Optimization of Black Box Pollution Attacks for Bloom Filters

Pablo Sayans
Universidad Carlos III de Madrid
Madrid, Spain

***Abstract*—Bloom filters are extensively used in a wide number of applications in the field of computer science. From Internet devices like routers or firewalls to spam filters in mail services. As such, they are also a target of cyberattacks to cause Denial of Service attacks and other types of malicious activities. To perform these attacks, it is necessary to pollute the filter. In this work we discuss existing pollution attacks to Bloom filters and ways to improve a recently presented black box attack algorithm-. This attack tests several elements for each insertion and adds the best candidate to the filter, maximizing in every round the number of bits set to 1. We will perform different tests to find an adequate parametrization of the algorithm to accelerate the pollution of Bloom filters. We will also introduce new variants of this algorithm. One of them will seek to reduce the number of tested elements each round, reducing the execution time. The other variant of the attack will try to predict the configuration of the filter after a determined number of rounds or when reaching certain False Positive Probability (FPP), customizing the attack from this point to that configuration to reduce execution time. The results for the parametrization and the first variant have been positive, reducing the execution time. The second variant of the attack offered mixed results due to the lack of precision when measuring the FPP in some test scenarios.**

Keywords—Bloom filter, Security, Pollution attacks

I. INTRODUCTION

Burton Howard Bloom developed in 1970 a space-efficient probabilistic data structure that allowed to determine the membership of an element to a set [1]. This data structure was later named after its creator. From then on, it has been used in a variety of applications in the field of computer science, being especially critical in the correct functioning of some Internet infrastructure devices [2].

Bloom filters are probabilistic filters. When an element is checked in the filter, it can assure that an element is not part of the set or tell that with high probability an element is part of the set [1]. This means that false negatives are not possible, but there is a certain probability of having false positives when querying the filter. For this reason, Bloom filters are typically used as a pre-check of the membership of an object to the set. If the element returns a negative result, it is certain the element is not part of the set. On the contrary, if the filter returns a positive value, there is a high probability of the element being part of the set, but we cannot assure the element is indeed a member of the set. There is a small probability of that element to be a collision with other existing elements, and thus a false positive. For that reason, when there is a match, the element is checked against an upper level filter which can infer without doubt whether the element is part of the set or not. In other words, Bloom filters are very useful to discard elements before performing the real membership check.

The problem with probabilistic filters is, the larger the amount of matches in the Bloom filter, the higher the chances of collisions [3] and the larger the amount of queries running against upper-level tables. Those tables are slower and more demanding in terms of computing resources and time, so if a lot of queries are simultaneously performed, it could result in an interruption of the service. They are also used as direct filters without further layers [2],[4], so a filter that returns too many false positives will cause a malfunctioning of the infrastructure, since it will return a positive match for most of the performed queries and a Denial of Service attack would be equally possible [5].

Here is when the security aspects come to play [6]. If the filter gets polluted, all queries will result in a positive result, and thus the upper level filter will always be queried. To prevent this, filters are carefully configured based on the number of elements that will ultimately be inserted, avoiding filling the filter. The problem comes when an attacker can intentionally insert more elements that the ones the filter was configured to store in the first place. This pollution will end up filling the filter and thus making the upper levels of the system to work all the time, causing a denial of service attack.

As it was previously stated, Bloom filters have been used for many applications. Traditionally, some of their main ones are network and security appliances, such as firewalls [7], multi-level switches or routers [8],[9]. They are used, for example, to distribute multicast packets through the network [10]. These devices use hardware-based implementations of Bloom Filters to accelerate the computation.

Bloom Filters are also used in software-based applications. An example of this are some web proxies which use them to check if a webpage is cached or not. The well-known open source Squid proxy makes use of them [4]. Another example are spam email blacklists [11],[12]. In fact, this can be used by the attacker to send an enormous amount of emails between two accounts of his ownership through a relay that uses Bloom Filters to check the state of the filter. If the mails do not pass through the filter, it is considered a positive. The same applies to received mails, which are considered negatives. They are also used in databases and Big Data environments [13]. For example, in the Redis database they are recommended to check in an application if a username is already taken [14].

An attacker able to find effective ways to pollute the Bloom filter running behind these infrastructures can conduct Denial of Service (DoS) attacks. The attack itself might not be provoked by the act of filling the Bloom Filter, which also happens, but by removing the capabilities of the filter to discard results before querying the real database or any other existing data structures in upper levels in environments stressed by a lot of simultaneous queries.

The paper is organized as follows. A brief description of Bloom filters and counting Bloom filters is given in Section II. Existing attacks over the filters are explained in section III. The lookup attack that is studied and optimized in this paper is described in section IV. Section V is reserved for explaining the developed testing environment. Optimizations of the regular lookup attack are discussed in Section VI. The new proposed pollution attacks are presented from Section VII to Section VIII. Finally, we analyze the obtained results in Section IX and conclude in Section X with possible future works.

II. Bloom filters and Counting Bloom filters

The Bloom filter consists of a vector of bits and several hashing functions. The filter is originally configured with M positions of bits and K hash functions. All the M bits are set to 0. When an element is inserted in the filter, K number of hash functions are applied to the element. Each of these functions returns a position of the filter, which is set to 1. This is shown in Figure 1. In a similar way, when the filter is queried, the K hash functions are again applied to the searched element, returning K positions of the filter. A positive is produced if all these positions are set to 1. On the contrary, if any of the positions returned by the hashing functions is equal to 0, the element is not a member of the set and a negative result is returned.

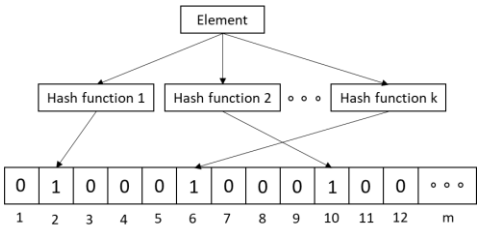


Fig. 1. Example of a regular Bloom filter

As it was stated at the beginning of the paper, Bloom filters have a low probability of returning a false positive, which means the queried element matches the same positions in the filter as other inserted elements. Obviously, any time an element is inserted in the filter the number of bits set to 1 increase and the False Positive Probability (FPP) also increases. This probability is measured considering the size of the filter (M), the number of hash functions used (K) and the number of elements already inserted (N). The formula is as follows [15]:

$$FPP \approx \left(1 - \left(\frac{m-1}{m}\right)^{n \cdot k}\right)^{k}$$

Typically, M has large values as small filters are not very helpful, so the formula can be approximated by:

$$FPP \approx \left(1 - e^{-\frac{k \cdot n}{m}}\right)^{k}$$

A problem with regular Bloom filters is they allow the addition of elements, but they do not allow their removal. The vector is made of bits, and thus two or more elements may share a position of the vector, if the position is set to 0 to remove an element, we are also removing all the elements that share that position with the deleted element. For this reason, a variant of regular Bloom filters was invented called Counting Bloom Filters (CBF). CBFs have a counter in each of the positions of the vector, instead of bits. When an element is inserted in the filter, the K functions are again executed, returning each a vector position, and incrementing all of them by 1. This can be seen in Figure 2. When an element is fetched, the K functions are once more executed, returning each K positions of the vector. If all the positions have values equal or bigger than 1, it will result in a positive result. If any of the positions is equal to 0 the element does not exist in the filter and thus it returns a negative result. Finally, when an element is removed out of the filter, the K functions are executed, they return K positions of the filter and each of these positions is decremented by 1 in case of being greater than 0.

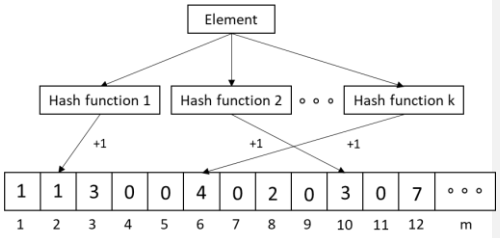


Fig. 2. Example of a counting Bloom filter

The FPP for CBF is the same as the one for regular BF [3]:

$$FPP \approx \left(1 - e^{-\frac{k \cdot n}{m}}\right)^k$$

For the pollution algorithms used in this work, it will be necessary to remove elements from the filter. For this reason, all the tests will be performed using Counting Bloom Filters instead of regular Bloom Filters.

III. Existing attacks on Bloom filters

A. White box and black box filters

As we have seen before, Bloom filters are used in critical components of the Internet infrastructure and a lot of software applications, as they are a valuable resource. As such, they are also an interesting target for cybercriminals. A lot of attempts have been made to perform effective attacks on these data structures. Overall, two categorizations of attacks can be done. First, there are those attacks whose reliability is based on the previous knowledge of the filter implementation [5]. This scenario can be the most dangerous. Many developers implement Bloom filters using non-cryptographic hash functions, as this makes it less costly. It also makes the output of the hash predictable and thus the positions of the filter that will be incremented. Some attacks exploit this fact to forge an effective load that fills the filter or set to 1 specific positions that generate collisions with a particular element [10]. Another type of attacks does not assume anything about the filter. In this scenario, the pollution is performed by analyzing the feedback given by the filter after inserting determined inputs [16]. The scale of the interaction available with the filter may vary.

The attack in which this work is based on does not assume anything about the filter. The pollution attempts will be performed against a black box filter with unknown size, number of hash functions and their details, although we assume they are cryptographically strong, which means there is a random distribution of the elements along the filter and its pollution will not rely on this to be successful.

B. Goal of the attack

In the first section of this work we have focused on example attacks performed to cause a Denial of Service Attack, which are typically achieved by filling the filter enough to make it ineffective, but these are not the only existing attacks to Bloom filters. Sometimes the attacker is more interested in finding an element whose search result collides with another element inserted in the filter. This has its applications for Bloom filters that work as network filters, allowing only certain elements to pass through [5].

This paper is focused on the first kind of attack. The goal is to fill the filter to the point of becoming useless, spending the least amount of time possible.

IV. Lookup attack

The present work is based on the lookup attack described in [15]. The lookup attack is a deterministic attack to pollute Bloom filters with the largest number of elements that increase from 0 to 1 different positions of the filter's vector. The algorithm works as follows (Algorithm 1). We start with an empty filter with size M and K hash functions. An F vector with a determined length is randomly generated. All the elements of this vector will be used each round of the attack to estimate the actual FPP of the filter for the round. The attack will have N rounds. For each of those rounds a vector T with a determined length will be randomly generated. Each of the elements of T in the round will be inserted and the actual FPP will be checked using F. After that, the element is removed, and we compare the result with the best result obtained in previous rounds. In the end, the goal is to determine which of all the elements of T in a particular round is the one the increases the FPP the most. In other words, that element will be the one that sets more counters to 1 in the filter.

Algorithm 1 Insertion on the lookup FPP attack

```
1: Randomly generate a set of test elements T
2: Randomly generate a set of elements F
3: Initialize Δ_max to zero
4: Measure FPP using F and assign it to FPP_before
5: for x in T do
6:         insert(x)
7:         Measure FPP using F and assign it to FPP_after
8:         Δ= FPP_after − FPP_before
9:         if Δ > Δ_max then
10:                z = x;
11:                Δ_max = Δ
12:        end if
13:        remove(x)
14: end for
15: insert z;
```

Logically, a larger size of T means more random elements tested against the filter, and the probability to find elements that increase the FPP will be bigger. This is especially critical in the latest stages of the attack because the probability of finding elements that set counters to 1 is very low. However, bigger sizes of T also mean it will be more demanding in terms of computation resources. In a similar way, the larger the size of the vector F, the higher the precision obtained in the measurement of the FPP, because a larger number of elements are tested against the filter. The problem with this is, again, it is more costly in resources.

For the attack to work, it is mandatory that the attacker has access to the filter to estimate the False Positive Probability of the targeted filter. This means the attacker can query a large set of random elements and check whether each of these elements exists in the filter or not. The FPP is then calculated as:

$$FPP = \frac{\text{number of matched elements}}{\|\vec{F}\|}$$

This can be achieved in some real-world scenarios, such as email blacklists described before.

The goal of this work is not only to discover more efficient ways to pollute BFs based on this attack, but also to determine

the best configuration in terms of time and FPP to perform the pollution.

Overall, the goal is to offer an improvement of the lookup attack. We will start by analyzing the best parametrization possible for vector T, since its size is a decisive factor in terms of time. Small values for T make the attack very fast, but also less reliable. On the contrary, a big value for T will produce accurate results, but also it will be very slow, which makes any real-world attack completely ineffective. The goal is first to find a balanced value for T. After that, new variants of the original algorithm will be tested. First, a variant in which the goal of the round will be to only improve the results of the previous one, which should reduce the pollution ability, but it should also decrease the execution time. The goal is to determine whether it is worth it or not. Finally, a new variant of the original blackbox attack will be studied in which we will try to predict the filter configuration based on the FPP and the round. If a correct prediction is performed, a more targeted attack can be performed in which we could approximate to the best FPP possible in each round and save a lot of time compared to a regular lookup execution.

V. The testing environment

To conduct all the tests described before, it was necessary to implement Counting Bloom Filters and the lookup attack. To determine the best programming language to implement both it was taken into account the fact that the implementation must be efficient and must allow the manipulation of unsigned variables as chunks of bytes to insert elements and query the filter. It was also necessary to be able to execute this software in any platform. After considering all the previous points, the C language has been chosen to implement it. It allows low-level manipulation of variables, including bit-wise operations. Also, it is a compiled language and gives total control of memory allocation, which makes it quite efficient. Lastly, it can be compiled and executed on almost any platform.

Both the lookup attack and the CBF have been programmed in a way that they allow the parametrization of the filter and the attack[1]. Filters can be created with any M size and K number of hash functions. The implementation of the filter is independent of the chosen hash functions. Any hash function can be used if the returned digest is passed as an unsigned integer, which will be later converted to a position of the filter. In a similar way, the lookup attack allows several configuration options, such as the size of the vector T and F and the number of n rounds executed.

All the tests have been performed against filters of different sizes and with different number of hashes. Specifically, it has been tested against filters with vector size M=2048, M=3072, M=4096 and M=5120 and a number of hash functions K=2, K=3 and K=4, in all of their combinations. The hash functions used are MD5 and SHA1 implemented in the openssl.h library, taking the most significant 64 bits for one position of the vector and the least significant 64 bits for another position. The automation of the tests has been developed in Bash scripting.

The results have been measured in FPP and computing time. The latter is calculated using the time.h library, which allows the calculation of the number of processor ticks consumed by the program and then dividing it by the number of processor clock ticks per second [17]. This means it may vary when executing this software in different platforms, although the results should not be affected by the full availability of the allocated resources at the moment of the execution, so the results should be identical in case of re-execution. All the tests have been executed in a Windows 10 laptop with a virtualized Ubuntu 2020 Desktop. The host machine has an eighth generation Intel i7 processor and 16GB of RAM. The guest virtual machine had assigned 8GB of RAM and 2 CPUs.

VI. Optimization of the test vector T

The first objective is to explore the impact of the size of the test vector T on both the FPP and the time needed for the attack. This is done in the next subsections, considering first a fixed T and then values of T different for each round.

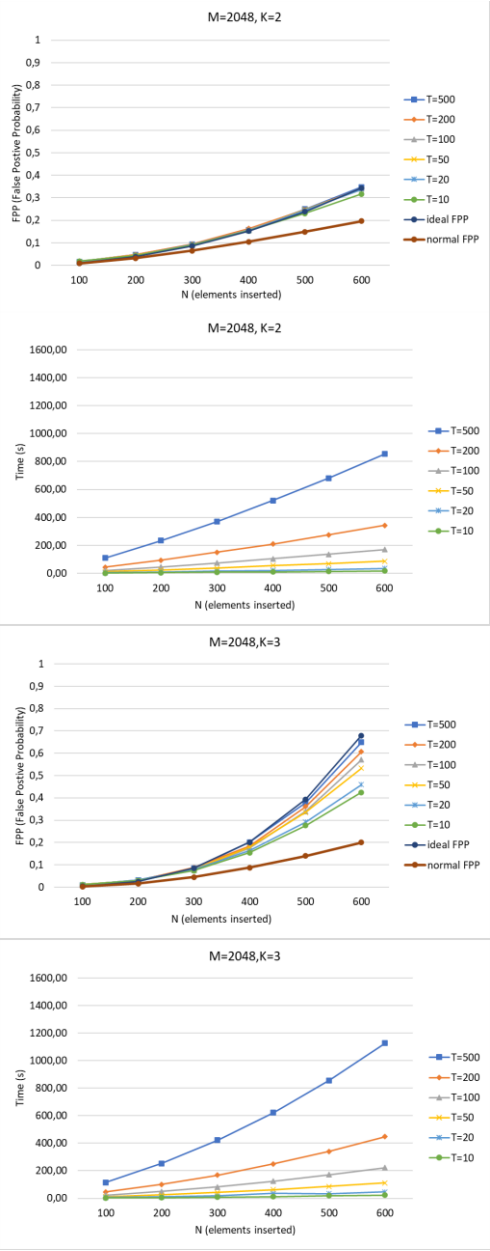
A. Constant T

As it was stated before, different parametrizations of the vectors F and T return different results when executing the lookup attack. The bigger the size of both vectors, the better FPP will be obtained when executing the attack. The drawback will be the higher consumption of computational resources, which requires longer times to complete the execution of the attack. A balance between those two factors must be found to perform effective attacks on the filter. This balance occurs when T has enough elements to find good candidates for the round (particularly in the latest rounds of the attack) but does not have too many elements, since the FPP calculation has to be performed for every single one of them. In order to do that, we tested several values for T that go from only 10 elements up to 500. For the test, F has a size of 10.000 elements. To understand the difference between those two values of T, we can do some numbers. In the first case (T=10), the filter is queried 10*10.000 = 100.000 times per round, while in the T=500 case the filter is queried 500*10.000=5.000.000 times per round. As it can be seen, there is a huge difference between both values of T, and it translates directly to the cost of performing every round of the attack.

In the Figure 3 we show the results. All the tests were executed against a filter with M=2048 and different number of hash functions. The ideal FPP and the normal FPP are also included for reference. The ideal FPP is the maximum possible value for the FPP assuming the attack has set from 0 to 1 the maximum number of counters in the filter every round (this is equal to the number of hash functions K). The normal FPP is the estimated FPP of the filter in a business as usual

[1] Both the source code and the results for the different tests can be found in https://github.com/PSayans/counting-bloom-filter

environment, assuming it will slowly increase when new elements are inserted.



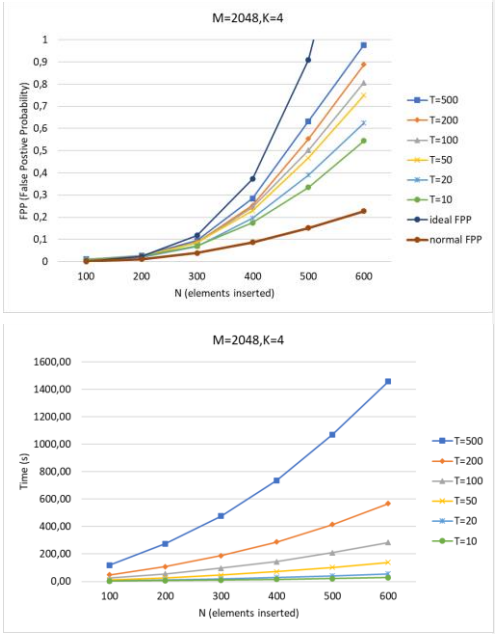


Fig. 3. Results in terms of FPP and execution time in seconds for different lengths of T in filters diferent sizes and number of hash functions

For K=2 there is close to no difference between using higher values for T or smaller ones. As K=2 only increments two counters per round, it is not as necessary to try many different values to find a good candidate in the round that improves the FPP. As it was expected, time keeps linear no matter the value of the FPP is, since as we know all the elements are tested in every round and it has always the same computational cost. For K=3, it starts to be obvious that a greater number of hash functions makes more difficult to find good candidates for the round. In this case makes it clear the more elements tested, the better the results obtained. We can also appreciate longer times in execution. This is logical since more hash functions require more computational time to perform those calculations when we query the filter, which happens a lot of times in every round of the attack. The K=4 scenario is the one where we find the greatest difference between using a small or big value for T. We can see that in the early stages of the attack (N=100, N=200 and N=300) the resulting FPP is quite similar between them, even close to the ideal FPP for the round, but from then on the difference starts to become bigger, as a larger T allows to test more elements it is easier to find a better candidate for the round. In the same way, the time needed to perform the attack becomes exponentially bigger. We can assume the same think would occur if we tested filters with more hash functions, increasing the distance between T=500 and T=10.

Comentado [PR1]: En las figuras 3 y 4 agrupas muchos plots y luego en las siguientes haces una figura para cada dos plots (FPP + Time).
Si puedes es mejor que las agrupes del mismo modo en todos los casos.

Independently of the FPP results, for the tested scenarios there is a huge computational difference between filters with different K. In the K=4 test, the attack lasted a 42% longer than for K=2 with the same value for T. This is interesting from the perspective of the security of the architecture. The more hash functions used in the filter, the harder for the attacker to perform an effective attack. This has its own drawbacks for the defender. More hash functions for the filter also means more resources needed to perform as expected in normal conditions.

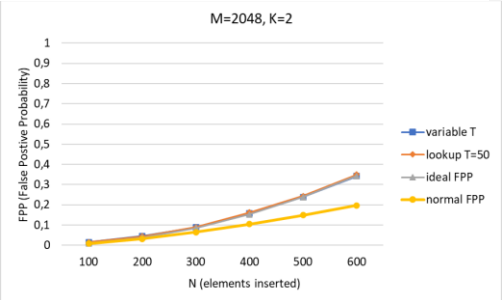
Depending on what we really want to achieve with the attack we may select a different value for T. If the goal is to fill the filter at any cost, we would focus our efforts in improving the obtained FPP at the end of the round, and thus we would use a higher value for T despite the high computational cost. However, if the main goal is to cause a Denial of Service attack in a real environment, time is critical for the attack to be successful since a slower attack may be detected by their cyberdefenses. Overall, and with the analyzed data, we can conclude that T=50 is a balanced parametrization for the attack, since the results for K=2 and K=3 are similar to the ones obtained for T=500, and in K=4 is close to the 75% of pollution of the filter, which is more than enough to make the filter ineffective. For this reason, in future tests of the algorithm and its variants we will use vectors of 50 random elements for T. If the filter had greater values for K, it might be interesting to use a higher value for T.

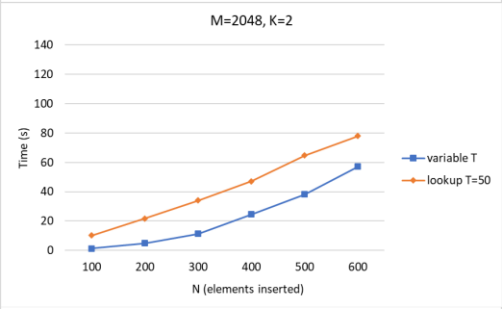
B. Variable T

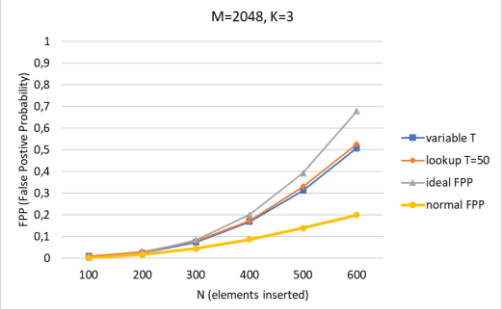
Using a fixed length for T is not efficient. The reason is that finding a good candidate for a particular round has not the same cost in the first stages of the attack as it could have in the latest. When the filter is empty, any inserted value would increment the FPP of the filter. In the subsequent rounds, almost any value of T would set all the K values of the filter to 1, if it has a value of M big enough. It is easy to set different counters to 1 since almost all of them are set to 0. However, when enough rounds have been executed, the filter has a lot of counters set to 1 or more, so it will be harder to find good candidates to insert in the filter, since most of them would increment an already set to 1 counter. For this reason, it may be convenient to adjust the length of the vector T depending on what round we are in, and thus saving computational time in the early stages of the attack. To do that, instead of a fixed value for T we can use a variable size based on the number of rounds N to determine which value of T will be used for a particular round. There are a lot of possible algorithms to determine the size of T based on the round. In this case, we have chosen to follow a linear progression. In the previous section we have stated that T=50 is an acceptable value for the attack even in the final rounds of the attack, so we can make T smaller in earlier rounds and increase the size of the vector as we insert elements in the filter. The value of T for a round N could be calculated as:

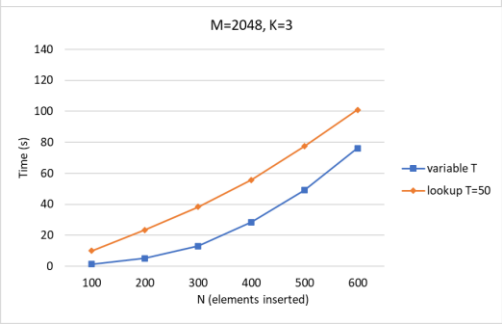
$$T_n = T_{n-1} + 0.1$$

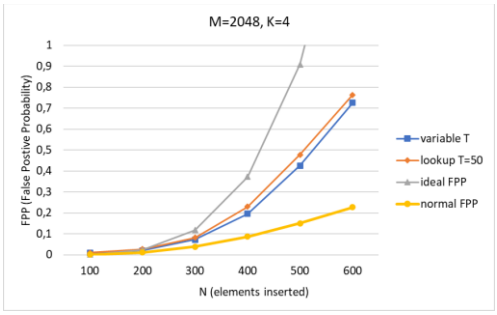
As in previous tests, the attack has been performed against several combinations of M and K, reporting the results for all of them. The following figures summarize the results for the variable T attack. These are the results for M=2048 and M=5120. The results for the rest of the configurations can be found in the Github page of the project.

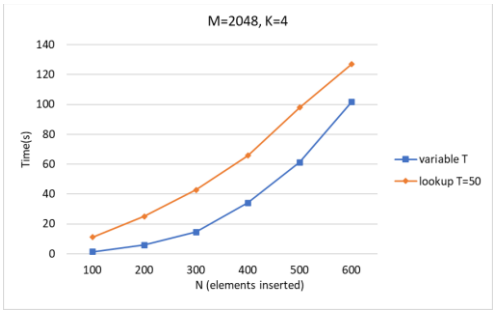


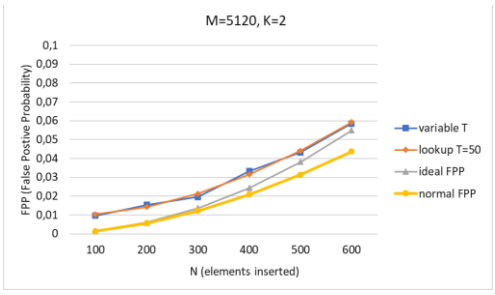


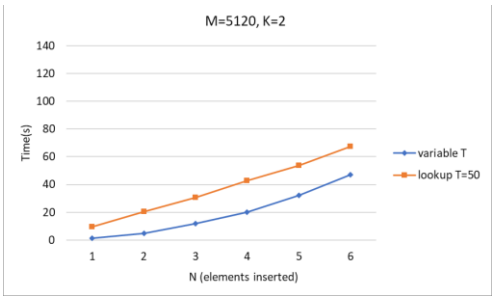


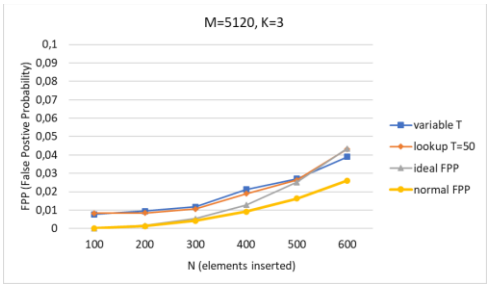


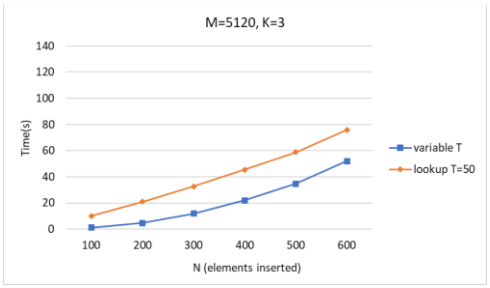


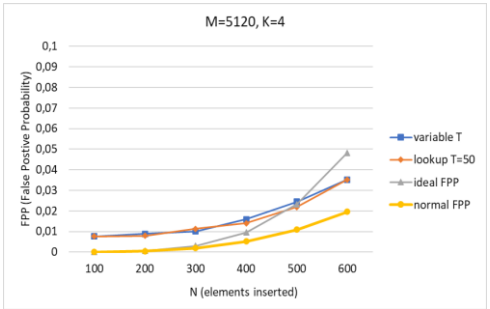












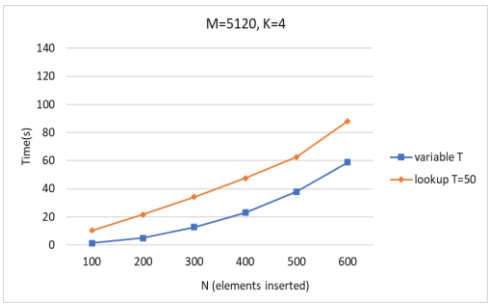


Fig. 4. Comparison of results in terms of FPP and execution time in seconds for different filter configurations for fixed T and variable T

As it can be observed, a variable T improves the results of the lookup attack for T=50 in a 32% in regard of the execution time. This improvement remains constant no matter what combination of M and K we are facing. This seems logic as it keeps proportionally smaller in all the stages of the attack, independently of the number of hash functions and the number of rounds performed. There is, however, a small degradation in the results obtained for the FPP. This is more observable in those filter configurations in which the FPP is harder to increase. Despite this, the saved computational time largely compensates the lost FPP, becoming an excellent configuration for the size of T along the whole execution of the attack.

VII. Improved delta attack

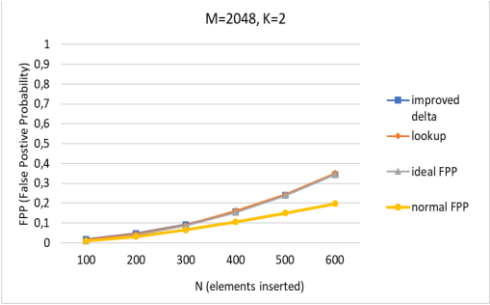
The focus of the lookup attack is to get the best False Positive Probability (FPP) possible in each round. To achieve that, a T vector with randomly generated elements is tested against the filter in each of the rounds. At the end, the element which returns the highest FPP is taken, and another round starts. This mechanism has its benefits. The attacker knows that for every round he is picking the best candidate to pollute the filter, since all the elements have been tested. The problem is that the best element, or at least an acceptable one, could be found but the round would keep executing until the end of the vector T. Although this technique is effective to maximize the FPP each round, it is also slow since it is forced to iterate over the whole T vector each round. In addition to this, every round of the attack is independent from the previous one, so the best element is always picked among the elements of T without considering how much the FPP was improved in the previous round.

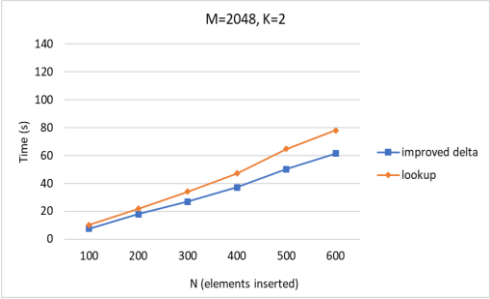
A variation of the lookup attack is proposed. A new parameter maximum value of delta is initialized to 0 before the first round starts. When in the round, if any existing element improves the previous maximum delta (in the case of first round any element), then the iteration stops, and that element is inserted. The new maximum delta is now the delta of the inserted element. This new maximum delta value will be used in the next iteration of the loop and so on. If none of the random elements contained by T improves the maximum delta for a certain round, then the lookup attack is normally applied to the round and the element with the best regular delta is inserted. In this way, the iteration over the vector T is performed while an element with a better delta than the previous round has not been found. In the best case, one of the first elements of the vector will improve the delta and the round would stop, going to the next one. In the worst-case scenario, the whole vector will be iterated over and none of the elements will improve the previous delta. In this scenario the regular lookup will be applied, and the best candidate will be the one inserted. The algorithm is more formally described in Algorithm 2.

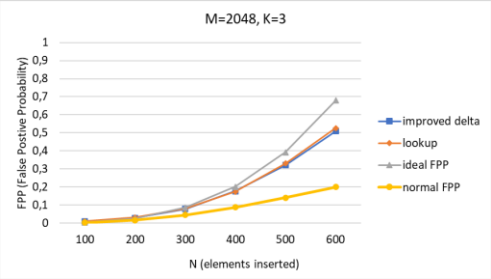
For this variant of the attack, the tests have been run against the same filter configurations as in Section VI. However, due to the results being quite similar, only the ones related to M=2048 have been included in this paper. The rest of the results can be found in the Github page of the project.

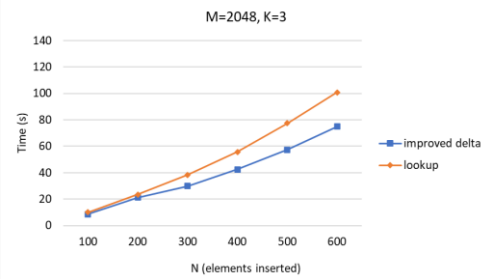
Algorithm 2 Improved delta attack

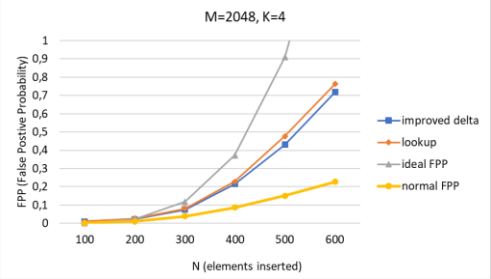
1: Randomly generate a set of test elements T
2: Randomly generate a set of elements F
3: Initialize Δ_{best} to zero
4: Measure FPP using F and assign it to FPP_{before}
5: **for** x in T **do**
6: $\Delta_{max} = 0$
7: insert(x)
8: Measure FPP using F and assign it to FPP_{after}
9: $\Delta = FPP_{after} - FPP_{before}$
10: **if** $\Delta > \Delta_{best}$ **then**
11: z = x;
12: $\Delta_{best} = \Delta$;
13: break;
14: **end if**
15: **if** $\Delta > \Delta_{max}$ **then**
16: z = x;
17: $\Delta_{max} = \Delta$
18: **end if**
19: remove(x)
20: **end for**
21: insert z;











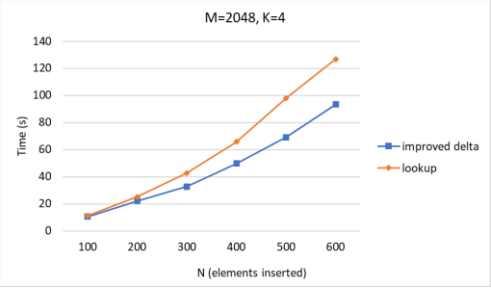


Fig. 6.Fig. 5. Results in terms of FPP and execution time in seconds for improved delta attack in different filter configurations

Applying an already optimized T=50 to both attacks, we get an average enhancement of a 21% for the improved delta attack. As it has also happened with similar tests in the past, more hash functions also mean a small loss in terms of FPP. Despite this, the saving in computational time is bigger than the lost FPP. At this point, it is important to remark that it is not necessary to reach an FPP close to 1 to make the filter ineffective. At the very moment the filter returns more positives than negatives the filter becomes incapable of work as expected. For this reason, small deviations in the FPP are not as important as making the attack faster. It is also important to mention that the attack is far from being completed in round N=600, so the difference in saved time between a regular lookup attack and improved delta will increase in the latest rounds of the pollution.

The time improvement might also be bigger when using a larger T in the attack, since the lookup attack is forced to iterate over the whole length of T while the improved delta attack only needs to find an element which increases the FPP for the round. It also should improve for a larger number of rounds performed. It is left for future works to determine the efficiency of this algorithm with higher values for T and N.

VIII. Prediction Attack

In this section, attacks that try to infer the filter configuration in terms of M and K are presented. The goal is to use that information to then speed up the attack.

A. *Prediction by fixed round*

The lookup attack is based on the premise that the filter is a black box of which nothing is known about. As such, we need to query F every round to test whether the inserted element for the round has improved the FPP of the filter or not. If we were able to determine which kind of filter we are dealing with, we would be able to perform a much more effective attack, since we could stop iterating over the vector T at the moment we find an element whose delta is close to the ideal delta for that round. Being this one as follows:

$$\Delta_{ideal} = FPP_{ideal_n} - FPP_{ideal_{n-1}}$$

The ideal FPP for a round n is calculated as:

$$FPP_{ideal} = \left(\frac{n \cdot k}{m}\right)^k$$

Ideally, this attack would stop at a determined round N and would perform an FPP fetch. Later, the obtained FPP would be searched among a list of pre calculated ideal FPPs for that round and for different filter configurations. The closest configuration for the obtained FPP would be chosen and the attack would continue taking that configuration as the reference to calculate the ideal delta for each of the following rounds.

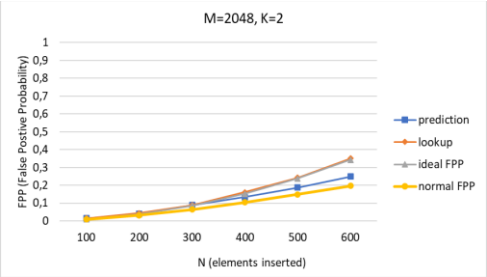
The reasoning behind this is the following. If you know the configuration of the filter you are attacking, you also know the maximum FPP you can obtain for a certain round. In a regular lookup attack you have two problems. The first one is the attacker does not know the configuration of the filter, and thus he cannot know if he has found an element that has the best delta possible in that round. The second problem is that

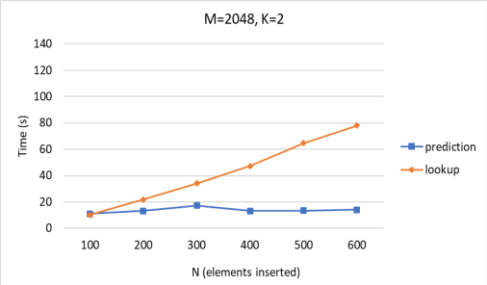
even if he knew the configuration of the filter, he would still iterate over the whole vector T before inserting the best element, even if he found it at the beginning of T. With this attack we expect to make use of the strongest ability of the lookup attack, which is its capacity to perform effective pollutions on blackbox filters and later measure the FPP of the filter. Theoretically, after N rounds the FPP of the filter should be close to its ideal FPP, especially if the attack has been performed with an acceptable value for T and thus a great number of candidates have been tested each round. At that point, we could check the obtained FPP with a list of plausible filter configurations. The closest ideal FPP in that list should be the attacked filter. From that moment, the attack would be performed not by checking the delta in the round, but by checking if the obtained delta for the inserted element is actually equal to the ideal delta in that round, and then stopping it. The algorithm is more formally described in Algorithm 3.

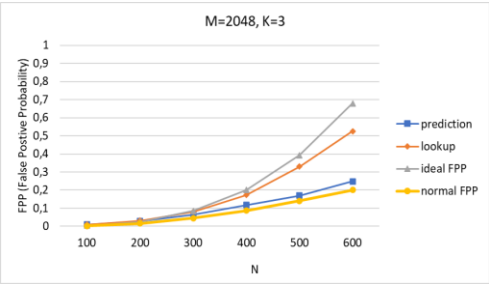
Algorithm 3 Prediction of the filter type by round

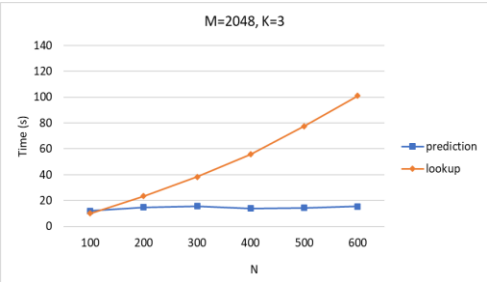
1: Initialize Max lookup
2: Initialize round counter to 0
3: **while** round counter < Max lookup **do**
4: Apply lookup attack (Algorithm 1)
5: **end while**
6: Measure the FPP
7: Look for the filter configuration closest to FPP
8: Randomly generate a set of test elements T
9: Randomly generate a set of elements F
10: Calculate ideal FPP for F in the previous round and assign it to $FPP_{ideal\ before}$
11: Calculate ideal FPP for F in current round and assign it to $FPP_{ideal\ after}$
12: $\Delta_{ideal} = FPP_{ideal\ after} - FPP_{ideal\ before}$
13: Measure FPP using F and assign it to FPP_{before}
14: **for** x in T **do**
15: insert(x)
16: Measure FPP using F and assign it to FPP_{after}
17: $\Delta = FPP_{after} - FPP_{before}$
18: **if** $\Delta >= \Delta_{ideal}$ **then**
19: z = x;
20: remove(x)
21: break;
22: **end if**
23: **if** $\Delta > \Delta_{max}$ **then**
24: z = x;
25: $\Delta_{max} = \Delta$
26: **end if**
27: remove(x)
28: **end for**
29: insert z;

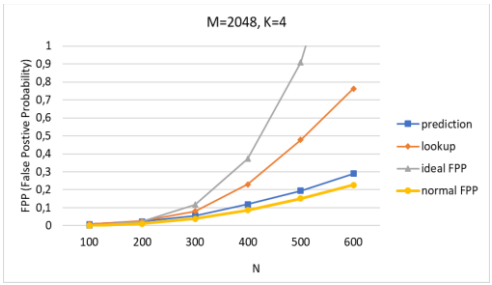
As with the rest of the attacks, the tests have been executed against different filter configurations to see its behavior. The number of rounds performed with the lookup attack are the half of the total rounds, being the other half the ones performed with the prediction. For attacks with N larger than 200 the lookup rounds are fixed to only 100 rounds, being the rest the prediction attack. Only a selection of the results is The results are the following:

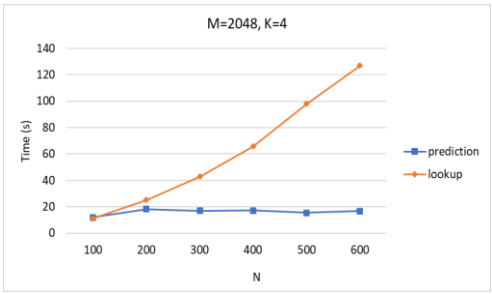


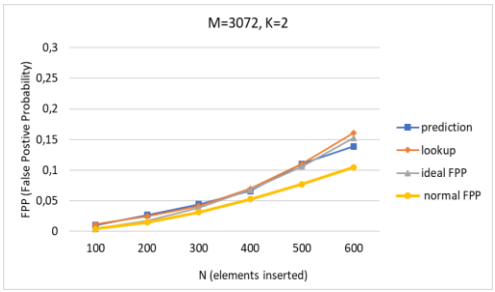


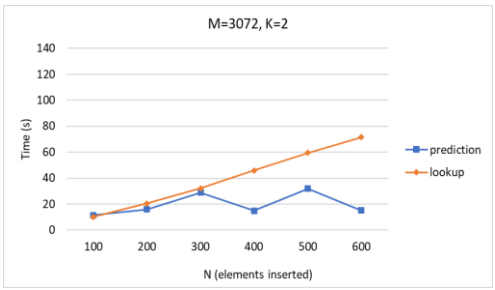


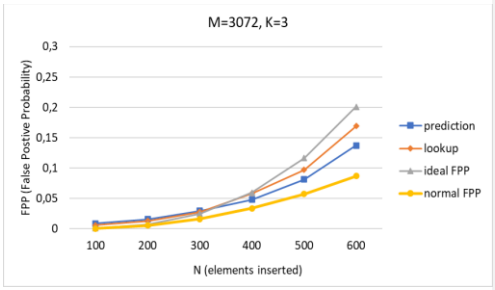


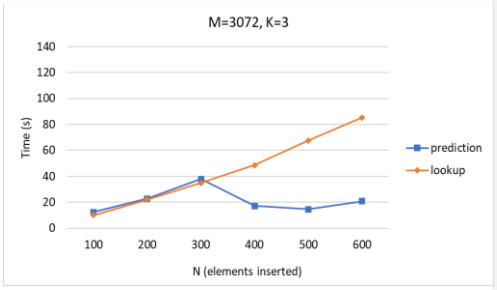


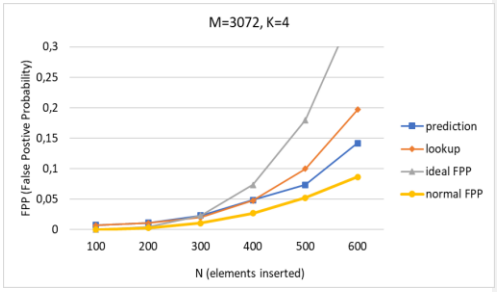












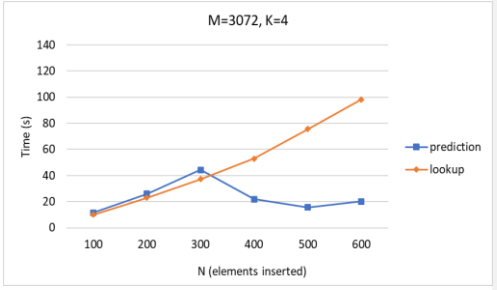


Fig. 7.Fig. 6. Results in terms of FPP and execution time in seconds for prediction attack in filters with different configurations

As it can be seen, when N is very low the attack fails completely to predict the configuration of the filter, and the

execution time is the same or a little longer than in the regular lookup attack. However, the results improve the original attack when enough number of rounds are executed. This is logical as the filter in the early stages of the attack is very low populated, and it is quite hard to estimate with precision the FPP, since just a few of the counters are set to 1, specially in filters with bigger values for m. This also implies this attack will not perform well when only a few rounds are executed. In addition to this, a small loss in the FPP for the round can be appreciated but it is compensated by an enormous improvement in computation time, which is a key factor when performing DoS attacks, where time is crucial for it to be successful. However, the attack has its problems. There is a probability of missing the correct configuration for the filter. This, again, happens normally in early stages of the attack or for certain filter configurations, in which the precision of the measurement of the FPP is very low.

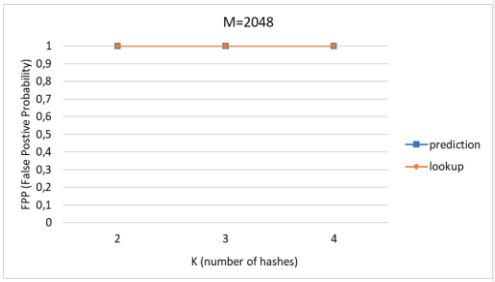
Under these circumstances, the attack cannot be used in a consistent way and thus it is only recommended to use when a better way to measure the FPP is found.

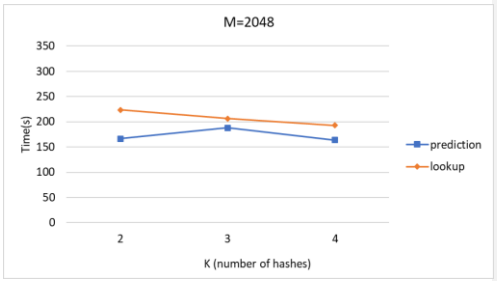
B. Prediction by fixed FPP

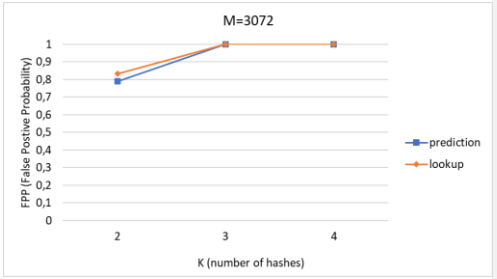
As we have seen before, the configuration of the filter we are attacking can be predicted by stopping the lookup attack in a determined round and then calculating the FPP for that round to compare it with a list of preset FPPs. The problem with the previous attack is the lack of precision when the FPP of the filter is low. In addition to this, each filter configuration returns different FPPs for a determined N round, so using a fixed number of rounds for all the attacks could return a very populated filter or a very low populated one. In the first scenario, we have performed more lookup rounds that the ones needed to predict the filter type and thus a lot of computational time was wasted. In the second scenario, the number of rounds performed are not enough to obtain a measurable FPP, and there is a high chance the prediction will fail, making the attack ineffective. In that case, a lot of elements must be inserted to get an accurate measurement of the FPP, so a lot of lookup rounds must be performed to get a useful feedback from the filter.

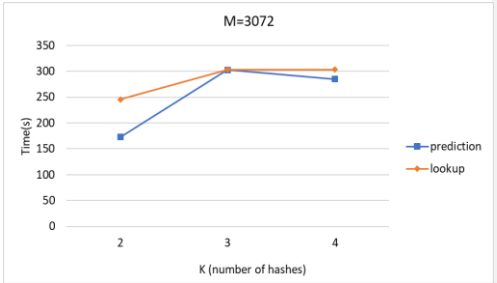
So, instead of executing the lookup attack for a fixed amount of rounds, a variation of it is proposed in which an undetermined amount of lookup rounds are executed until reaching certain FPP, and then we compared it with a list of precalculated FPPs for different combinations of M and k. The closest configuration is chosen, and the rest of the attack is executed looking for the best value for the ideal delta in the round, as it was done in the previous prediction attack. For the tests, a fixed FPP of 0.1 has been configured, which we consider a value high enough to predict effectively different filter configurations.

In this scenario, all the tests have been performed using a fixed number of rounds N=1500. The reason why in this case we use a very high N is because it is unknown how many rounds will cost to reach an FPP of 0.1, but we need a fixed number of rounds to be executed. Otherwise, the algorithm could last an excessive amount of time. We also know from previous results of the lookup attack [15] that in certain configurations of the filter a lot of rounds are required to reach an FPP equal or greater to 0.1, so 1500 rounds seems an acceptable value for N in this attack. The results are the following:









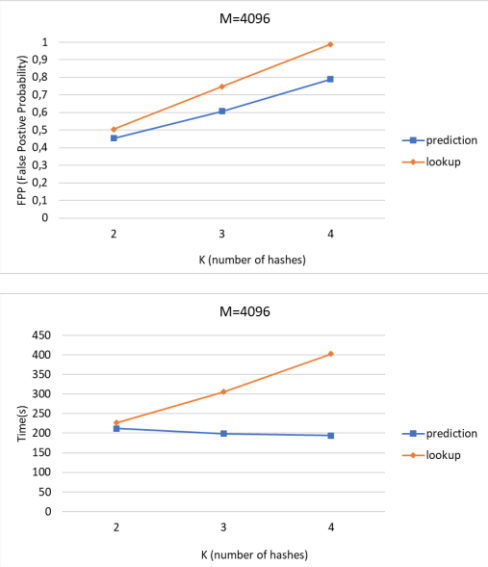


Fig. 8.Fig. 7. Results in terms of FPP and execution time in seconds for prediction attack with fixed FPP in filters with different configurations

For this attack we have obtained mixed results. The problem in this case is that there are lot of filter configurations very similar for the round in which the attack gets to the fixed configured FPP. The lack of precision in the measurement of the FPP causes a bad reading of the state of the filter and thus confusing the attacked filter with another one with similar FPP for the round in which the lookup attack stopped. Observing the results, it is concluded that the precision starts to decay when the FPP for a particular round and configuration of the filter is similar to the FPP in another filter in the same round but with a different configuration. To be successful, a better measurement of the FPP=0.1 must be performed. This might be achieved by increasing the number of tested elements in F, but there will be an increment in the execution time, and thus a worse performance of the attack.

In conclusion, the algorithm is unpredictable in the tested conditions. For that reason, this attack is not considered recommendable to use unless a better way to measure effectively the FPP of the filter is found.

IX. Analysis of the Results

As a summary, we can conclude that there is not a perfect way to perform a pollution attack against Bloom filters with the studied attack. We can achieve a better pollution rate at the cost of losing performance and the other way around, but we cannot have both. The key is to find a balance between an acceptable False Positive Probability each round and the good execution time needed for a real-world scenario. For the filter configurations analyzed we have found good values to reach the mentioned balance.

In addition to this, new variants of the lookup attack algorithm have been developed. One of them, the improved delta attack, assumes that it is not necessary to find the best element possible from T, but only an acceptable one. This attack, again, sacrifices FPP to obtain a better performance, but is worth enough since the lost FPP is smaller than the saved execution time. The other developed attack takes advantage of the efficiency of the regular lookup attack when it comes to pollute a BF and tries to identify the configuration of the filter after several rounds. If a successful identification has been performed, the attack starts to pollute the filter much faster than a regular lookup attack, since we know the maximum FPP we can get every round, and it is not necessary a further search in the round. The problem with this attack is the lack of precision it may have when measuring the FPP. In the tests performed not always is correctly identified, and when this happens the attack is not efficient.

There is also another problem. The identification of the filter must be performed in the early stages of the attack, when the capacity of the lookup attack to fill the filter is close to the ideal FPP. The more rounds executed, the higher the distance between the ideal FPP and the actual FPP of the filter, which makes the prediction near impossible. For that reason alone, only by improving the measurement of the FPP we can perform an effective prediction attack.

X. Conclusions and Future Works

Overall, we have achieved a significant improvement in lookup attack by varying the parametrization of several inputs. In some cases, there has been a small loss of FPP but in return we have gained a notable improvement of the performance for most of the scenarios which compensate the mentioned loss.

However, not all the new algorithms have proven to be effective in all the cases. This is mainly caused by the lack of precision in the measurement of the FPP, which causes poor results in the execution. This occurs in filter configurations in which ideal FPP for a particular round has a similar value to another filter with a different configuration. This filters typically have a very small value for the FPP at early stages of the attack, and thus it is very hard to measure it with precision. This lack of precision happens no matter what value of F is used up to 30.000 elements, so the problem may be either the way to test the FPP itself or the number of tested elements in F. A better precision for the FPP measurement is left to future works.

In addition to this, in the case of improving the precision, it is suggested to reduce the FPP required for the round to be lower than the ideal FPP using a threshold. It could reduce even further the execution time since it is not necessary to find the exact element that increments the maximum number of counters. With a threshold we could stablish a good enough element for every round and advance to the next one earlier.

Another proposed work is instead of taking only one configuration as the predicted filter, to take a couple of them very close in FPP and start two attacks at the same time, picking after several rounds the one that is filling the filter faster.

REFERENCES

[1] B. Bloom. “Space/Time Tradeoffs in Hash Coding with Allowable Errors.”Communications of the ACM13:7, 422—42, 1970

[2] A. Broder and M. Mitzenmacher, “Network Applications of Bloom Filters: A Survey”, 2011

[3] K. Kim, Y. Jeong and Y.L.S. Lee, “Analysis of Counting Bloom Filters Used for Count Thresholding”, Department of Electrical Engineering, Pohang University of Science and Technology (POSTEC), 2019

[4] Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder, “Summary cache: A scalable wide-area web cache sharing protocol”, IEEE/ACM Trans. Netw. 8, 281–293, 2000.

[5] M. Naor and Y. Eylon, “Bloom Filters in Adversarial Environments”, ACM Trans. Algorithms 15, 3, Article 35, 2019

[6] L. Luo, D. Guo, R. Ma, O. Rottenstreich and X. Luo, “Optimizing Bloom. Filter: Challenges, Solutions, and Comparisons,” IEEE Communications, Surveys and Tutorials, vol. 21, no. 2, pp. 1912–1949, 2019.

[7] L. Maccari, R. Fantacci, P. Neira and R.M. Gasca, “Mesh Network Firewalling with Bloom Filters”, IEEE International Conference on Communications (ICC). 1546 - 1551. 10.1109/ICC.2007.259, 2007

[8] D. Guo, Y. He and Y. Liu, “On the feasibility of gradient-based data-centricrouting using Bloom filters,”IEEE TPDS,vol. 25, no. 1, pp. 180-190, 2014.

[9] D. Bauer, P. Hurley, R. Pletka, and M. Waldvogel, “Bringing Efficient Advanced Queries to Distributed Hash Tables,” Proc. IEEE Conf. Local Computer Networks, pp. 6-14, 2004

[10] T. Gerbet, A. Kumar and C. Lauradoux, “The Power of Evil Choices in Bloom Filters”, 2014.

[11] Ka. Li and Z. Zhong, “Fast statistical spam filter by approximate classifications”, In Proceedings of the International Conference on Measurement and Modeling of Computer Systems and the International Federation for Information Processing (SIGMETRICS/Performance’06). ACM, 347–358, 2006

[12] J. Yan and P. Leong Cho, “Enhancing collaborative spam detection with Bloom filters”, In Proceedings of the Annual Computer Security Applications Conference (ACSAC’06), IEEE Computer Society, 414–428, 2006

[13] R. Patgiri, S. Nayak and S. Borgohain, “Role of Bloom Filter in Big Data Research: A Survey”, International Journal of Advanced Computer Science and Applications. 9. 655-661. 10.14569/IJACSA.2018.091193, 2018

[14] *Bloom Filter Pattern*, Redis Labs, Accessed on: September 15, 2020. [Online]. Available: https://redislabs.com/redis-best-practices/bloom-filter-pattern/

[15] P. Reviriego and O. Rottenstreich, “Pollution Attacks on Counting Bloom Filters for Black Box Adversaries” 16th International Conference on Network and Service Management (CNSM), 2020

[16] R. J. Tobin and D. Malone, “Hash pile ups: Using collisions to identify unknown hash functions,” in Proc. International Conference on Risks and Security of Internet and Systems (CRiSIS), 2012.

[17] *ISO 9899*, ISO/IEC, Accessed on: September 15, 2020. [Online]. Available: http://www.open-td.org/jtc1/sc22/wg14/www/docs/n1124.pdf

[18] M. Antikainen, T. Aura, and M. Särelä, “Denial-of-Service Attacks in Bloom-Filter-Based Forwarding”, Networking, IEEE/ACM Trans. Netw. 22, 1463-1476, 2014