Pollution Attacks on Counting Bloom Filters for Black Box Adversaries

Pedro Reviriego
Universidad Carlos III de Madrid
Madrid, Spain
revirieg@it.uc3m.es

Ori Rottenstreich
Technion
Haifa, Israel
or@technion.ac.il

***Abstract*—The wide adoption of Bloom filters makes their security an important issue to be addressed. For example, an attacker can increase their error rate through polluting and eventually saturating the filter by inserting elements that set to one a large number of positions in the filter. This is known as a pollution attack and requires that the attacker knows the hash functions used to construct the filter. Such information is not available in many practical settings and in addition a simple protection can be achieved through using a random salt in the hash functions. The same pollution attacks can also be done to counting Bloom filters that in addition to insertions and lookups support removals. This paper considers pollution attacks on counting Bloom filters. We describe two novel pollution attacks that do not require any knowledge of the counting Bloom filter implementation details and evaluate them. These methods show that a counting Bloom filter is vulnerable to pollution attacks even when the attacker has only access to the filter as a black box to perform insertions, removals, and lookups.**

***Index Terms*—Bloom filter, Security, Pollution attacks.**

I. INTRODUCTION

Since their invention fifty years ago Bloom filters [1] have found numerous applications in computing and networking [2], [3]. They are widely used for set representation while speeding membership checking in applications where a small amount of false positives is tolerable. Bloom filters are for example used to reduce the number of accesses to external memory in both networking devices and computing systems, to speed up routing, packet classification or security checking [4]. Filters are also used to accelerate access to forensic information [5] or packet inspection [6]. The original Bloom filter data structure only supports insertions and lookups but it has since been extended and optimized to support removals [7], multi-set membership checking, or to reduce the false positive probability [8]. In fact, after five decades, the optimization of the Bloom filter is still an active research topic [9]–[11].

The security of Bloom filters is a key issue given their wide use in various systems and applications. As most data structures, Bloom filters can be the target of algorithmic complexity attacks [12] that try to reduce their effectiveness. This type of attacks has been described, for example, for hash tables [13], for cardinality sketches [14], for frequency estimation sketches [15], for cuckoo filters [16] and for more complex packet classification algorithms [17].

In the case of Bloom filters, an attacker can try to create false positives for arbitrary elements by inserting other elements thus effectively evading the filter when used for detection [18]. This attack, known as target-set coverage attack [15], can be easily done if the filter implementation details, i.e. the hash functions it makes use of are known to the attacker. However, even when that is not the case and the attacker can only access the filter as a black box (i.e. having no knowledge of how the filter is implemented), false positives can still be created for selected elements [15], [18]. Another type of attacks, known as pollution or saturation attacks, aims at reducing the performance of the filter by increasing its false positive probability or even saturating it to one [19]. This can be done by inserting elements that do not collide, that is map to the different positions on the filter. Again, finding not colliding elements is straightforward when the hash functions used in the filter are known [19]. However, when the attacker has only access to the filter as a black box, avoiding those collisions is not straightforward.

While the basic Bloom filter cannot support removals of elements from the set, it has a common variant named Counting Bloom filter that allows removals through keeping an array of counters rather than bits [7]. In this paper, attacks on a Counting Bloom filter for which the attacker has no knowledge of the implementation details are presented. The rest of the paper is organized as follows. A brief description of Bloom filters and existing attacks is given in Section II. The proposed pollution attacks are presented in Section III and evaluated in Section IV. Finally, we conclude in Section V.

II. PRELIMINARIES

This section provides a brief description of Bloom filters and known approaches to attack them.

A. *Bloom filters*

Bloom filters implement set representation allowing fast and simple membership testing. Consider a represented set S and a membership lookup of an element x. Upon a lookup, when the filter returns a negative indication, then element x is not in the filter. Instead, when the filter returns a positive indication, there is a probability, known as False Positive Probability (FPP) that the element is not in set S. This probability depends on the size of the set S, the memory allocated to the filter, and its configuration. The structure of a Bloom filter is shown in Figure 1. It is formed by an array of m bits and elements x are mapped to the filter using a set of k hash functions $h_1(), h_2(), ..., h_k()$ [8]. To insert an element x the bit positions that correspond to $h_1(x), h_2(x), ..., h_k(x)$ are set to one. To perform a lookup for an element x, the same positions $h_1(x), h_2(x), ..., h_k(x)$ are read and only if all of them are one, then a positive indication is returned. Otherwise, the lookup operation returns a negative.

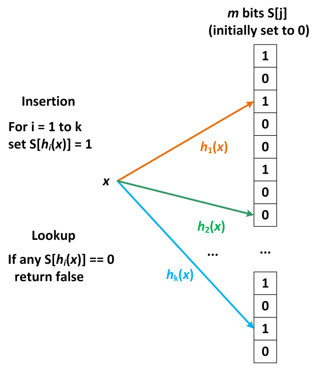

Fig. 1. Illustration of a Bloom filter with a single array of m bits to which element x is mapped

In approximate membership lookups, there are two kinds of errors: false negatives and false positives. An inherent property is that by construction, a Bloom filter cannot have false negatives as if element x has been inserted to the filter, the positions it maps to would be set to one upon its insertion and thus a lookup would result in a positive answer. However, false positives can occur as even if x has not been inserted its corresponding positions may have been set to one when inserting other elements mapping to them. Typically, Bloom filters are designed to achieve a small false positive probability. This can be done by selecting a size m for the filter array that is significantly larger than the size of the set S to be stored in the filter. The False Positive Probability (FPP) of a Bloom filter of size m in which $n = |S|$ elements have been inserted is approximated [20] by:

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

For large enough m the probability can be approximated by:

$$FPP \approx \left(1 - e^{-\frac{k \cdot n}{m}}\right)^{k}. \tag{2}$$

Therefore, as the filter length m increases, the FPP decreases. Conversely, as the set size n increases the FPP increases.

In some cases, an alternative implementation is used for the Bloom filter. Instead of having a single array to which all the hash functions map, there is a different array for each hash function. This is shown in Figure 2. Each array can be mapped to a different memory and when the k memories are accessed in parallel, operations can be completed in a single memory access time. This is convenient for hardware implementations, for example when the Bloom filter is implemented on a Field Programmable Gate Array (FPGA) or on an Application Specific Integrated Circuit (ASIC). Such an implementation with k multiple arrays of length m/k of the Bloom filter achieves a similar FPP as the traditional implementation with a single array of length m and thus the choice between the two is typically dictated by the platform used for system [3].

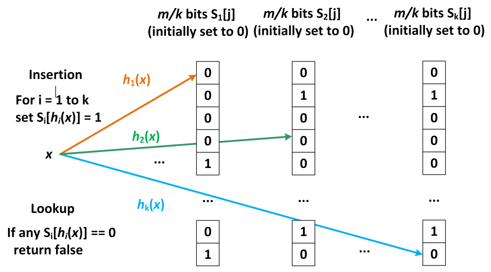

Fig. 2. Illustration of a Bloom filter with k arrays to which element x is mapped

The Counting Bloom Filter (CBF) [7] generalizes the functionality of the Bloom filter by supporting element removals. Insertions increment counters while removals decrement the counters. Lookups are positive when all the counters checked are larger than zero. The CBF implies a similar FPP as the Bloom filter when its number of counters equals the number of bits of the Bloom filter.

B. *Attacks on Bloom filters*

The security of Bloom filters has been studied in several works over the last decade, for example in [15], [18], [19] and several attacks on the filters have been described. One type of attacks is **denial of service attacks** that target systems that rely on the Bloom filter to reduce processing complexity. This reduction in processing is achieved by discarding elements that return a negative from further processing. Therefore, an attacker can perform a denial of service attack by identifying a large number of elements that are false positives and creating a burst of false positive lookups. This can be done by performing only lookups to the filter [19].

A second type of attacks known as **target-set coverage attack** consists of forging a false positive for a given element x or set of elements by inserting other elements on the filter [15]. This can be used by an attacker for example to bypass detection when the filter is used to implement a white-list [19]. This target-set coverage attack in general requires inserting new elements on the filter.

Finally, a third type of attacks, known as **pollution or saturation attack**, tries to increase the FPP of the filter by selecting the elements to be inserted [18]. The goal is to minimize the number of positions that store a zero as that would maximize the FPP. These pollution or saturation attacks require to find elements that do not collide with each other on the filter [18]. This can be easily done when the hash functions used in the filter are known to the attacker or can be inferred from collisions [21].

III. Pollution attacks on Counting Bloom filters

In this section, the model of the attacker is discussed first. Then two different attacks are described and analyzed theoretically: a lookup time attack and a delta on false positive probability attack. The first method exploits the time to complete a lookup to identify elements that pollute the filter while the second relies on the increment on the FPP implied by an insertion of an element to the filter.

A. Adversarial model

An attacker can have different levels of access to the Counting Bloom filter. For example, when the attacker knows the implementation details of the filter, i.e. the hash functions used, he can easily forge false positives for arbitrary elements. This knowledge can be easily obtained for open source based implementations [18] while for other systems, hash functions may be inferred from collisions depending on the hash function used [21]. However, a salt that modifies the input to the hash functions by adding a different random number for each filter instance can be used to reduce the attacker's ability to infer the hash functions [14]. In this paper, we consider an attacker that sees the filter as a black box. Namely, he has no knowledge on how the filter is implemented and can only perform user operations on the filter: insertions, removals and lookups.

For the black box model, further limitations can be placed on the type and/or number of operations that the attacker can perform [19]. In our case, we do not place limitations on the type of operations that an attacker can make. Therefore, the attacker can insert, remove and lookup for elements. As for the number of operations, the only restriction placed is that the number of insertions and removals has to be much smaller than the number of lookups. The reasoning behind this is that under normal operation that would be the case and thus an excessive number of insertions or removals compared to lookups can be detected as an anomaly. In summary, an attacker can perform operations on the filter choosing arbitrarily the elements to insert, remove or lookup but does not know the hash functions used internally by the filter.

B. Lookup time attack

The first attack exploits the differences in lookup time in a sequential implementation of the filter lookup. The use of the lookup time has been previously proposed to identify collisions in hash tables [22]. A pseudocode of such implementation is given in Algorithm 1. The lookup time depends on the number of accesses to the array. A positive lookup would access $a = k$, positions. Instead, for a negative access, the number of positions accessed a depends on when the first zero is found and can take values $a \in [1, k]$. In particular the lookup time makes it possible to identify non-member elements that do not encounter collisions for their first hash functions.

To increase the number of ones on the filter upon the insertion of a single element, an attacker could first do a lookup for a set of elements Z and insert the one that has the lowest lookup time. When the set Z is sufficiently large, such element z would in most cases encounters a negative indication and observes a zero bit in position $h_i(z)$ for some small i. This ensures that at least one position is set to one by the insertion and that it corresponds to the first accessed bits. The insertion for the attack algorithm is summarized in Algorithm 2. First a set of random elements Z is generated for testing. Then, a lookup is done for each of them and the one with the lowest time is selected and inserted into the filter.

Algorithm 1 Sequential lookup on a Bloom filter

```
1: input element $x$
2: for $i = 1$ to $k$ do
3:     if $S[h_i(x)] == 0$ then
4:         return negative;
5:     end if
6: end for
7: return positive;
```

Algorithm 2 Insertion on the lookup time attack

```
1: Randomly generate a set of test elements $Z$
2: Initialize $time_{min}$ to a large value
3: for $x$ in $Z$ do
4:     lookup($x$)
5:     if $time_{lookup}(x) < time_{min}$ then
6:         $z = x$;
7:         $time_{min} = time_{lookup}(x)$
8:     end if
9: end for
10: insert $z$;
```

For this attack to be feasible, the lookup time has to depend on the number of accesses to the array such that a lookup with more accesses takes more time than another lookup with fewer accesses. That may not necessarily be the case when for example the filter is implemented on a system that uses a memory hierarchy and the positions that correspond to some elements are in the cache while for others they are not. The same applies to parallel filter implementations on which each array is mapped to a different memory and all memories are accessed at the same time so that lookup operations are always completed in one memory access cycle.

Let us now consider the effect of this attack on the number of ones in the filter and on its false positive probability. For the single array implementation, under normal operation, the insertion of an element x on a filter on which other e elements have been previously inserted would on average increment the number of ones in the array by:

$$I_{normal}(e) \approx \sum_{i=1}^{k} \binom{k}{i} \cdot p_0(e)^i \cdot p_1(e)^{k-i} \cdot i \tag{3}$$

where $p_j(e)$ $(j \in \{0, 1\})$ is the probability for a bit in the array to have a value of j that is a function of the number of elements e earlier inserted to the filter. In more detail, p_1 after e elements have been inserted can be approximated by:

$$p_1(e) \approx 1 - e^{-\frac{k \cdot e}{m}} \tag{4}$$

When the attack algorithm is applied to the insertion, the increment would be on average:

$$I_{attack}(e) \approx 1 + \sum_{i=1}^{k-1} \binom{k-1}{i} \cdot p_0(e)^i \cdot p_1(e)^{k-1-i} \cdot i \quad (5)$$

If the attack is used for all the insertions to the filter, then the FPP would be approximately:

$$FPP_{time-attack-single} \approx \left(\frac{\sum_{e=1}^{n-1} I_{attack}(e)}{m} \right)^k \quad (6)$$

An interesting observation is that this attack behaves differently for the k array filter implementation (note that the time based attack is applicable only when arrays are accessed sequentially). In that case the attack can only affect the first array (as elements that map to a zero on that first array would have the smallest lookup time) while the other $k-1$ arrays would operate normally. Therefore, the attacker would only change the number of ones on the first array. In fact, after n element insertions on the filter, the number of ones on the first table would be exactly $\frac{n}{m}$ and thus the FPP can be approximated by:

$$FPP_{time-attack-multiple} \approx min\Big(\frac{n}{m}, 1\Big) \cdot \big(1 - e^{-\frac{n}{m}}\big)^{k-1}. \quad (7)$$

In this case, the attacker would in the worst case be able to increase the FPP of the first array to one leaving the other arrays unaffected. Therefore, in this worst case, the FPP of the filter would be that of a filter with only $k-1$ arrays. This would occur when the number of inserted elements n is equal to m. If further insertions are made, the rest of the arrays could be polluted. This is not considered in the paper and is left for future work.

C. Delta on False Positive Probability attack

The lookup time based attack presented in the previous section has a number of limitations. Firstly, it does not apply to some filter implementations, for example those using several arrays mapped to different memories and being checked in parallel. Another limitation is that the collisions are avoided only on the first hash function. This limits the effectiveness of the attack in increasing the FPP compared to existing attacks that avoid collisions on all hash functions [19]. However, differently from our black box model, those attacks assume that the attacker knows the hash functions used to implement the filter. In that case, the attacker can just randomly generate elements until one that maps to positions that are all zero is found. However, finding such an element when the attacker has only a black box access to the filter is not straightforward.

An interesting observation is that the insertion of an element z in the filter increases its FPP by an amount that depends on the number of bits that are set to one by the insertion. Therefore, by observing the FPP after an insertion it may be possible to infer the number of bits that have been set to one. In more detail, let us consider a filter that has o bits with the value of one. Then its FPP would be approximately $(\frac{o}{m})^k$. If a new element z is inserted, it would increment the number of ones by $j \in [0, k]$ and the resulting FPP would be approximately $(\frac{o+j}{m})^k$. Therefore, the value of j could potentially be inferred by measuring the FPP before and after the insertion.

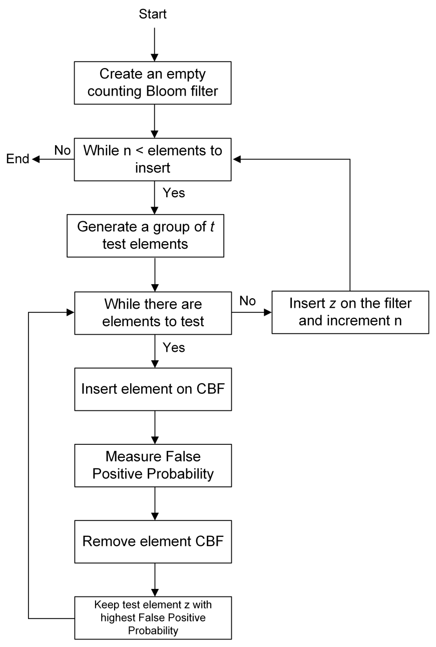
Start
Create an empty counting Bloom filter
While n < elements to insert
No
End
Yes
Generate a group of t test elements
While there are elements to test
No
Insert z on the filter and increment n
Yes
Insert element on CBF
Measure False Positive Probability
Remove element CBF
Keep test element z with highest False Positive Probability

Fig. 3. Flow diagram of the simplified delta on FPP attack

Except for very low filter occupancy, it can be assumed that the number of ones is much larger than the number of hash functions so that $o \gg k$. Then, $(\frac{o+j}{m})^k - (\frac{o}{m})^k$ can be approximated by $\frac{k \cdot j \cdot o^{k-1}}{m^k}$ and thus the increment in the FPP can be approximated by:

$$\Delta\ FPP \approx \frac{k \cdot j \cdot o^{k-1}}{m^k} = \Big(\frac{k \cdot j}{o}\Big) \cdot FPP. \quad (8)$$

This increment would be much smaller than the FPP and thus may not be easily measured. In fact, to completely avoid collisions, elements that set $k-1$ bits to one have to be differentiated from elements that set k bits and in that case the difference is even smaller.

An alternative approach that requires less accurate measurements is to test several elements and pick the one that has the largest FPP. In some cases, this procedure would pick elements that have not set all k positions to one, but this scheme should achieve good pollution with a deterministic execution time and number of operations on the filter. Figure 3 illustrates this procedure. For each insertion, a group of t elements are tested one by one by inserting the element, measuring the FPP and removing the element. Then the element with the largest FPP is selected and inserted on the filter. This process is repeated for the n element insertions. At the end of the process, the filter should have a number of positions that are not zero much larger than when random elements are inserted. The main parameters of the algorithm are the size of the test set t and the number of elements checked to estimate the false positive probability f.

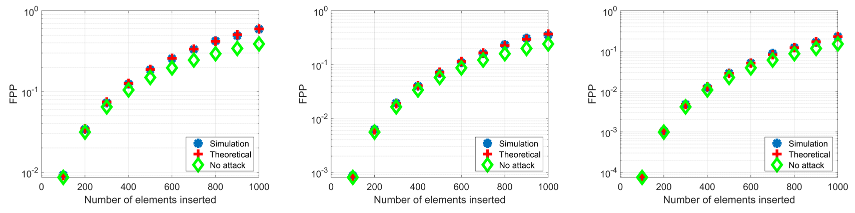

Fig. 4. False positive probability for the lookup time attack on a single array filter, k = 2 (left), k = 3 (middle) and k = 4 (right)

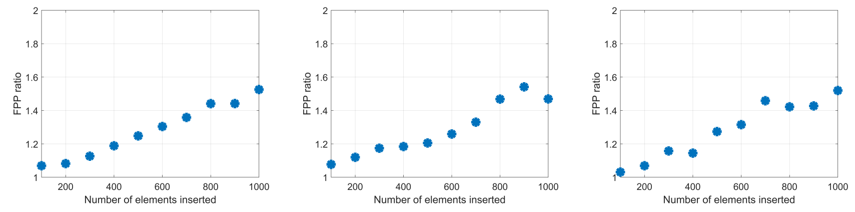

Fig. 5. Ratio of the False positive probability with lookup time attack versus no attack on a single array filter, k = 2 (left), k = 3 (middle) and k = 4 (right)

To speed up the attack both t and f can be modified as a function of the measured FPP. If the FPP is low, then p_0 would tend to be high and the number of elements that need to be tested to find an element that sets all k positions to one should be small. Conversely, if the FPP is high, p_0 would be small and finding such an element would be harder so t needs to be larger. On the other hand, the size of the set used to measure the FPP should be inversely proportional to the FPP. That is, larger values of f are needed when the FPP is low.

In more detail, the probability that a given insertion sets all k positions to one is given by $(p_0)^k$. Therefore to have with high probability at least one test element that sets all k bits the test set t should be larger than $\frac{1}{(p_0)^k}$. Correspondingly, to be able to detect $\Delta\ FPP$, the FPP should be measured with an accuracy better than that. The standard deviation of the measurement is given by $\sqrt{\frac{FPP\cdot(1-FPP)}{f}}$ [23]. Therefore, by setting $f = \frac{g}{FPP}$ where g is the parameter we want to determine and assuming that the FPP is much smaller than one, the standard deviation can be approximated by $\frac{FPP}{\sqrt{g}}$. Now g can be selected so that the standard deviation is smaller than $\Delta\ FPP$ by making $g \gg (\frac{o}{k})^2$. This allows computing the value of $f = \frac{g}{FPP}$ to use.

The proposed algorithm is just a proof of concept to show the feasibility and effectiveness of the attack. Potentially, more efficient algorithms can be developed to speed up the attack. The study of optimized attacks algorithms is left for future work. The number of positions set to one in the filter after n insertions would depend on the effectiveness of the algorithm that in turns depends on the size of the testing set T and the FPP measuring set F. In the best case, the number of ones would be $n \cdot k$ and thus the FPP would approximately be $(\frac{n\cdot k}{m})^k$. In the following section, it is shown that values close to this upper bound can be achieved in practical configurations.

IV. Evaluation

To show the feasibility and assess the effectiveness of the proposed attacks, they have been implemented and tested using counting Bloom filters with $k = 2, 3, 4$ hash functions and both a single array and multiple arrays configurations. In particular, configurations with $k = 2, 3, 4$ arrays of 1024 positions or a single array of 2048, 3072 and 4096 positions have been tested. The first simulation was done for the insertion time attack with the single array implementation. The attack was run on each insertion and the number of ones o was measured and used to estimate the FPP using $(\frac{o}{m})^k$. Figure 4 shows the results. Note that the attack does increase the FPP and that the simulation values match the theoretical estimates from equation 6. To better assess the increase on the FPP, the ratio of the FPPs when the filter is under attack and not are shown in Figure 5. The increase on the FPP grows with the number of insertions but is below 2x in all cases. This ratio does not depend on the number of hash functions k and therefore the attack seems to have the same impact regardless of the value of k.

The second simulation considered the time attack on a multiple array filter implementation and the results are summarized in Figure 6. It can be observed that again, the FPP increases due to the attack but the increment is not large. This is better seen in Figure 7 that shows the ratio of the FPP for the attack and no attack. Again, the ratio increases with the number of insertions but it is still below 2x in all cases. For the multiple array implementation, the ratio is exactly the same for all values of k as the attack only affects one of the arrays and the FPP is the product of the FPP of all the arrays. Therefore, the relative impact is the same regardless of the number of arrays k. Finally, the third simulation tests the delta FPP attack on the $k = 2, 3, 4$ array configuration. The size of the set F used to measure the FPP is fixed to the minimum of 10^6 elements or $\frac{10240}{FPP}$ to reduce the size for high FPP values. Instead, the size of the test set T on each insertion is fixed to $5 + 1024 \cdot FPP$ so that the test set is larger as FPP increases. Larger impacts on the FPP could be achieved by increasing the size of both F and T.

The results are shown in Figure 8. It can be observed that this attack has a larger impact on the FPP that again increases with

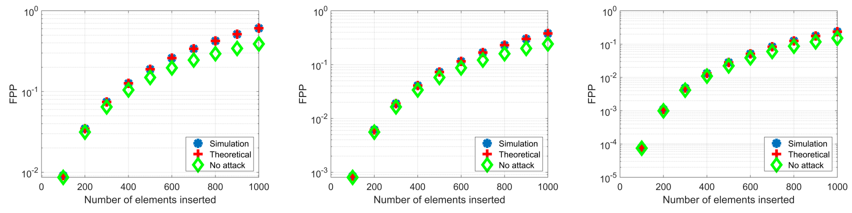

Fig. 6. False positive probability for the lookup time attack on a multiple array filter, k = 2 (left), k = 3 (middle) and k = 4 (right)

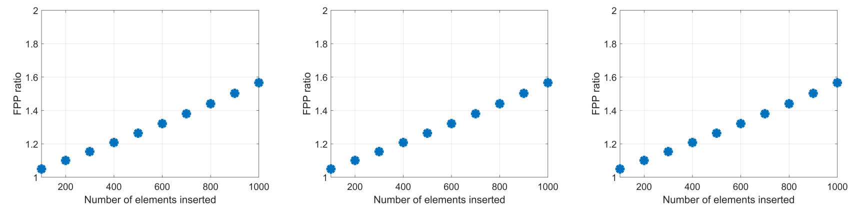

Fig. 7. Ratio of the False positive probability the lookup time attack versus no attack on a multiple array filter, k = 2 (left), k = 3 (middle) and k = 4 (right)

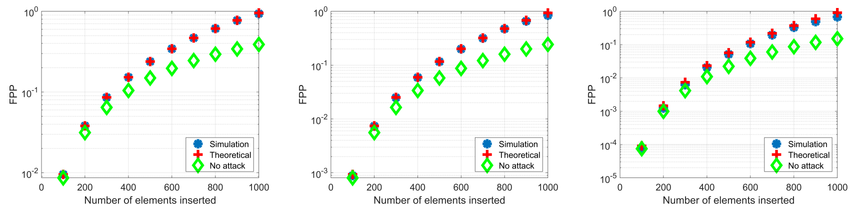

Fig. 8. False positive probability for the delta FPP attack on a multiple array filter, k = 2 (left), k = 3 (middle) and k = 4 (right)

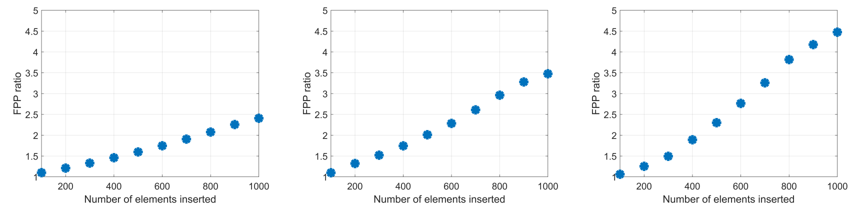

Fig. 9. Ratio of the False positive probability with delta FPP attack versus no attack on a multiple array filter, k = 2 (left), k = 3 (middle) and k = 4 (right)

the number of insertions. A better comparison can be made by looking at Figure 9 that shows the ratio of the FPP under attack to the normal operation FPP. It can be seen that the ratio gets to values over 2x,3x,4x for $k = 2, 3, 4$ respectively compared to less than 2x for the lookup time based attack. The impact on the FPP also increases with the number of insertions. Therefore, as expected, the delta FPP attack is more effective than the lookup time based attack. Looking at the value of k, the attack is more effective as k increases. This can be explained as in the case of the delta FPP attack, all arrays are affected by the attack. Therefore, the FPP of each array increases and the total that is obtained as the product increases more with more arrays.

V. Conclusion

In this paper, two pollution attacks against counting Bloom filters have been proposed. Unlike previous works that assume that the attacker knows or can infer the hash functions used in the filter, this work considers an attacker that has only a black box access to the filter. Such an attacker can perform only insertions and removals but has no knowledge of the filter implementation details. The analysis and results show that such an attacker can indeed pollute the filter and increase its false positive probability.

Acknowledgment

Pedro Reviriego was supported by the ACHILLES project PID2019-104207RB-I00 and the Go2Edge network RED2018-102585-T funded by the Spanish Ministry of Science and Innovation and by the Madrid Community project TAPIR-CM P2018/TCS-4496. Ori Rottenstreich was partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate, by the Alon fellowship, by German-Israeli Science Foundation (GIF) Young Scientists Program and by the Taub Family Foundation.

References

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," Communications of ACM, vol. 13, no. 7, pp. 422–426, 1970.

[2] S. Tarkoma, C.E. Rothenberg and E. Lagerspetz, "Theory and practice of Bloom filters for distributed systems," IEEE Communications Surveys and Tutorials, vol. 14, no. 1, pp. 131-155, 2012.

[3] A. Z. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," Internet Mathematics, vol. 1, no. 4, pp. 485–509, 2005.

[4] S. Geravand and M. Ahmadi, "Bloom filter applications in network security: A state-of-the-art survey," Computer Networks, vol. 57, no. 18, pp. 4047-4064, 2013.

[5] M. H. Haghighat, M. Tavakoli and M. Kharrazi, "Payload Attribution via Character Dependent Multi-Bloom Filters," in IEEE Trans. on Information Forensics and Security, vol. 8, no. 5, pp. 705-716, 2013.

[6] J. Grashöfer, F. Jacob and H. Hartenstein, "Towards Application of Cuckoo Filters in Network Security Monitoring," International Conference on Network and Service Management (CNSM), 2018.

[7] L. Fan, P. Cao, J. Almeida and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," IEEE/ACM Trans. on Networking, vol. 8, no. 3, pp. 281-293, 2000.

[8] 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.

[9] S. Kiss, É. Hosszu, J. Tapolcai, L. Rónyai and O. Rottenstreich, "Bloom Filter with a False Positive Free Zone," in Proc. IEEE Infocom, 2018.

[10] P. Reviriego and O. Rottenstreich, "The Tandem Counting Bloom Filter - It Takes Two Counters to Tango," in IEEE/ACM Trans. on Networking, vol. 27, no. 6, pp. 2252-2265, 2019.

[11] O. Rottenstreich, P. Reviriego, E. Porat and S. Muthukrishnan, "Constructions and Applications for Accurate Counting of the Bloom Filter False Positive Free Zone," in Proc. ACM Symposium on SDN Research (SOSR), 2020.

[12] S. A. Crosby and D. S. Wallach, "Denial of Service via Algorithmic Complexity Attacks," in Proc. USENIX Security Symposium, 2003.

[13] U. Ben-Porat, A. Bremler-Barr, H. Levy and B. Plattner, "On the Vulnerability of Hardware Hash Tables to Sophisticated Attacks," in Proc. IFIP NETWORKING, 2012.

[14] P. Reviriego and D. Ting, "Security of HyperLogLog (HLL) Cardinality Estimation: Vulnerabilities and Protection," in IEEE Communications Letters, vol. 24, no. 5, pp. 976-980, 2020.

[15] D. Clayton, C. Patton, and T. Shrimpton, "Probabilistic Data Structures in Adversarial Environments," in Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS), 2019.

[16] P. Reviriego and D. Larrabeiti, "Denial of Service Attack on Cuckoo Filter based Networking Systems", in IEEE Communications Letters, vol. 24, no. 7, pp. 1428-1432, 2020.

[17] L. Csikor et al, "Tuple space explosion: a denial-of-service attack against a software packet classifier," in Proc. ACM International Conference on Emerging Networking Experiments And Technologies (CoNEXT), 2019.

[18] M. Naor and Y. Eylon, "Bloom Filters in Adversarial Environments", ACM Trans. on Algorithms, vol. 15, no. 3, pp. 35:1–35:30, 2019.

[19] T. Gerbet, A. Kumar and C. Lauradoux, "The Power of Evil Choices in Bloom Filters," in Proc. IEEE/IFIP International Conference on Dependable Systems and Networks, 2015.

[20] K. Christensen, A. Roginsky and M. Jimeno, "A new analysis of the false positive rate of a Bloom filter," Information Processing Letters, vol. 110, no. 21, pp. 944–949, 2010.

[21] 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.

[22] R. J. Lipton and J. F. Naughton, "Clocked adversaries for hashing," Algorithmica, vol. 9, no. 3, pp. 239–252, 1993.

[23] A. Gelman and J. Hill, "Data Analysis Using Regression and Multilevel/Hierarchical Models", Chapter 20, Cambridge University Press, 2007.