The Power of Evil Choices in Bloom Filters

Thomas Gerbet
Université Joseph Fourier, France
e-mail: thomas.gerbet@e.ujf-grenoble.fr

Amrit Kumar, Cédric Lauradoux
INRIA, France
e-mail: {amrit.kumar, cedric.lauradoux}@inria.fr

***Abstract*—A Bloom filter is a probabilistic hash-based data structure extensively used in software including online security applications. This paper raises the following important question: Are Bloom filters correctly designed in a security context? The answer is no and the reasons are multiple: bad choices of parameters, lack of adversary models and misused hash functions. Indeed, developers truncate cryptographic digests without a second thought on the security implications. This work constructs adversary models for Bloom filters and illustrates attacks on three applications, namely SCRAPY web spider, BITLY DABLOOMS spam filter and SQUID cache proxy. As a general impact, filters are forced to systematically exhibit worst-case behavior. One of the reasons being that Bloom filter parameters are always computed in the average case. We compute the worst-case parameters in adversarial settings, show how to securely and efficiently use cryptographic hash functions and propose several other countermeasures to mitigate our attacks.**

Keywords*—Bloom filters; Hash functions; Digest truncation; Pre-image attack; Denial-of-Service;

I. INTRODUCTION

Bloom filter [1], is a space-time efficient probabilistic data structure for *set-membership query*. They are very popular among software developers since they often reduce the memory consumption of applications. While the goal of some of these applications is simply to provide a robust end-to-end service, Bloom filters also form a backbone of more critical and sensitive infrastructures such as malware/phishing detection tools and spam filters among many others.

In this work, we show that Bloom filters should be judiciously deployed as this seemingly "innocuous" data structure can be easily exploited and can be forced to behave as per the terms of an adversary. Our findings demonstrate that if Bloom filter parameters are ill-chosen, they are prone to severe *algorithmic complexity attacks* [2].

We present attacks on critical infrastructures built on Bloom filters. Our attacks rely on the vulnerability of weak hash functions or the inordinate use of secure hash functions. In fact, due to the hashing abstractions, software developers are often in the dilemma of selecting a "*good hash function*" for the concerned application. Considering the application distinctions, hash functions can be broadly classified into *non-cryptographic* and *cryptographic* ones. Clearly, using the latter to perform the task of the former entails an efficiency overhead. Developers are however tempted to employ non-cryptographic hash functions to boost the performance of their applications. Another temptation for developers related to hash function is digest truncation. Hash functions, in general, produce more bits than required by an application. Several bits of the digest are thus discarded. There might be several motivations to truncate digests: it could be a deliberate and arbitrary choice of the developer or it could be required by the underlying algorithm. Unfortunately, either of these temptations is a security sin and hence can be easily exploited.

We define adversary models for our attacks and present our findings on several applications. The main result of our adversarial model is the computation of the worst case error probability for Bloom filters. We show that if Bloom filter parameters are not chosen according to the worst case error probability, an adversary can pollute a filter with well chosen inputs, forcing it to deviate from its normal behavior. She can also query the filter to make it produce erroneous answers (or false positives). Once again, our attacks rely on a kind of forgery similar to finding pre-images and second pre-images of digests. The forgery is feasible either due to the use of non-cryptographic hash functions or due to the truncation of cryptographic digests: the bad habits of developers have not changed since [2]. Finally, we also show how to strengthen Bloom filters against adversaries. We explore the trade-off between the query time, the memory consumption and the security with respect to the proposed countermeasures.

The contribution of the paper is threefold. First, we describe adversary models for Bloom filters. We show how they can be polluted/saturated using pre-image attacks and how it increases the false-positive probability. Then, we show how to forge false-positives. In the adversarial settings, we have the liberty to assume that the inputs to the filter are non-uniformly distributed. This observation leads to our second contribution: we compute the worst case false-positive probability and obtain new equations for Bloom filter parameters. Finally, we provide techniques to save calls to cryptographic hash functions when used in a Bloom filter. To support our contributions, we provide three attacks on software applications based on Bloom filters: Bloom-enabled SCRAPY web spider, BITLY DABLOOMS spam filter and SQUID web cache. Our attacks retain some form of DoS and rely on forging Uniform Resource Locators (URLs) matching certain pre-image or second pre-image property.

The paper is organized in three parts. In the first part, we succinctly provide essential material on hash functions (Section II) and Bloom filters (Section III). In the second part, we first describe our adversary models (Section IV) and then proceed with the three illustrations. Section V presents SCRAPY and our pollution and query attacks against its *déjà vu* URL list. In Section VI, we extend our attack to a variant of Bloom filter: DABLOOMS, a data structure proposed by BITLY to filter malicious URLs. Section VII introduces SQUID proxy and describes an attack against its cache digest mechanism. The third part of the paper describes countermeasures (Section VIII) and the related work (Section IX).

II. Hash functions

A hash function compresses inputs of arbitrary length to digest/hash of fixed size. It is a very popular primitive used in algorithms [3] and in cryptography/security [4]. The design of a "*good hash function*" depends on the field of application. Non-cryptographic hash functions such as MurmurHash [5] or Jenkins hash [6] are designed to be fast, to uniformly distribute their outputs and to satisfy several other criteria such as the *avalanche test* [5]. The SMHASHER suite [5] provides a good overview of these functions and the tests they must satisfy.

The design of a cryptographic hash function is very different. Cryptographic hash functions are slower than their non-cryptographic siblings. Furthermore, a cryptographic hash function h must verify the following properties:

- **Pre-image resistance:** Given a digest d, it is computationally infeasible to find an input x, such that $h(x) = d$.
- **Second pre-image resistance:** Given an input x and the digest $h(x)$, it is computationally infeasible to find another input $x' \neq x$, such that $h(x) = h(x')$.
- **Collision resistance:** It is computationally infeasible to find two inputs $x \neq x'$ such that $h(x) = h(x')$.

Non-cryptographic hash functions are not designed to satisfy these properties [2], [7].

Let us consider a hash function h with ℓ-bit digests. The choice of ℓ is critical for cryptographic hash functions because the basic complexities for finding pre-image, second pre-image and collisions are 2^{ℓ}, 2^{ℓ} and $2^{\ell/2}$ respectively. A cryptographic hash function is considered secure if there is no attack with a lower complexity. The NIST recommendation for cryptographic hash functions are SHA-256, SHA-384, SHA-512 [9] and SHA-3 [10].

We provide in this paragraph the notion of *truncated digests*. In fact, many applications need to reduce the size of the full digest. Truncated digests must be carefully used as explained in [8], since it is commonly assumed that for a truncated digest of ℓ' bits the security is reduced at least to $2^{\ell'}$ (pre-image and second pre-image) and $2^{\ell'/2}$ (collision). If ℓ' is too small, computation of pre-image, second pre-image or collisions are feasible. In the following, we cover a popular application of hashing: Bloom filters.

III. Bloom filters

Bloom filters introduced by Bloom [1] is a space-efficient probabilistic data structure that provides an algorithmic solution to the set-membership query problem, which consists in determining if a given item belongs to a predefined set. To this end, Bloom filter offers a succinct representation of a set of items which can dramatically reduce space, at the cost of introducing *false positives*. If false positives do not cause significant problems, then Bloom filter may provide improved performance of an application.

Since its conception, Bloom filters have been used for various purposes in networking: resource routing [11], web caching [12] or filtering [13]. They have also been used in cryptography in designing efficient primitives such as searchable encryption [14] and private set intersection [15] among others. As a consequence, there is a significant literature on Bloom filters. Our contribution to Bloom filter theory (see Section VIII) is the analysis of false positive probability under adversarial settings, where an adversary may choose items to be inserted into the filter.

In the following, we recall the Bloom filter data structure, and in the next section, we explain how an adversary can pollute/saturate a filter or how he can generate items leading to false positives.

Description

Essentially, a Bloom filter is represented by a binary vector $\vec{z}$ of size m. In the following, we define the *support* of a vector $\vec{z}$ of size m, *i.e.*, $\vec{z} = (z_0, \ldots, z_{m-1})$ denoted by $\mathrm{supp}(\vec{z})$ as the set of its non-zero coordinate indexes:

$$\mathrm{supp}(\vec{z}) = \{i \in [0, m-1], z_i \neq 0\} \ .$$

We also denote $w_H(\vec{z})$, the Hamming weight of the filter $\vec{z}$, *i.e.*, the number of 1s in the filter. We use x with eventual subscripts to denote items inserted in the filter, while y with eventual subscripts to denote items queried to the filter.

In case of a classical Bloom filter, the vector $\vec{z}$ is initialized to $\vec{0}$. The filter is then incrementally built by inserting items of a set $\mathcal{S}$ by setting certain bits of the filter to 1. By checking if these bits are set to 1, one may verify the belonging of an item in the filter. A detailed description follows:

a) Insertion: An item $x \in \mathcal{S}$ is inserted into a Bloom filter by first feeding it to k independent hash functions $\{h_1, \ldots, h_k\}$ (supposed to be uniform) to retrieve k digests modulo m: $I_x = \{h_1(x) \bmod m, \ldots, h_k(x) \bmod m\}$. These reduced hashes give k bit positions of $\vec{z}$. Finally, insertion of x in the filter is achieved by setting the bits of $\vec{z}$ at these positions to 1. Since all the operations on digests are truncated modulo m in Bloom filters, for simplicity we omit $\bmod m$ in the rest of the paper.

b) Query: To determine if an item y belongs to $\mathcal{S}$, we check if y has been inserted into the Bloom filter $\vec{z}$. Achieving this requires y to be processed (as in insertion) by the same hash functions to obtain k indexes of the filter, $I_y = \{h_1(y), \ldots, h_k(y)\}$. If the bit of $\vec{z}$ at any of these indexes is 0, then the item is not in the filter, otherwise if $I_y \subseteq \mathrm{supp}(\vec{z})$, the item is present (with a small false positive probability).

Bloom filters can store items of arbitrary size using only m bits and the insertion/query runs in constant time: computation of k hash functions. However, this space and time efficiency of Bloom filter, comes at the cost of false positives.

c) False positive probability: False positives arise due to the collision on the reduced digests (see [11] for a detailed analysis). Let n be the number of insertions into the filter, then the probability of obtaining a false positive is:

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

For large m and relatively small k, we have:

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

We note that (1) is not the most accurate approximation of the false positive probability (see [16] for a more accurate result). However, it is often used in software implementations and hence we abide by it throughout the paper.

There are two competing forces behind the false positive probability. On the one hand, using more hash functions gives a higher chance to find a bit not set for an item which is not a member of the filter, while on the other hand, using fewer hash functions increases the fraction of bits not set in the filter and hence decreases the false positive probability. The optimal number of hash functions that minimizes the false positive probability is (see [11]):

$$k_{\text{opt}} = \frac{m}{n} \cdot \ln 2 \quad , \tag{2}$$

and the corresponding false positive probability satisfies:

$$\ln(f_{\text{opt}}) = -\frac{m}{n} \cdot (\ln 2)^2 \quad . \tag{3}$$

Another important result is on the expected number of set bits in the filter. Let X be the random variable representing the number of 0s in the filter. After the insertion of n random elements, it follows that:

$$\begin{aligned} \mathbb{E}(X) &= \sum_{i=0}^{m} i \cdot \mathbb{P}(X = i) \\ &= \sum_{i=0}^{m} i \cdot \binom{m}{i} p^i (1-p)^{m-i} \\ &= mp \ , \end{aligned} \tag{4}$$

where, $p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$.

Hence, in the optimal case, the expected number of 0s in the filter is $\frac{m}{2}$. Broder and Mitzenmacher [11] further show that the exact fraction of unset bits is extremely concentrated around its expectation, using a simple martingale argument. Specifically, they prove using the Azuma-Hoeffding inequality, that for any $\epsilon > 0$,

$$\mathbb{P}(|X - mp| \geq \epsilon m) \leq 2e^{-\epsilon^2 m^2 / nk} \quad . \tag{5}$$

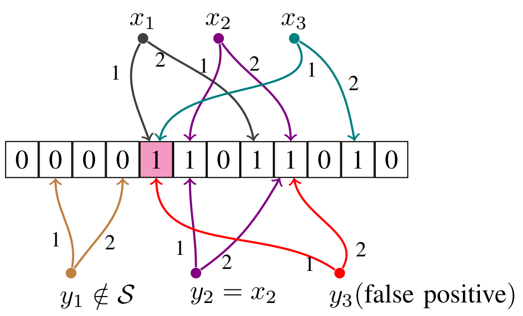

Fig. 1: Bloom filter with $m = 12$ and $k = 2$.

*Example 3.1 (**Bloom filter**):* Fig. 1 presents a Bloom filter of size $m = 12$ bits, constructed using $k = 2$ hash functions. The inserted set consists of 3 items, $\mathcal{S} = \{x_1, x_2, x_3\}$. A collision occurs on the first reduced digests of x_1 and x_3 (colored cell). Items y_1, y_2, y_3 are checked for belonging. The item $y_3 \notin \mathcal{S}$ is found to be present in the filter and hence is a false positive. Item y_1 however does not belong to $\mathcal{S}$.

The previous equations on the false positive rate of Bloom filter are well-established. They are valid as long as the digests of the inputs are uniformly distributed. We now see how these parameters behave under adversarial settings.

IV. Adversary Models

As a general principle, developers while designing applications built on a Bloom filter decide on the maximum number of elements to be inserted, the desired false positive probability and a hash function. Once these parameters are chosen, they can compute the filter size and the optimal number of hash functions using (2) and (3).

In this work, we assume that the Bloom filters are always deployed and maintained by trusted parties. This assumption is necessary, otherwise any bit of the filter can be tampered by the adversary. The scenario where Bloom filters are maintained by possibly untrusted parties is hence meaningless and has indeed led to big failures such as List Of All Friends (LOAF). LOAF (now discontinued) was designed to allow a user to send email messages along with his address book compressed in the form of a Bloom filter. The motivation behind sending address books was that the friends of my friends are trusted. Therefore, the Bloom filters of a user's friends can be used as a *whitelist* spam filter. When an email is received, the source address is checked against the Bloom filter. If it is present, the email is not marked as spam. Otherwise, it is suspicious and must be analyzed using a more complex spam filter. The trivial attack here is to send a fake Bloom filter (for instance, $\vec{z}$ where all the bits are set to 1) allowing a malicious user to whitelist any email address in the world.

We also assume that the implementation of the Bloom filter is public and known to the adversary. Moreover, we assume that the operations on the filter are always predictable. These hypotheses are usually verified in open source software.

Before describing our adversary models, we define *Pre-image* and *Second pre-image attacks* on Bloom filters. These attacks can be perceived as natural extensions of their hash function counterparts. Nevertheless, these notions require explicit treatment due to the subtleties introduced by the data structure.

Definition 4.1: **Pre-image for a Bloom filter.** Given a Bloom filter, $\vec{z}$, with only one item inserted into it, an associated pre-image for the filter is a string $y \in \{0,1\}^*$ with $I_y = \{h_1(y), \ldots, h_k(y)\}$ such that $I_y \subseteq \text{supp}(\vec{z})$.

Definition 4.2: **Second pre-image for a Bloom filter.** Given a Bloom filter, $\vec{z}$, with only one item $x \in \{0,1\}^*$ with $I_x = \{h_1(x), \ldots, h_k(x)\}$ inserted into it, an associated second pre-image for the filter is another string $y \neq x$ with $I_y = \{h_1(y), \ldots, h_k(y)\}$ such that $I_y \subseteq \text{supp}(\vec{z})$. The difference with respect to pre-image is that the item x is now given to the adversary.

Remark 4.1: Thinking along the lines of (second) pre-image attacks on hash functions, the complexity of finding a pre-image or a second pre-image for a Bloom filter should intuitively be $\frac{1}{m^k}$, for each hash function h_i is uniform over

$[0, m-1]$. However, since in case of Bloom filters, the order of hashes in the set I_y is not important, the probability of finding a (second) pre-image in case of Bloom filters is $\frac{w_H(\vec{z})^k}{m^k} = \left(\frac{w_H(z)}{m}\right)^k$. This holds due to the fact that the total number of (second) pre-images is $w_H(z)^k$, *i.e.*, all permutations of $\mathrm{supp}(\vec{z})$ with repetitions.

In the following, we define three adversaries for Bloom filters: *chosen-insertion adversary*, *query-only adversary* and *deletion adversary*.

A. Chosen-insertion Adversary

The first adversary can choose the items to be inserted in the Bloom filter. She can either add the items to the filter by herself or can arrange to make the trusted party do it for her. We consider two cases: in the first scenario the adversary controls all the items to be inserted in the filter and the second in which the filter is non-empty at the time of the attack and hence contains some already inserted items. The goal of this adversary in both the cases is to obtain a false positive probability which is higher than the one expected by the developer. This is achieved by increasing the number of set bits in the filter, which we refer to as a *pollution attack*. In the worst case and in case of certain types of Bloom filters, the adversary can set all the bits of the filter to one: this is referred to as a *saturation attack*.

By carefully choosing the items, it becomes possible to exceed the expected false positive probability, which eventually forces the application to deviate from its expected behavior. To be precise, a *polluting item* x should maximize the number of bits set to 1:

$$\begin{aligned} \forall i \neq j \in [1,k],\ h_i(x) \neq h_j(x)\ , \\ \forall i \in [1,k],\ h_i(x) \notin \mathrm{supp}(\vec{z})\ . \end{aligned} \tag{6}$$

In the above equation, $\vec{z}$ denotes the filter after each insertion. After n such insertions into the filter, the number of set bits attains the value kn. An illustrative example is presented in Fig. 2. Items x_1, x_2, x_3, x_4 are so chosen such that all $h_j(x_i)$ are distinct, for $j \in [1,2]$ and $i \in [1,4]$. The colored cells are the bits set to 1 by the adversary after crafting the corresponding items.

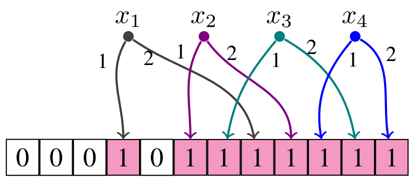

Fig. 2: Chosen-insertion adversary ($k = 2$).

Hence, with carefully selected items, the number of set bits in the filter can be made larger than the expected value. In fact, as previously mentioned (see (5)), with the optimal parameters: m, n, and k_{opt}, the expected number of set bits in the filter is (refer to (2)):

$$\begin{aligned} \frac{m}{2} &= \frac{nk_{\text{opt}}}{2\ln 2} \\ &\approx 0.72nk_{\text{opt}}\ . \end{aligned}$$

Comparing it to nk_{opt} bits set to 1 by the adversary, she increases the number of 1s in the filter by 38%. For a chosen k, the adversary sets nk bits of the m-bit filter to 1, hence the false positive probability achieved in the attack is:

$$f^{\text{ADV}} = \left(\frac{nk}{m}\right)^k\ . \tag{7}$$

Fig. 3 shows how the false positive probability behaves under a chosen-insertion attack. We choose a Bloom filter of size $m = 3200$ with a capacity of 600 items. Equation 2 for optimal parameters gives $k_{\text{opt}} \approx 4$, and $f_{\text{opt}} = 0.077$. When the number of inserted items is low, f^{ADV} and f are superimposed until $\lceil\frac{\sqrt{m}}{k}\rceil$ items have been added. This is due to the *Birthday paradox*: the first $\sqrt{m}$ items' indexes are likely to be all different. It implies that the adversary does not need to compute pre-images for the first $\lceil\frac{\sqrt{m}}{k}\rceil$ items she wants to insert. Let us now consider the threshold probability of $f_{\text{opt}} = 0.077$. This threshold is reached after 600 insertions if the indexes are uniformly distributed. An adversary however can attain this value after only 422 well-chosen insertions. After 600 chosen insertions, she obtains a false positive probability of $0.314 \approx 4f_{\text{opt}}$. In certain scenarios, an adversary may not be able to control all the insertions into the filter. To this end, let us consider the case of 400 normal insertions followed by insertions chosen by the adversary. The threshold of 0.077 is then reached after 510 insertions. At the end of 600 insertions, she obtains a false positive probability of $0.17 \approx 2f_{\text{opt}}$.

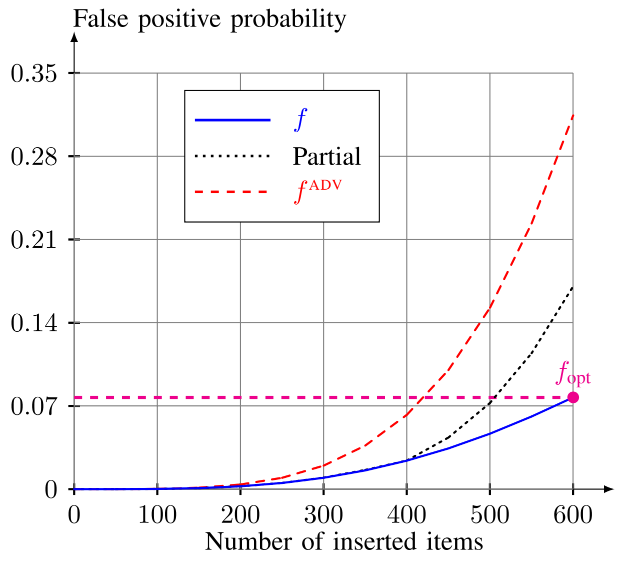

Fig. 3: False positive probability as a function of inserted items ($m = 3200$, $k = 4$ and $f_{\text{opt}} = 0.077$).

Now, we consider saturation attacks. The expected number of items to fully saturate a filter is: $\lfloor\frac{m \log m}{k}\rfloor$. This directly follows from the *Coupon collector's problem*: given m coupons, find the expected number of draws with replacement before having drawn each coupon at least once. In the case of Bloom filter, k coupons are drawn in each draw. In contrast to this result, our attack only requires $\lfloor\frac{m}{k}\rfloor$ items, since each item sets

k bits to 1. This allows the adversary to gain a factor of $\log m$ to achieve saturation.

An important question is to estimate the feasibility of forging a polluting item, *i.e.*, an item which satisfies (6). To this end, let us consider a filter $\vec{z}$ of Hamming weight $w_H(\vec{z}) = W$, for an integer $W > 0$. We want to insert a polluting item x to $\vec{z}$. There are $\binom{m-W}{k}$ ways to choose such an item. The probability to find such an item is then:

$$\frac{\binom{m-W}{k}}{m^k} \quad .$$

If we wish to insert n polluting items in an empty filter, there are $\binom{m-(n-1)k}{k}$ ways to choose the n-th item. If m is large compared to k, finding polluting elements is much simpler than finding pre-image or second pre-image for Bloom filter. We illustrate in Section V, the practical cost of forging polluting items.

B. Query-only Adversary

The second adversary cannot insert items into the filter. However, she knows the current state of the filter or a part of it. Similar to a chosen-insertion adversary, she can either generate queries by herself or force the trusted party to query on her behalf. The query-only adversary can have two distinct objectives. She can either craft items that generate false-positives hence force the application to err, or that her items' digests are well-distributed in the filter leading to latency.

There could be several motivations for an adversary to make the filter generate false positives. A typical scenario would be to attack applications incorporating Bloom filters, but which do not tolerate false positives at all. In such applications, Bloom filters conjointly work with a remote mechanism (for example a database storing the items) to get rid of false positives. In general, when the item is found to be present in the filter, the remote mechanism provides a confirmation of a true positive. This ensures that the application does not err. Generation of large number of false positives would lead to *false positive flooding* enabling an adversary to hit the second mechanism and attempt to mount a DoS.

For a given Bloom filter $\vec{z}$, a query-only adversary wants to generate an item y such that:

$$\forall i \in [1, k],\ \ h_i(y) \in \text{supp}(\vec{z}) \ \ . \qquad (8)$$

Fig. 4 shows different examples of false positives: Items y_1, y_2, y_3 are detected as being present in the filter while these items were never inserted in the filter. Items y_1 and y_3 are particularly interesting. While y_1 verifies $h_1(y_1) = h_2(y_1)$, y_3 satisfies $h_1(y_3) = h_2(x_1)$ and $h_2(y_3) = h_1(x_1)$.

Knowing the positions of the 1s, the adversary has $W^k = (w_H(\vec{z}))^k$ choices to forge a false positive. There are 25 choices in Fig. 4. The probability to forge a false positive y is: $\left(\frac{W}{m}\right)^k$.

It is also possible to consider a query-only adversary making *dummy queries*. The adversary queries for items which are not in the filter. Let y be an item chosen by the adversary, then it must satisfy:

$$\forall i \neq j \in [1, k],\ \ h_i(y) \neq h_j(y) \ \ ,$$
$$\forall i \in [1, k-1]\ , h_i(y) \in \text{supp}(\vec{z}) \text{ and } h_k(y) \notin \text{supp}(\vec{z}) \ \ .$$

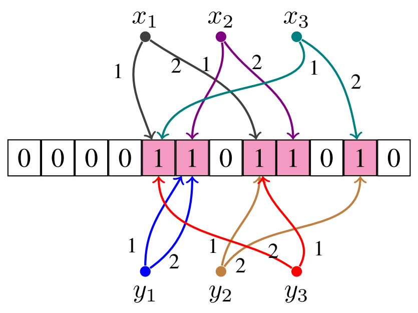

Fig. 4: Crafted false positives: $w_H(\vec{z}) = 5$.

The probability of finding such an item is:

$$\frac{(m - W) \cdot \binom{W}{k-1}}{m^k} \quad .$$

The idea of this attack is to make the query as expensive as possible. It targets applications with very large Bloom filters and forces the party running the Bloom filter to make more memory accesses and more computations than expected. The goal of the adversary is to reach the worst case execution time for each query.

C. Deletion Adversary

Bloom filters basically support insertions and queries. However, certain variants of Bloom filters also allow deletion, a typical example is *Counting Bloom filter* [11], where instead of being a bit vector, the filter is an array of counters, which are incremented/decremented when items are inserted/deleted to/from the filter. *False negatives* are the drawbacks of these variants [17]. Hence, an adversary who does not control the insertions into the filter can nevertheless forge an item and make it delete from the filter.

We assume that the filter is fully or at least partially known to the deletion adversary. The goal of the adversary is then to create false negatives: she wants to make an item x with $I_x = \{h_1(x), \ldots, h_k(x)\}$ disappear from the filter. To this end, the adversary needs to find item(s) x' in the filter with $I_{x'} = \{h_1(x'), \ldots, h_k(x')\}$ such that $I_{x'} \cap I_x \neq \phi$. Consequently, deleting the item x' from the filter would decrease at least one counter for x and hence the item x also gets deleted. The probability to find such an x' is given by:

$$1 - \left(\frac{W - |I_x|}{m}\right)^k \quad .$$

We highlight that deletion of an item may require other deletions. These deletions may remove several other items from the filter as a side effect.

To summarize, our attacks against Bloom filter can be classified depending on the bit of the filter targeted by the adversary. The chosen-insertion adversary generates pre-image on the 0 of a Bloom filter. The query-only adversary generates pre-image on the 1 of a Bloom filter and sends queries for which the bits at the first $k-1$ indexes are set to 1, while the last index could be either 0 or 1. The deletion adversary targets certain 1 in the filter. In each case, we consider brute force search: an item is selected at random and its k indexes are computed. If the bit in the filter at any of these indexes is already set to 1 or 0 depending on the adversary, the item is discarded and a new one is tried.

Feasibility of our attacks: We highlight that our attacks on Bloom filter based applications are feasible since either non-cryptographic hash functions are used or cryptographic digests are truncated. While non-cryptographic hashes can be easily broken, truncation of cryptographic digests drastically reduces the attack complexity. Furthermore, the probability of success of our attacks is higher than the generally accepted (second) pre-image attacks on hash functions. Table I presents the summary of the success probability of our attacks.

TABLE I: Comparative summary of our attacks for a Bloom filter of Hamming weight W.

Attack	**Probability**
(Second) pre-image (hash function)	$\frac{1}{2^\ell}$
(Second) pre-image (Bloom filter)	$\left(\frac{W}{m}\right)^k \leq \left(\frac{k}{m}\right)^k$
Pollution	$\frac{\binom{m-W}{k}}{m^k}$
False positive forgery	$\left(\frac{W}{m}\right)^k \leq \left(\frac{1}{2}\right)^k$
Dummy query	$\frac{(m-W)\cdot\binom{W}{k-1}}{m^k}$
Deletion	$1-\left(\frac{W-k}{m}\right)^k \leq 1-\left(\frac{W-\lvert I_x\rvert}{m}\right)^k$

From the computed probabilities (see Table I), it is possible to order each attack in terms of their computational feasibility. The pollution attack has the highest success probability. The most difficult attack is the deletion one. Between these two attacks, lies the difficulty of mounting query attacks which depend on the number of items inserted in the filter.

In order to put our adversary models to test and measure the impact of our attacks, in the following sections we focus on three critical applications of Bloom filters: web spider, spam filter and web cache.

V. Blinding Some Species of (Web)Spiders

A *spider*, also known as a *robot* or a *crawler* is a mechanism for bulk discovering or downloading of publicly available web pages from the world wide web. Recently, crawlers have been appositely employed to design automatic tools to track the web for personalized topics and notify clients of pages that match these topics [18]. Interested readers may consult the survey by Olston and Najork [19] for more applications and further details.

In this section, we first present Scrapy [20], a high level screen scraping and web crawling framework, used to crawl websites and extract structured data from their pages. It has been employed for wide range of applications, varying from data mining, monitoring, to information processing and automated testing, among others[1]. In the sequel, we present attacks against Scrapy.

[1]http://scrapy.org/companies/

A. Scrapy: simplified architecture

Scrapy extracts data from HTML and XML sources with the help of XPath and provides built-in support for sanitizing the scraped data using a collection of reusable filters shared between the spiders. Scrapy is capable of performing crawling at different scales and depths. For instance, it can crawl specific domains or pages related to a given topic of interest.

Execution of a crawling process recursively performs the following steps. These steps are common to any spider and hence are not specific to Scrapy.

1) Select a URL form the list of scheduled ones.
2) Fetch the URL.
3) Archive the results.
4) Select URLs of interest and add to scheduling list.
5) Mark the current URL as visited.

Step (5) is crucial to eliminate previously visited URLs. In light of this, different data structures have been deployed, varying from a list, to hash table and Bloom filters among others. In case of Scrapy, the default duplicate filter to mark visited URLs uses URL fingerprints of length 77 bytes in Python 2.7. Hence to scrape a site with 2 million pages, the list of visited URLs might grow up to 2M $\times$ 77B = 154 MB. In order to scrape 100 such domains, memory of 15GB space would be required. The developers have left the possibility to use alternative data structures open. Considering the low memory footprint of Bloom filters, developers have proposed to employ them in Scrapy[2]. Bloom filters are indeed used in various other crawlers, for instance Heritrix [21]. However, we show in the sequel that Bloom filters must be judiciously used, else it might invite serious attacks.

B. Attacks

Scrapy provides a possibility to integrate any alternative data structure to mark visited URLs. It by default uses hash table but Bloom filters are also supported to reduce the memory footprint. The implementation is based on the python module pyBloom[3]. It is a popular implementation of Bloom filter in python and can be easily plugged into Scrapy (version 0.24). pyBloom proposes the following cryptographic hash functions: SHA-512, SHA-384, SHA-256 and MD5. The indexes corresponding to a URL are generated by a hash function with sufficiently large digest and seeded using a known deterministic salt. In turn, the choice of the hash function depends on the size of the filter.

An adversary may easily mount a chosen-insertion attack to pollute the spider's filter. She generates or owns an initial web page. She then creates links on it with well-chosen URLs, such that each URL upon insertion in the filter sets previously unset bits to 1. Whence, the false positive probability of the filter increases to (7), if her web page is the starting point of the crawling process. Once the initial web page is completely crawled, Scrapy upon starting to crawl any other page not owned by the adversary would detect it to have been already visited with probability (7). Hence, it would terminate the

[2]http://alexeyvishnevsky.com/?p=26

[3]https://github.com/jaybaird/python-bloomfilter

crawling process while falsely believing that the web page has already been crawled. *We have blinded the spider.* We note that this attack is not specific to SCRAPY, but extends to any spider employing Bloom filter in an insecure manner.

We performed empirical tests to determine the cost of finding polluting URLs. All the experiments in this work were performed on a laptop computer with CPython 2.7.3 interpreter. Our target platform is a 64-bit processor laptop computer powered by an Intel i7-3520M processor, with 4 cores running at 2.90 GHz. The machine has 4MB cache and is running 3.5.0-35 Ubuntu Linux. Throughout the paper, we have employed fake-factory[4] (version 0.2) a python package to generate fake but human readable URLs.

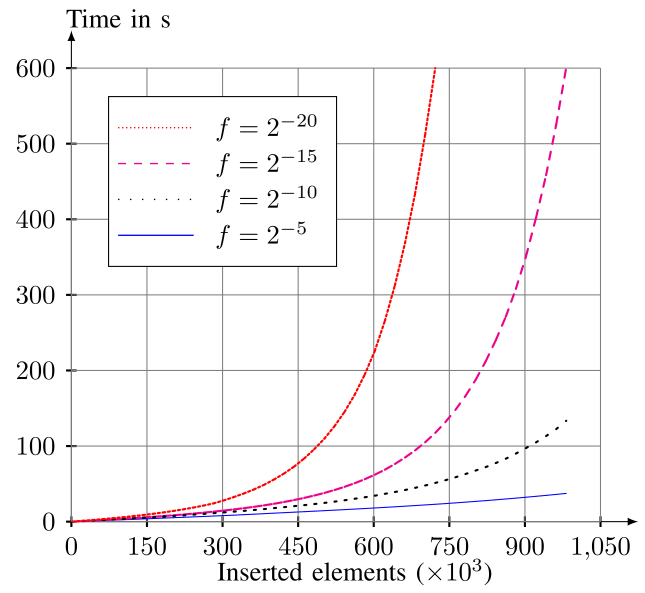

Fig. 5: Cost of creating polluting URLs.

Fig. 5 describes the time needed to forge 10^6 URLs to pollute SCRAPY's filter. We choose the number of items $n = 10^6$ and the false-positive probability: 2^{-5}, 2^{-10}, 2^{-15} and 2^{-20}. The size m, k and the hash functions are automatically computed by PYBLOOM. We observe that the time needed to find the polluting items grows exponentially. This is a direct consequence of the probability of finding a polluting item (see Table I). Indeed, the number of ways to choose the n-th polluting item decreases exponentially as n increases, which explains the exponential increase in time to find these polluting items. For $f = 2^{-5}$, it takes 38 seconds to generate 10^6 URLs, while for $f = 2^{-20}$, it takes 2 hours.

It is also possible to mount query-only attacks. The adversary does not want to have some of her pages to be crawled by SCRAPY. She does not trust the `robots.txt` file since many spiders are impolite, *i.e.*, they do not respect `robots.txt` policies. To hide her secret pages from SCRAPY, she publishes a few pages (the decoys) with some links to her secret pages (the ghosts). She chooses the URLs of her ghost pages to make SCRAPY think that they have already been seen (false positive). All her pages are organized in a web tree such that the leaves are the ghosts and the root (entry point) and all the nodes are decoys. In the most simple setting, we have one ghost and a single decoy. Given a URL, finding a decoy (or the ghost) has the probability $\left(\frac{k}{m}\right)^k$. This task is time consuming as shown in Fig. 6. It presents the cost to generate a false positive as a function of the filter occupation (the ratio between the current number of insertions to the capacity of the filter, which is 10^6 in our case). In order to generate false positives, random items were generated and tested against the filter.

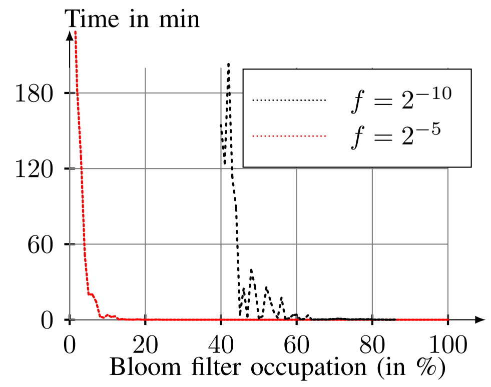

Fig. 6: Cost of creating ghost URLs.

A more simple solution consists in generating from a URL several decoys to hide the ghost. We need $\Theta(k \log k)$ random values to hide a URL (coupon collector's problem). Fig. 7 illustrates an example for $k = 3$.

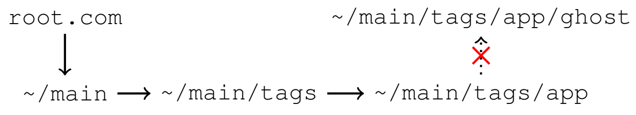

Fig. 7: A root domain `root.com` with the page tree `main, main/tags, main/tags/app` is chosen. Once SCRAPY recursively visits the decoys, the ghost page `main/tags/app/ghost` is considered as already visited. The ghost page hence remains hidden.

VI. BITLY SPAM FILTER: DABLOOMS

URL shortening services such as BITLY are targeted by cybercriminals to lure users into phishing/malware traps (see [22]). In order to prevent misuse, these services apply filters to detect malicious URLs. The current filtering policies were studied in [23]. DABLOOMS is an experimental data structure proposed by BITLY to prevent the shortening of malicious URLs.

A. *Dablooms in a nutshell*

Classical Bloom filters have two limits. They do not support deletion and the number of items must be defined a priori. DABLOOMS is the combination of two variants of Bloom filters which overcome these issues: counting Bloom filters [11] and scalable Bloom filters [24].

In a counting Bloom filter, the bits used in a classical Bloom filter are replaced by small counters. These counters are incremented/decremented during insertions/deletions.

[4]https://pypi.python.org/pypi/fake-factory/0.2

DABLOOMS uses 4-bit counters. Counting Bloom filters offer the possibility of deletion but they have several drawbacks. The size of the filter is increased compared to the original design and false negatives exist. False negatives can be the results of deletion or insertions (counter overflow).

Scalable Bloom filters can work with an arbitrary number of items while keeping the false positive probability reasonable at the cost of the memory size. A scalable Bloom filter is a collection of Bloom filters created dynamically. To each filter is associated an insertion counter. When the counter reaches a certain threshold δ, a new filter is created with a counter set to 0. Let us denote f_i to be the false positive probability of the i-th filter for $i \geq 0$. In DABLOOMS and [24], at a given moment, the data structure consists of λ filters with error probabilities satisfying:

$$\forall\ 1 \leq i \leq \lambda - 1,\ f_i = f_0 r^i,\ \text{with } 0 < r < 1\ .$$

In DABLOOMS, r is equal to 0.9. When an item is queried to the filter, it is searched in all the filters. The overall false positive F is defined in [24] by:

$$F = 1 - \prod_{i=0}^{\lambda-1} (1 - f_i)\ .$$

The hash function used in DABLOOMS is MurmurHash [5] combined with a trick from Kirsch and Mitzenmacher [25] to reduce the number of calls to MurmurHash.

B. Attacks

We describe the effect of a pollution attack carried out by a chosen-insertion adversary. In the sequel, we also present a deletion attack.

As described earlier, the adversary first generates phishing or malware websites with well-chosen URLs. Then, she needs to make BITLY include her URLs in DABLOOMS. The most obvious choice to achieve this is to flood the web with her malicious URLs and wait until it is spotted by BITLY. Another option is to register her URLs directly to anti-phishing websites such as PHISHTANK[5] to get her URLs recognized as the ones hosting malicious contents and to eventually get included in DABLOOMS.

Fig. 8 describes the effect of pollution on DABLOOMS. There are $\lambda = 10$ filters and each filter can include $\delta = 10000$ items and we have chosen $f_0 = 0.01$ and $r = 0.9$. We consider two cases for pollution:

- All the filters are polluted (dashed curve).
- Only the last i filters are polluted. This number varies from 1 to 9 (dotted curves).

We observe from Fig. 8 that, if only the last filter is polluted, *i.e.*, the first 9 filters remain untouched, then the blue curve and the dashed curve superimpose for most of the part. Clearly, as more and more filters are polluted, the false positive probability achieved by the attacker gets increased.

Deletion attacks on DABLOOMS are also feasible due to the presence of counting Bloom filters. The forgery of the required URLs is straightforward since MurmurHash can be inverted in constant time (see [7]). Furthermore, by exploiting counter overflow, an adversary can force DABLOOMS to create an *empty filter*, even after all the n items have been inserted. Let us define $a = nk \bmod 16$ and b such that $nk = a + 16b$. The goal of the adversary is to obtain after n insertions a filter set to 0 everywhere except one counter set to the value a. All the other insertions make overflow b other counters. The insertion counter of the filter may say it is full while none of the values inserted will be detected in the future. Such a filter is a complete waste of memory. Empty filters make DABLOOMS bigger and useless.

[5]www.phishtank.com

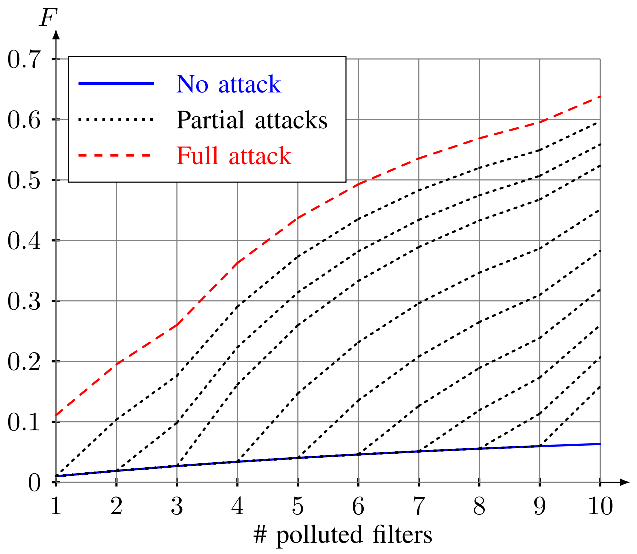

Fig. 8: Polluting DABLOOMS. The last curve in black from left represents the false positive probability achieved when only the last filter is polluted.

VII. YET ANOTHER ATTACK ON SQUID WEB PROXY

Web proxies are means to efficiently handle the daunting task of managing web traffic and alleviating network bottleneck. These proxies reduce the bandwidth consumption by relying on caching and reusing frequently requested web pages. Achieving the full benefits of the proxy mechanism requires them to co-operate and share their caches. SQUID [26] is a notable example of a caching proxy leveraging *cache digest* [27], a summary of the contents of the *Internet Object Caching Server*. A SQUID server first stores the fetched pages in a hash table and then at regular intervals of 1 hour, transforms it into space-efficient cache digest which could later be shared with the peers. When a request for a URL is received from a client, a cache can use digests from its peers to determine if any of them have previously fetched the URL. The cache then requests the object from the closest peer.

Crosby and Wallach [2] have previously attacked the hash table in version 2.5STABLE1. Our work extends their attack to cache digests in version 3.4.6.

Attacking Cache Digests

Cache digests in SQUID are built as a Bloom filter. Although the protocol design allows for a variable number of hash functions, in practice, SQUID employs 4 hash functions for the "sake of efficiency" and dissuades developers from using more. Furthermore, instead of computing 4 independent hash functions over a URL, SQUID generates a 128-bit MD5 hash of the key (comprising of the URL and the HTTP retrieval method) and then splits it to obtain the indexes of the filter. For unexplained reasons, the filter parameters are not optimal. In fact, to insert n items in the filter, the considered filter size is $5n+7$, instead of the optimal size $6n$. This choice leads to a higher false positive probability. For instance, if $n = 200$, the entailed false positive probability is 0.09 instead of the optimal value 0.03, hence an increase by a factor of 3.

Attacking SQUID's cache digest requires us to set up a client, two SQUID proxies as siblings in a LAN and an HTTP server responding to every GET request of the client received via one of these proxies. When the proxies are configured as siblings, they work together and exchange cache digests to avoid unnecessary hits between them. Each unnecessary hit between proxies costs bandwidth and adds latency to the client's request. We suppose that the proxies are honest, this avoids the trivial attack where a proxy transmits a fake filter to its peers. Our attack relies on a malicious client who generates fake URLs and asks one of the proxies to fetch these pages. These URLs pollute the cache digest of the concerned proxy. Once the cache digest of the first proxy is fully built, we start querying the second proxy and count the cache digest false positive (*i.e.*, unnecessary hits to the first proxy). Each false positive adds at least one round-trip time (10 ms in our setup) between the two caches to the response delay. With 100 URLs added to pollute a clean cache digest (51 URLs are already present when the cache is totally clean), the filter size is 762 bits. We observed that out of 100 queries, cache pollution increases the false positives hits to 79% in contrast to 40% when the cache is unpolluted.

VIII. COUNTERMEASURES

We explore different solutions to render Bloom filters resistant to adversaries. A first and obvious solution could be to use an alternative data structure such as hardened hash table. However, we lose the benefit of Bloom filters: low memory footprint. However, an alternative consists in recomputing all the parameters of a Bloom filter from (7). It gives us the worst case parameters. It increases the memory consumption but we can keep using non-cryptographic hash functions. It defeats chosen-inputs adversary but not the query-only one. Another solution consists in using keyed hash functions. It defeats all classes of adversaries but increases the query time.

A. Worst-case Parameters for Bloom Filters

While designing an application based on Bloom filter, the developer has two constraints: m, the memory available and n, the capacity of the filter. Based on these parameters, he minimizes the false positive probability and obtains the optimal number of hash functions to employ (2). The adversary on the other hand tries to increase this false positive probability by inserting specially crafted items (7).

An adaptive approach to the pollution attacks could be to choose the parameters such that the adversary's advantage could be minimized. In other words, the developer would fix m and n as earlier, but now instead of finding k that minimizes the false positive probability of the filter, he chooses k that minimizes the false positive probability envisaged by the adversary, *i.e.*, $f^{\text{ADV}} = \left(\frac{nk}{m}\right)^k$. Differentiating with respect to k gives:

$$\frac{\partial f^{\text{ADV}}}{\partial k} = f^{\text{ADV}} \cdot \left(1 + \ln\left(\frac{nk}{m}\right)\right) \ .$$

The zero of the derivative is:

$$k_{\text{opt}}^{\text{ADV}} = \frac{m}{en} \ , \tag{9}$$

and the second derivative test confirms that $k_{\text{opt}}^{\text{ADV}}$ is the point of minimum of f^{ADV}. The false positive probability achieved by the adversary would then be:

$$f_{\text{opt}}^{\text{ADV}} = e^{-m/en} \ . \tag{10}$$

However, given n and m, and assuming that the optimal value $k = k_{\text{opt}}^{\text{ADV}}$ is used by the developer, the actual false positive probability of the filter in this worst case scenario is obtained by using this k in (1):

$$f_{\text{worst}} = \left(1 - e^{-1/e}\right)^{\frac{m}{ne}} \ , \tag{11}$$

$$\ln\left(f_{\text{worst}}\right) = -0.433\frac{m}{n} \ . \tag{12}$$

From (2), (3) and the obtained results, we highlight that:

$$\frac{k_{\text{opt}}}{k_{\text{opt}}^{\text{ADV}}} = e\ln 2 = 1.88 \quad \text{and} \quad \frac{f_{\text{worst}}}{f_{\text{opt}}} = (1.05)^{\frac{m}{n}} \ .$$

Consequently, the adaptive approach requires a considerably lesser number of hash functions and hence, the associated Bloom filter would be more time efficient at the cost of a slightly higher false positive probability.

Let us now consider a scenario where the developer for a given m and n chooses $k_{\text{opt}}^{\text{ADV}}$ to obtain the false positive probability of the filter satisfying (12). However, if the developer wishes to obtain the same false positive probability, while only fixing n and not the size of the filter, the new filter size m' can be obtained from (3). Comparing these two filter sizes for the same false positive probability gives: $\frac{m'}{m} = 4.8$.

Hence, we observe that the filter size increases almost by a factor of 5 in the latter case. With this solution, developers can keep their fast non-cryptographic hash functions but at the cost of a larger Bloom filter. It defeats chosen-insertion adversaries but not the query-only ones.

B. Probabilistic Solutions

The first countermeasure proposed to defeat algorithmic complexity attacks was to use *universal hash functions* [28]. These have been empirically studied in case of hash tables in [2]. We propose to use *Message Authentication Code*(MAC) [4], which has been considered as an overkill until now.

We assume that the server running the Bloom filter works with a MAC. The server chooses a key for the MAC at the beginning and uses it to insert/check items submitted by clients. The key is chosen from a large set, therefore it is computationally infeasible for an adversary to either guess the key or compute pre-images for all the possible values. Thus, the adversaries defined in Section III can not make a brute force search to satisfy (6) or (8) because the function is no longer predictable. Hence, attacks against SCRAPY, DABLOOM and SQUID can easily be mitigated since in all these applications, the Bloom filter is stored on the server side.

The crucial question that remains is the impact of probabilistic solutions on the processing time of a Bloom filter. We focus on the benchmark of cryptographic hash functions and MAC for Bloom filters. It is commonly admitted that those functions are too slow compared to non-cryptographic hash functions and hence are rarely employed. We decided to make a fair comparison between the two classes of functions. We observed that in several implementations of Bloom filters, there are k calls to the hash function with different salts. This in fact forces many bits of the digests to remain unused. One could reuse these bits to reduce the number of calls to the expensive hash function and obtain comparative results. As all cryptographic hash functions pass the NIST test suite to check their uniformity, we can assume that all the bits of the digest can be used. The indexes of an item require to have $k \cdot (\lfloor \log_2 m \rfloor + 1)$ bits of digests. Fig. 9 shows the domain of application of the different cryptographic hash functions (and thus their respective HMAC construction) for different false positive probabilities. A single call to SHA-512 or HMAC-SHA-512 is enough to build any Bloom filter with optimal parameters for $f \leq 2^{-15}$ and m smaller than one GB. For $f \geq 2^{-20}$, we need to make several calls to the hash function.

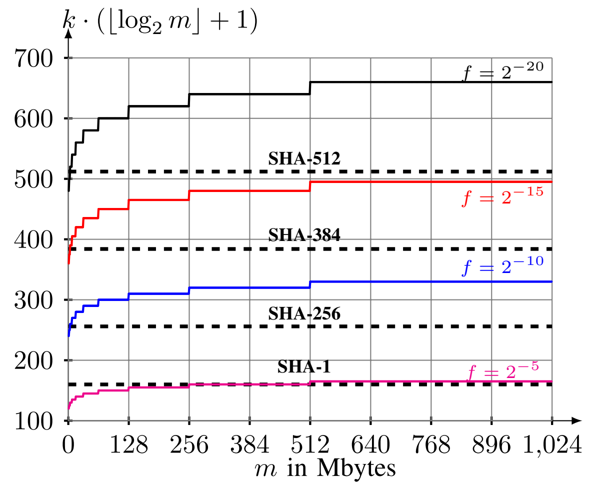

Fig. 9: Domain of application of hash functions.

Table II presents a comparative summary of the cost of reusing bits of different cryptographic hash functions and HMAC-SHA-1. Our implementations are in C and the cryptographic primitives are taken from the OpenSSL (version 1.0.1) library. The filter has a false positive probability of 2^{-10} and contains 1 million elements, creating a filter of size 2.48MB. The items inserted in the filter are of 32 bytes long (corresponding to SHA-256 prefixes). Clearly, recycling of bits performs significantly better than the naive $k = 10$ calls to the hash function. HMAC-SHA-1 is costlier due to the 2 implicit calls to SHA-1. Compared to the popular choice of MurmurHash, recycling is a better and more secure alternative. SIPHASH [7], a non-cryptographic keyed hash function outperforms HMAC-SHA-1 by a factor 7 without recycling. Our technique reduces the gap to only a factor 4 making MAC affordable for Bloom filters.

TABLE II: Time to query a filter.

	Timing (μs)		
Hash function	**Naive**	**Recycling**	**Speedup ($\times$)**
MurmurHash-32 [5]	0.7	-	-
MD5	5.9	0.28	21
SHA-1	6	0.29	20.6
SHA-256	51	0.49	104
SHA-384	53.3	0.78	68
SHA-512	53.6	0.8	67
HMAC-SHA-1	11.8	1.2	9.83
SIPHASH [7]	1.7	0.3	5.66

MACs have the advantage to defeat all the adversaries and to keep the original parameters of Bloom filters. The drawback is that the query time is increased compare to non-cryptographic hash functions.

IX. RELATED WORK

Our work is closely related to *algorithmic complexity attacks* which were first introduced in [29] and formally described by Crosby and Wallach in [2]. The goal of algorithmic complexity attacks is to force an algorithm to run in the worst case execution time instead of running in average time. For a hash table, it means that the search operation runs in linear time instead in constant time. The general impact of these attacks was DoS [39](due to the significant consumption of resources) or the creation of covert channels [40]. Algorithmic complexity attacks have successfully been mounted against many data structures and algorithms: hash tables [2], [7], [30]–[32], quick-sort [33], skip-lists [34], machine learning [35], regular expressions [36], packet analyzers [37] and file-systems [38]. In most of these works, weak non-cryptographic hash functions were responsible for the attacks. Our work contributes to algorithmic complexity attacks in three aspects. First, it extends algorithmic complexity attacks to a data structure having false positives. Second, our pollution attacks and false positive flooding are simpler and relatively much easier to mount compared to the pre-image attacks described by Crosby and Wallach [2]. They choose a bucket identifier, id in the hash table and search for keys $x_1, \dots, x_n$ which go to this bucket, *i.e.*, $h(x_1) = \cdots = h(x_n) = id$ (multiple pre-images). Third, we provide different adversary models to analyze Bloom filters. These models encompass previous attacks and hence can be re-used to analyze other data structures. For instance, the attacks on hash table and skip-lists combine a chosen-insertion and query-only adversary. For attacks against quick-sort and regular expressions, it is a chosen-insertion adversary.

The problem of saturation of Bloom filters is well-identified in the software development community, typically in the situations where the number of insertions is not controlled. We show the best strategy for an adversary to pollute and saturate a filter in the chosen-insertion model and present the entailed complexity. Similarly, the use of Bloom filters is often criticized by the web crawler community due to the intrinsic false-positives (see [19]). Our paper materializes those criticisms and shows how fast they may arise.

Several countermeasures have been proposed to prevent algorithmic complexity attacks. Crosby et al. [2] suggest using universal hash functions [28]. They are used in the Bloom filter included in the HERITRIX web spider [21]. Aumasson et al. have applied several tools from cryptanalysis in [7] to attack non-cryptographic hash functions. They also propose a new function SIPHASH [7] as an efficient and secure alternative. It is interesting to notice that Venkataraman et al. [41] design probabilistic counting algorithms for fast detection of superspreaders, and pay attention to the implementation: *"We use the OPENSSL implementation of the SHA-1 hash function, picking a random key during each run, so that the adversary cannot predict the hashing values."* However, they also advert the usage of non-cryptographic hash functions: *"For a real implementation, one can use a more efficient hash function."* The authors in [42] have also used keyed hash functions to compute packet statistics in the presence of an adversary.

An abounding literature is devoted on designing secure Bloom filters [43] or private Bloom filters [44]. These replace the usual hash functions by group ciphers. Bellovin et al. [44] use Pohlig-Hellman encryption for instance. These cryptographic primitives resist pre-image and second pre-image attacks at the cost of a higher computational time. Kerschbaum [45] has used partially homomorphic encryption to make private queries to a Bloom filter. Unfortunately, the scheme is computationally intensive, and hence can not be adapted to build high performance data structures.

Särelä et al. [46] study the security of multicast protocols based on Bloom filters. In these protocols, the Bloom filter represents the group entities. The authors propose a technique called *BloomCasting*, which enables controlled multicast packet forwarding. In order to control the entities who may send/receive packets to/from a group, the authors suggest to use keyed hash functions (among other possible alternatives such as a secret permutation) for Bloom filters.

Our idea of recycling bits of cryptographic digests is inspired by Nyberg accumulator [47]. Independently from the Bloom's work, Nyberg has proposed a data structure to solve the set-membership query problem. Without going into the details, the solution relies on a *"long hash function"*: large digests which are afterwards reduced to obtain the accumulator. Our attacks against Bloom filters do not hold against Nyberg accumulator because it would require finding pre-images for full digests of cryptographic hash functions. However, Nyberg's accumulator is larger than Bloom filters (by a factor $\log n$) which makes it less attractive to developers. We keep the idea of *"long hash function"* by recycling bits of cryptographic digests in Bloom filters. To obtain long digests, Nyberg suggests to combine cryptographic hash functions and pseudo-random generators. We salt and recycle bits.

The recommendations of the NIST [8] on the usage of cryptographic hash functions is a reference document for truncated digests. However, developers are still ignoring the threats of short truncated digests. RFC 6920 [48] also dedicates a few paragraphs to the threats associated with truncated digests, but it gives no specific consequences. Our work gives a concrete example of these threats.

X. CONCLUSION

Lumetta et al. have explained in [49] how the *power of two choices* can reduce Bloom filter's false positive probability. The technique has the advantage of keeping the filter size unchanged at the cost of more hashing. Our work demonstrates the *power of evil choices* in Bloom filters.

It is surprising that despite the far reaching attacks of Crosby et al. [2] and the several ensuing booster shots [7], [31], non-cryptographic hash functions are still present in sensitive software solutions. If non-cryptographic hash functions are prevalent, cryptographic ones are rarely used properly. After our work, we wish that software developers would become more cautious before truncating hashes or employing weak hash functions.

There are three natural extensions to our work: variants of Bloom filters, probabilistic counting algorithms [50] and *extensible-output hash functions*. Variants of Bloom filters are interesting to extend our adversary models. Many of these variants either add new functionalities or improve on the efficiency. Another important question related to these filters is the existence of other solutions having a better worst-case false positive probability than the classical one.

Probabilistic counting algorithms [50] are very popular in computing statistics on large datasets with a reduced memory. Hashing (and the truncation that comes along) is the core mechanism. It will be interesting to analyze their existing implementations in an adversarial setting.

The ideal hash function for Bloom filters should be an efficient and secure keyed hash function with extensible output. Recently, the NIST has released the SHA- 3 standard [10]. It includes two extensible-output functions SHAKE-128 and SHAKE-256. We look forward to knowing if they can be keyed and how do they perform with Bloom filters.

ACKNOWLEDGMENT

This research was supported by the Labex PERSYVAL-LAB (ANR–11-LABX-0025) and the project-team SCCyPhy. The authors thank Pascal Lafourcade for reading a draft of the paper and for providing his feedback.

REFERENCES

[1] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[2] S. A. Crosby and D. S. Wallach, "Denial of Service via Algorithmic Complexity Attacks," in *USENIX Security Symposium*. Washington, USA: USENIX Association, December 2003, pp. 3–3.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[4] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. CRC Press, Inc., 1996.

[5] A. Appleby, "SMHasher - Test your hash functions." 2010, https://code.google.com/p/smhasher/.

[6] R. Jenkins, "A Hash Function for Hash Table Lookup," 1996, http://www.burtleburtle.net/bob/hash/doobs.html.

[7] J.-P. Aumasson and D. J. Bernstein, "SipHash: A Fast Short-Input PRF," in *Progress in Cryptology - INDOCRYPT 2012*, ser. Lecture Notes in Computer Science 7668. Kolkata, India: Springer, December 2012, pp. 489–508.

[8] Q. Dang, "Recommendation for Applications Using Approved Hash Algorithms," National Institute of Standards & Technology, Tech. Rep. SP 800-107 Revision 1, august 2012.

[9] National institute of standards and technology, "Secure Hash Standard (SHS)," National Institute of Standards & Technology, Tech. Rep. FIPS PUB 180-4, march 2012.

[10] ——, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," National Institute of Standards & Technology, Tech. Rep. FIPS PUB 202, may 2014, draft.

[11] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, 2005.

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

[13] J. W. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed content delivery across adaptive overlay networks," in *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication - ACM SIGCOMM 2002*. Pittsburgh, PA, USA: ACM, August 2002, pp. 47–60.

[14] E.-J. Goh, "Secure Indexes," Cryptology ePrint Archive, Report 2003/216, 2003, http://eprint.iacr.org/2003/216/.

[15] C. Dong, L. Chen, and Z. Wen, "When private set intersection meets big data: an efficient and scalable protocol," in *ACM SIGSAC conference on Computer & communications security - CCS '13*. Berlin, Germany: ACM, November 2013.

[16] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. H. M. Smid, and Y. Tang, "On the false-positive rate of Bloom filters," *Inf. Process. Lett.*, vol. 108, no. 4, pp. 210–213, 2008.

[17] D. Guo, Y. Liu, X. Li, and P. Yang, "False negative problem of counting bloom filter," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 22, no. 5, pp. 651–664, May 2010.

[18] "Giga Alert," http://www.gigaalert.com/products.php.

[19] C. Olston and M. Najork, "Web Crawling," *Foundations and Trends in Information Retrieval*, vol. 4, no. 3, pp. 175–246, 2010.

[20] "Scrapy," http://scrapy.org/.

[21] "Heritrix," https://webarchive.jira.com/wiki/display/Heritrix/Heritrix.

[22] N. Nikiforakis, F. Maggi, G. Stringhini, M. Z. Rafique, W. Joosen, C. Kruegel, F. Piessens, G. Vigna, and S. Zanero, "Stranger danger: exploring the ecosystem of ad-based URL shortening services," in *International World Wide Web Conference, WWW '14*. Seoul, Republic of Korea: ACM, April 2014, pp. 51–62.

[23] N. Gupta, A. Aggarwal, and P. Kumaraguru, "bit.ly/can-do-better," IIT Kanpur, February 2014, poster presented at Security Privacy Symposium - SPS 2014.

[24] P. S. Almeida, C. Baquero, N. M. Preguiça, and D. Hutchison, "Scalable Bloom Filters," *Inf. Process. Lett.*, vol. 101, no. 6, pp. 255–261, 2007.

[25] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better Bloom filter," *Random Struct. Algorithms*, vol. 33, no. 2, pp. 187–218, 2008.

[26] D. Wessels, *Squid: The Definitive Guide* . O'Reilly Media, 2004.

[27] A. Rousskov and D. Wessels, "Cache digests," in *Computer Networks and ISDN Systems*, 1998, pp. 22–23.

[28] L. Carter and M. N. Wegman, "Universal Classes of Hash Functions (Extended Abstract)," in *ACM Symposium on Theory of Computing - STOC*. Boulder, CO, USA: ACM, May 1977, pp. 106–112.

[29] A. Peslyak, "Designing and Attacking Port Scan Detection Tools," *Phrack Magazine*, vol. 8, no. 453, p. 13, 1998, http://phrack.org/issues/53/13.html#article.

[30] N. Bar-Yosef and A. Wool, "Remote Algorithmic Complexity Attacks against Randomized Hash Tables," in *International Conference on Security and Cryptography - SECRYPT 2007*. Barcelona, Spain: Springer Berlin Heidelberg, July 2007, pp. 117–124.

[31] A. Klink and J. Wälde, "Multiple implementations denial-of-service via hash algorithm collision," Open Source Computer Security Incident Response Team, Tech. Rep. oCERT advisory 2011-003, march 2011.

[32] U. Ben-Porat, A. Bremler-Barr, H. Levy, and B. Plattner, "On the Vulnerability of Hardware Hash Tables to Sophisticated Attacks," in *Networking (1)*, ser. Lecture Notes in Computer Science, vol. 7289. Springer, 2012, pp. 135–148.

[33] M. D. McIlroy, "A killer adversary for quicksort," *Softw. Pract. Exper.*, vol. 29, no. 4, pp. 341–344, Apr. 1999.

[34] D. Bethea and M. K. Reiter, "Data Structures with Unpredictable Timing," in *European Symposium on Research in Computer Security - ESORICS 2009*, ser. Lecture Notes in Computer Science, vol. 5789. Saint-Malo, France: Springer, September 2009, pp. 456–471.

[35] L. Huang, A. D. Joseph, B. Nelson, B. I. P. Rubinstein, and J. D. Tygar, "Adversarial machine learning," in *ACM Workshop on Security and Artificial Intelligence, AISec 2011, , 21, 2011*. Chicago, IL, USA: ACM, October 2011, pp. 43–58.

[36] S. A. Crosby, "Denial of Service through Regular Expressions," in *USENIX Security Symposium: Work-In-Progress Reports*. Washington, USA: USENIX Association, December 2003, p. 1.

[37] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos, "Tolerating Overload Attacks Against Packet Capturing Systems," in *USENIX Annual Technical Conference*. Boston, MA, USA: USENIX Association, June 2012, pp. 197–202.

[38] X. Cai, Y. Gui, and R. Johnson, "Exploiting Unix File-System Races via Algorithmic Complexity Attacks," in *IEEE Symposium on Security and Privacy - S&P 2009*. Oakland, California, USA: IEEE Computer Society, 2009, pp. 27–41.

[39] J. Mirkovic and P. L. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms," *Computer Communication Review*, vol. 34, no. 2, pp. 39–53, 2004.

[40] X. Sun, L. Cheng, and Y. Zhang, "A Covert Timing Channel via Algorithmic Complexity Attacks: Design and Analysis," in *IEEE International Conference on Communications, ICC 2011*. IEEE, June 2011, pp. 1–5.

[41] S. Venkataraman, D. X. Song, P. B. Gibbons, and A. Blum, "New streaming algorithms for fast detection of superspreaders," in *Network and Distributed System Security Symposium, NDSS 2005*. San Diego, CA, USA: The Internet Society, 2005.

[42] S. Goldberg, D. Xiao, E. Tromer, B. Barak, and J. Rexford, "Path-quality monitoring in the presence of adversaries," in *International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2008*. Annapolis, MD, USA: ACM, June 2008, pp. 193–204.

[43] R. Nojima and Y. Kadobayashi, "Cryptographically Secure Bloom-Filters," *Transactions on Data Privacy*, vol. 2, no. 2, pp. 131–139, 2009.

[44] S. Bellovin and W. R. Cheswick, "Privacy-Enhanced Searches Using Encrypted Bloom Filters," in *DIMACS/PORTIA Workshop on Privacy-Preserving Data Mining*. Piscataway, NJ, USA: DIMACS/PORTIA, March 2004, pp. 274–285.

[45] F. Kerschbaum, "Public-Key Encrypted Bloom Filters with Applications to Supply Chain Integrity," in *Data and Applications Security and Privacy XXV - 25th Annual IFIP WG 11.3 Conference, DBSec 2011*, ser. Lecture Notes in Computer Science 6818. Richmond, VA, USA: Springer, July 2011, pp. 60–75.

[46] M. Särelä, C. E. Rothenberg, A. Zahemszky, P. Nikander, and J. Ott, "BloomCasting: Security in Bloom Filter Based Multicast," in *Nordic Conference on Secure IT Systems, NordSec 2010*, ser. Lecture Notes in Computer Science, vol. 7127. Espoo, Finland: Springer, 2010, pp. 1–16.

[47] K. Nyberg, "Fast Accumulated Hashing," in *Fast Software Encryption - FSE 1996*, ser. Lecture Notes in Computer Science, vol. 1039. Cambridge, UK: Springer, February 1996.

[48] S. Farrell, D. Kutscher, C. Dannewitz, B. Ohlman, A. Keranen, and P. Hallam-Baker, "Naming Things with Hashes," Internet Requests for Comments, RFC Editor, RFC 6920, April 2013, http://tools.ietf.org/html/rfc6920.

[49] S. S. Lumetta and M. Mitzenmacher, "Using the Power of Two Choices to Improve Bloom Filters," *Internet Mathematics*, vol. 4, no. 1, pp. 17–33, 2007.

[50] P. Flajolet, "Theory and practice of probabilistic counting algorithms (abstract of invited talk)," in *Workshop on Analytic Algorithmics and Combinatorics - ANALC 2004*. New Orleans, LA, USA: SIAM, January 2004, p. 152.