame. This would directly save on memory required to grant that an attacker can not fill up the sliding window with very small or even minimal packets that have to be expected within the time frame but normally do not occur.

It should not go unmentioned that our approach to prevent replay attacks is only able to prevent attacks through previously encountered packets. A possible circumvention for an attacker with control over the flow of packets inside the network would be to steal packets in a way that does not only copy their data but prevents them from arriving in general. With such capabilities the attacker would prevent the Bloom filter from ever encountering their stolen packet before the so called preplay attack is carried out. [4] As single packets are not likely to spark suspicion, especially with a Bloom filter in place that is known to occasionally cause false positives. Depending on the underlying protocol in use, a retransmission might be carried out that resends the same payload as before but produces a different packet that does not have to be detected as a duplicate. Against such an attack our Bloom filter based approach is powerless but for our SBF based approach packets that fall outside the sliding window would be discarded if too much time passed between the initial sending of the packet and the attack. Packets that are new enough to still fall inside the sliding window, however, can not be detected. Taking these limitations of the Bloom filter into account is another important aspect to think about when considering utilizing it in replay detection.

Finally we have to conclude that the Bloom filter with its unique and interesting approach is not the best choice for replay detection. The amounts of memory it requires to become a useful addition to network security are too large for being applied in large scale and other approaches like modifications in the application layer are over all a better suited solution for most cases.

5 Conclusion

In this work we took a close look at the different Bloom filter variations and how their respective qualities might help in bringing replay detection to the PSP protocol. We explained different networking scenarios as well as the PSP protocol itself in order to quantify and narrow down a possible solution that we then tried to provide by testing the different aspects of both the standard Bloom filter as well as the stable Bloom filter. As a basis for our evaluation we outlined 3 different scenarios with respect to network bandwidth, SA lifetime and available memory. We argued that a Bloom filter should only create a false positive rate in the range of 0.1% – 0.01% as to not cause too much interference with TCPs congestion control. Trying to find parameters that are able to provide the desired performance we have now arrived at the conclusion that the standard Bloom filter is not cut out for environments in which available memory is a strictly limited resource. Its claim of memory efficiency is justified but is not able to provide the desired performance with the low amounts of memory available in a NIC. Out of our three scenarios we used, only the most relaxed scenario 3 could give us an idea of what amounts of memory would be required that are able to store an entire SAs lifetime worth of traffic inside a single Bloom filter. With amounts of RAM in the kilobytes no adequate performance is possible with respect to the required amounts of messages and our set false positive rate. The analysis of the Bloom filters various parameters did however provide valuable insights on the mechanics and probabilistic nature of this data structure that was able to provide a much better view on the topic.

Following up with the stable Bloom filter, we could utilize these findings in our analysis. The SBF is overall a much better fit for handling streaming data and impresses with its ability to provide a fixed false positive rate that can be set much more flexible compared to the standard Bloom filter. Through its approach of a sliding window and decaying parameters, the SBF provides desirable qualities that we tried to use by methodically tuning the parameters to the respective scenarios. This resulted in an understanding of the interplay between the different parameters and gave us the ability to find an optimal balance with respect to almost all of our outlined requirements. We tried to achieve very strict false positive rates that ideally do not surpass the threshold of 0.01% we set to minimize the impact on tcp's congestion control that could otherwise throttle throughput as tcp would otherwise interpret dropped packets caused by false positives in our replay detection as a sign for network congestion. By closely examining the impact of false negatives on our scenario, we showed that with respect to the security aspect, false negatives can not be accepted without risking our Bloom filter-based replay protection to be circumvented by an attacker who is able to overload the SBF. Identifying the important role of the Max and P parameters for false positives and false negatives, we tuned the parameters accordingly to achieve a solution that is more in line with a decaying Bloom filter than an SBF since we decided to decrement every cell at every iteration. Setting the

SBF parameters resulted in a performance that, although generally in line with our requirements in all other regards, still requires memory far beyond the outlined scenarios.

The initially set performance goals derived from Google PSP architecture specifications could not be met. With their goal of minimizing NIC memory requirements the conscious decision has been made to not provide protection against replay attacks and leave this task to application layer protocols. A Bloom filter is not able to fill this gap without sacrificing the initial goal of providing almost stateless performance.

For other use-cases with a lot of available memory in the hundreds of megabytes especially the stable Bloom filter could be an interesting solution for detecting duplicates. Especially when paired with a way of determining a packets age like PSPs initialization vector (IV) or sequence numbers in a protocol that supports them the coverage of the sliding window can be extended to provide protection against replayed traffic by discarding all packets that are not covered by the sliding window any more. We have shown that this way we are able to cover the entirety of an SA double rotation at a set rate of false positives and without allowing for false negatives. These qualities could be utilized in a data center with few open connections and specialized hardware that does not put harsh restrictions on memory and needs to ensure full protection against replay attacks.

Bibliography

- [1] Paulo Sérgio Almeida et al. “Scalable Bloom Filters”. In: *Information Processing Letters* 101.6 (Mar. 2007), pp. 255–261. ISSN: 0020-0190. URL: <https://www.sciencedirect.com/science/article/pii/S0020019006003127>.
- [2] *Amazon Redshift now leverages Bloom filters to improve data lake query performance by up to 2x* — [aws.amazon.com](https://aws.amazon.com/about-aws/whats-new/2020/05/amazon-redshift-now-leverages-bloom-filters-to-improve-data-lake-query-performance/). <https://aws.amazon.com/about-aws/whats-new/2020/05/amazon-redshift-now-leverages-bloom-filters-to-improve-data-lake-query-performance/>. [Accessed 04-09-2024].
- [3] AMD. *AMD UltraScale+ FPGAs*. <https://docs.amd.com/v/u/en-US/ultrascale-plus-fpga-product-selection-guide>. [Accessed 04-09-2024].
- [4] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. 1st. USA: John Wiley & Sons, Inc., 2001. ISBN: 0471389226.
- [5] *Announcing PSP Security Protocol is now open source* | *Google Cloud Blog* — [cloud.google.com](https://cloud.google.com/blog/products/identity-security/announcing-psp-security-protocol-is-now-open-source). <https://cloud.google.com/blog/products/identity-security/announcing-psp-security-protocol-is-now-open-source>. [Accessed 04-09-2024].
- [6] Rama Bansode and Anup Girdhar. “Common Vulnerabilities Exposed in VPN – A Survey”. In: *Journal of Physics: Conference Series* 1714.1 (Jan. 2021), p. 012045. DOI: 10.1088/1742-6596/1714/1/012045. URL: <https://dx.doi.org/10.1088/1742-6596/1714/1/012045>.
- [7] Erik Bernhardsson. *Ping the world*. <https://erikbern.com/2015/04/26/ping-the-world>. [Accessed 04-09-2024].
- [8] *Birthday Paradox* — [link.springer.com](https://link.springer.com/referenceworkentry/10.1007/978-1-4419-5906-5_440). https://link.springer.com/referenceworkentry/10.1007/978-1-4419-5906-5_440. [Accessed 04-09-2024].
- [9] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. *TCP Congestion Control*. RFC 5681. Sept. 2009. DOI: 10.17487/RFC5681. URL: <https://www.rfc-editor.org/info/rfc5681>.
- [10] Yanpei Chen et al. “Understanding TCP incast throughput collapse in datacenter networks”. In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*. WREN '09. Barcelona, Spain: Association for Computing Machinery, 2009, pp. 73–82. ISBN: 9781605584430. DOI: 10.1145/1592681.1592693. URL: <https://doi.org/10.1145/1592681.1592693>.

-
- [11] Kai Cheng, Limin Xiang, and M. Iwaihara. “Time-decaying Bloom Filters for data streams with skewed distributions”. In: *15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications (RIDE-SDMA’05)*. Apr. 2005, pp. 63–69. DOI: 10.1109/RIDE.2005.15.
- [12] Eirini Chioti et al. “Bloom Filters for Efficient Coupling Between Tables of a Database”. In: *Engineering Applications of Neural Networks*. Ed. by Giacomo Boracchi et al. Cham: Springer International Publishing, 2017, pp. 596–608. ISBN: 978-3-319-65172-9.
- [13] Google Cloud. *PSP Packet. 2_PSP_Security_Protocol.max-2000x2000.jpg*. 2024.
- [14] Google Cloud. *PSP Stateless mode*. <https://cloud.google.com/blog/topics/security/using-psp-security-protocol-securely-cloud>. 2024.
- [15] *Cloud Bigtable improves single-row read throughput by up to 50 percent | Google Cloud Blog — cloud.google.com*. <https://cloud.google.com/blog/products/databases/cloud-bigtable-improves-single-row-read-throughput-by-up-to-50-percent>. [Accessed 04-09-2024].
- [16] Thomas H Cormen and Charles E Leiserson. *Introduction to Algorithms, fourth edition*. en. London, England: MIT Press, Apr. 2022.
- [17] Fan Deng and Davood Rafiei. “Approximately detecting duplicates for streaming data using stable bloom filters”. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’06. Chicago, IL, USA: Association for Computing Machinery, 2006, pp. 25–36. ISBN: 1595934340. DOI: 10.1145/1142473.1142477. URL: <https://doi.org/10.1145/1142473.1142477>.
- [18] Sarang Dharmapurikar et al. “Deep packet inspection using parallel bloom filters”. In: *IEEE Micro* 24.1 (2004), pp. 52–61. DOI: 10.1109/MM.2004.1268997.
- [19] docs.python.org. *hashlib docs*. <https://docs.python.org/3/library/hashlib.html>. [Accessed 04-09-2024].
- [20] Reda El Abbadi and Hicham Jamouli. “Takagi–Sugeno Fuzzy Control for a Nonlinear Networked System Exposed to a Replay Attack”. In: *Mathematical Problems in Engineering* 2021.1 (2021), p. 6618105. DOI: <https://doi.org/10.1155/2021/6618105>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1155/2021/6618105>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1155/2021/6618105>.
- [21] *Encryption in transit | Documentation | Google Cloud — cloud.google.com*. <https://cloud.google.com/docs/security/encryption-in-transit>. [Accessed 04-09-2024].
- [22] Pietro Ferrara et al. “Static analysis for discovering IoT vulnerabilities”. In: *International Journal on Software Tools for Technology Transfer* 23.1 (Feb. 2021), pp. 71–88. ISSN: 1433-2787. DOI: 10.1007/s10009-020-00592-x. URL: <https://doi.org/10.1007/s10009-020-00592-x>.
- [23] Perry Gentry. “What is a VPN?” In: *Information Security Technical Report* 6.2 (Mar. 2001), pp. 15–22. DOI: 10.1016/S1363-4127(01)00103-0.

-
- [24] Shahabeddin Geravand and Mahmood Ahmadi. “Bloom filter applications in network security: A state-of-the-art survey”. In: *Computer Networks* 57.18 (Dec. 2013), pp. 4047–4064. ISSN: 1389-1286. URL: <https://www.sciencedirect.com/science/article/pii/S1389128613003083>.
- [25] google. *PSP Architecture Specifications*. 2022. URL: <https://github.com/google/psp>.
- [26] Kiran Gopinathan and Ilya Sergey. “Certifying Certainty and Uncertainty in Approximate Membership Query Structures”. In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2020, pp. 279–303. ISBN: 978-3-030-53291-8.
- [27] Deke Guo et al. “False Negative Problem of Counting Bloom Filter”. In: *IEEE Trans. on Knowl. and Data Eng.* 22.5 (May 2010), pp. 651–664. ISSN: 1041-4347. DOI: 10.1109/TKDE.2009.209. URL: <https://doi.org/10.1109/TKDE.2009.209>.
- [28] Jared Harwayne-Gidansky, Deian Stefan, and Ishaan Dalal. “FPGA-based SoC for real-time network intrusion detection using counting bloom filters”. In: *IEEE Southeastcon 2009*. Mar. 2009, pp. 452–458. DOI: 10.1109/SECON.2009.5174096.
- [29] Russ Housley. *Using the AES-GMAC Algorithm with the Cryptographic Message Syntax (CMS)*. RFC 9044. June 2021. DOI: 10.17487/RFC9044. URL: <https://www.rfc-editor.org/info/rfc9044>.
- [30] Alefiya Hussain, John Heidemann, and Christos Papadopoulos. “A framework for classifying denial of service attacks”. In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM’03. Karlsruhe, Germany: Association for Computing Machinery, 2003, pp. 99–110. ISBN: 1581137354. DOI: 10.1145/863955.863968. URL: <https://doi.org/10.1145/863955.863968>.
- [31] Lars-Erik Jonsson et al. *The Lightweight User Datagram Protocol (UDP-Lite)*. RFC 3828. July 2004. DOI: 10.17487/RFC3828. URL: <https://www.rfc-editor.org/info/rfc3828>.
- [32] Steffen Klassert and Antony Antony. *Wrapped ESP Version 2*. Internet-Draft draft-klassert-ipsecme-wespv2-01. Work in Progress. Internet Engineering Task Force, June 2024. 15 pp. URL: <https://datatracker.ietf.org/doc/draft-klassert-ipsecme-wespv2/01/>.
- [33] Myeong-Hyeon Lee and Yoon-Hwa Choi. “A Fault-Tolerant Bloom Filter for Deep Packet Inspection”. In: Jan. 2008, pp. 389–396. ISBN: 0-7695-3054-0. DOI: 10.1109/PRDC.2007.30.
- [34] Taeho Lee et al. “The Case for In-Network Replay Suppression”. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ASIA CCS’17. Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2017, pp. 862–873. ISBN: 9781450349444. DOI: 10.1145/3052973.3052988. URL: <https://doi.org/10.1145/3052973.3052988>.

-
- [35] Taewhi Lee, Kisung Kim, and Hyoung-Joo Kim. “Join processing using Bloom filter in MapReduce”. In: *Proceedings of the 2012 ACM Research in Applied Computation Symposium*. RACS '12. San Antonio, Texas: Association for Computing Machinery, 2012, pp. 100–105. ISBN: 9781450314923. DOI: 10.1145/2401603.2401626. URL: <https://doi.org/10.1145/2401603.2401626>.
- [36] Michael J. Lyons and David Brooks. “The design of a bloom filter hardware accelerator for ultra low power systems”. In: *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED '09. San Francisco, CA, USA: Association for Computing Machinery, 2009, pp. 371–376. ISBN: 9781605586847. DOI: 10.1145/1594233.1594330. URL: <https://doi.org/10.1145/1594233.1594330>.
- [37] Sreekanth Malladi, Jim Alves-Foss, and Robert Heckendorn. “On Preventing Replay Attacks on Security Protocols”. In: *Proc. International Conference on Security and Management* (June 2002).
- [38] *Manipulator-in-the-middle attack* | OWASP Foundation — [owasp.org](https://owasp.org/www-community/attacks/Manipulator-in-the-middle_attack). https://owasp.org/www-community/attacks/Manipulator-in-the-middle_attack. [Accessed 04-09-2024].
- [39] Matthew Mathis et al. “The macroscopic behavior of the TCP congestion avoidance algorithm”. In: *SIGCOMM Comput. Commun. Rev.* 27.3 (July 1997), pp. 67–82. ISSN: 0146-4833. DOI: 10.1145/263932.264023. URL: <https://doi.org/10.1145/263932.264023>.
- [40] Aleksandr Matrosov. *Stuxnet Under the Microscope - Revision 1.1*. Accessed: 2024-09-04.
- [41] Loizos Michael et al. “Improving distributed join efficiency with extended bloom filter operations”. In: *Proceedings of the 21st International Conference on Advanced Networking and Applications*. AINA '07. USA: IEEE Computer Society, 2007, pp. 187–194. ISBN: 0769528465. DOI: 10.1109/AINA.2007.80. URL: <https://doi.org/10.1109/AINA.2007.80>.
- [42] Microsoft Threat Intelligence Microsoft Incident Response. *criminal actor targeting organizations for data exfiltration and destruction* Microsoft Security Blog [microsoft.com](https://www.microsoft.com/en-us/security/blog/2022/03/22/dev-0537-criminal-actor-targeting-organizations-for-data-exfiltration-and-destruction/). <https://www.microsoft.com/en-us/security/blog/2022/03/22/dev-0537-criminal-actor-targeting-organizations-for-data-exfiltration-and-destruction/>. [Accessed 04-09-2024].
- [43] *Mitre Stuxnet* — [attack.mitre.org](https://attack.mitre.org/versions/v15/software/S0603/). <https://attack.mitre.org/versions/v15/software/S0603/>. [Accessed 04-09-2024].
- [44] *Multi-Factor Authentication Interception, Technique - Enterprise* – [attack.mitre.org](https://attack.mitre.org/versions/v15/techniques/T1111/). <https://attack.mitre.org/versions/v15/techniques/T1111/>. [Accessed 04-09-2024].
- [45] Hossein Pishro-Nik. *Law of Large Numbers*. https://www.probabilitycourse.com/chapter7/7_1_1_law_of_large_numbers.php. [Accessed 04-09-2024].
- [46] Felix Putze, Peter Sanders, and Johannes Singler. “Cache-, hash-, and space-efficient bloom filters”. In: *ACM J. Exp. Algorithmics* 14 (Jan. 2010). ISSN: 1084-6654. DOI: 10.1145/1498698.1594230. URL: <https://doi.org/10.1145/1498698.1594230>.

-
- [47] R. L. Rivest, A. Shamir, and L. Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: <https://doi.org/10.1145/359340.359342>.
- [48] Joseph A. Salowey, David McGrew, and Abhijit Choudhury. *AES Galois Counter Mode (GCM) Cipher Suites for TLS*. RFC 5288. Aug. 2008. DOI: 10.17487/RFC5288. URL: <https://www.rfc-editor.org/info/rfc5288>.
- [49] Karen Seo and Stephen Kent. *Security Architecture for the Internet Protocol*. RFC 4301. Dec. 2005. DOI: 10.17487/RFC4301. URL: <https://www.rfc-editor.org/info/rfc4301>.
- [50] Robert W. Shirey. *Internet Security Glossary, Version 2*. RFC 4949. Aug. 2007. DOI: 10.17487/RFC4949. URL: <https://www.rfc-editor.org/info/rfc4949>.
- [51] Sadhan Sood and Dmitri Loguinov. “Probabilistic near-duplicate detection using simhash”. In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. CIKM ’11. Glasgow, Scotland, UK: Association for Computing Machinery, 2011, pp. 1117–1126. ISBN: 9781450307178. DOI: 10.1145/2063576.2063737. URL: <https://doi.org/10.1145/2063576.2063737>.
- [52] David Starobinski, Ari Trachtenberg, and Sachin Agarwal. “Efficient PDA Synchronization”. In: *IEEE Transactions on Mobile Computing* 2.1 (Jan. 2003), pp. 40–51. ISSN: 1536-1233. DOI: 10.1109/TMC.2003.1195150. URL: <https://doi.org/10.1109/TMC.2003.1195150>.
- [53] *Steal or Forge Kerberos Tickets: Kerberoasting, Sub-technique Enterprise — attack.mitre.org*. <https://attack.mitre.org/versions/v15/techniques/T1558/003/>. [Accessed 04-09-2024].
- [54] S. Joshua Swamidass and Pierre Baldi. “Mathematical Correction for Fingerprint Similarity Measures to Improve Chemical Retrieval”. In: *Journal of Chemical Information and Modeling* 47.3 (2007). PMID: 17444629, pp. 952–964. DOI: 10.1021/ci600526a. eprint: <https://doi.org/10.1021/ci600526a>. URL: <https://doi.org/10.1021/ci600526a>.
- [55] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. “Theory and Practice of Bloom Filters for Distributed Systems”. In: *IEEE Communications Surveys Tutorials* 14.1 (First 2012), pp. 131–155. ISSN: 1553-877X. DOI: 10.1109/SURV.2011.031611.00024.
- [56] *Transmission Control Protocol*. RFC 793. Sept. 1981. DOI: 10.17487/RFC0793. URL: <https://www.rfc-editor.org/info/rfc793>.
- [57] *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: <https://www.rfc-editor.org/info/rfc768>.
- [58] Lukas Werthmanns. *Bloom filter script for evaluation*. code/bf.
- [59] Lukas Werthmanns. *development of alpha*. code/alpha.py. alpha_graph_1/graph-alpha_plot-m=1000000_n=95000-reps=1-steps=False-step_size=100.svg.
- [60] Lukas Werthmanns. *development of alpha 2*. code/alpha.py. alpha_graph_2/graph-alpha_plot-m=10000000_n=950000-reps=1-steps=False-step_size=10000.svg.

-
- [61] Lukas Werthmanns. *fps plot*. `code/fps_plot.py`. `img/sbf_P_comparison_k=8/graph-fpr_plot-m=1000000000_MAX=10_K=8_P=1_to_201.svg`.
- [62] Lukas Werthmanns. *fps plot*. `code/fps_k_plot.py`. `good_graphs_to_use2/sbf_k_comparison_k=8/graph-fpr_k_plot-m=1000000000_MAX=10_P=143_K=1_to_101.svg`.
- [63] Lukas Werthmanns. *m n relationship*. `code/m_n_plot.py`. `img/m_n_steps/graph-m_n_relationship-k=7-a=0.01-reps=1-steps=True-step_size=1000.svg`.
- [64] Lukas Werthmanns. *m n relationship - boxplot*. `code/m_n_plot.py`. `img/m_n_boxplot/graph-m_n_relationship-k=7-a=0.01-reps=10-steps=True-step_size=100.svg`.
- [65] Lukas Werthmanns. *mathis plot*. `code/calculate_avail_bandwidth.py`.
- [66] Wen Xia et al. “A Comprehensive Study of the Past, Present, and Future of Data Deduplication”. In: *Proceedings of the IEEE* 104.9 (2016), pp. 1681–1710. DOI: 10.1109/JPROC.2016.2571298.
- [67] MyungKeun Yoon. “Aging Bloom Filter with Two Active Buffers for Dynamic Sets”. In: *IEEE Transactions on Knowledge and Data Engineering* 22.1 (Jan. 2010), pp. 134–138. ISSN: 1558-2191. DOI: 10.1109/TKDE.2009.136.