SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2017; **47**:421–441
Published online 23 June 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2420

CloudEyes: Cloud-based malware detection with reversible sketch for resource-constrained internet of things (IoT) devices

Hao Sun[1,*,†], Xiaofeng Wang[1], Rajkumar Buyya[3] and Jinshu Su[1,2]

[1]*College of Computer, National University of Defense Technology, Changsha, Hunan,410073, China*
[2]*National Key Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha, Hunan, 410073, China*
[3]*Cloud Computing and Distributed Systems Laboratory, Department of Computing and Information Systems, University of Melbourne, Melbourne, VIC 3010, Australia*

SUMMARY

Because of the rapid increasing of malware attacks on the Internet of Things in recent years, it is critical for resource-constrained devices to guard against potential risks. The traditional host-based security solution becomes puffy and inapplicable with the development of malware attacks. Moreover, it is hard for the cloud-based security solution to achieve both the high performance detection and the data privacy protection simultaneously. This paper proposes a cloud-based anti-malware system, called CloudEyes, which provides efficient and trusted security services for resource-constrained devices. For the cloud server, CloudEyes presents suspicious bucket cross-filtering, a novel signature detection mechanism based on the reversible sketch structure, which provides retrospective and accurate orientations of malicious signature fragments. For the client, CloudEyes implements a lightweight scanning agent which utilizes the digest of signature fragments to dramatically reduce the range of accurate matching. Furthermore, by transmitting sketch coordinates and the modular hashing, CloudEyes guarantees both the data privacy and low-cost communications. Finally, we evaluate the performance of CloudEyes by utilizing both the campus suspicious traffic and normal files. The results demonstrate that the mechanisms in CloudEyes are effective and practical, and our system can outperform other existing systems with less time and communication consumption. Copyright © 2016 John Wiley & Sons, Ltd.

Received 14 October 2015; Revised 14 March 2016; Accepted 25 May 2016

KEY WORDS: reversible sketch; suspicious bucket cross-filtering; signature-based; malware detection; cloud-based

1. INTRODUCTION

The explosive development of Internet of Things (IoT) is leading a unprecedented revolution in the physical and cyber world. It is envisaged that the number of interconnected devices will exceed 7 trillion by 2025, with an estimate of about 1000 devices per person [1]. Such an enormous amount will deeply impact our digital lives in many application domains [2], for example, transportation, healthcare and so on, as depicted in Figure 1. However, targeted attacks caused by malicious software (malware) increase rapidly every year and expose more interest in the IoT devices. The McAfee Labs indicate attacks on the IoT devices will increase rapidly because of hypergrowth in the number of connected objects, poor security hygiene and the high value of data on these devices [3]. Various threats which exist for decades, like Spam, Privacy leak, Botnet, Distributed denial-of-service and Advanced persistent threat (illustrated in Figure 1), are still rampant in the IoT paradigm. For example, even an innocuous fridge can be employed to launch security attacks by sending spam mails [4].

*Correspondence to: Hao Sun, College of Computer, National University of Defense Technology, Changsha, Hunan 410073, P.R.China.
†E-mail: haosunlight@163.com

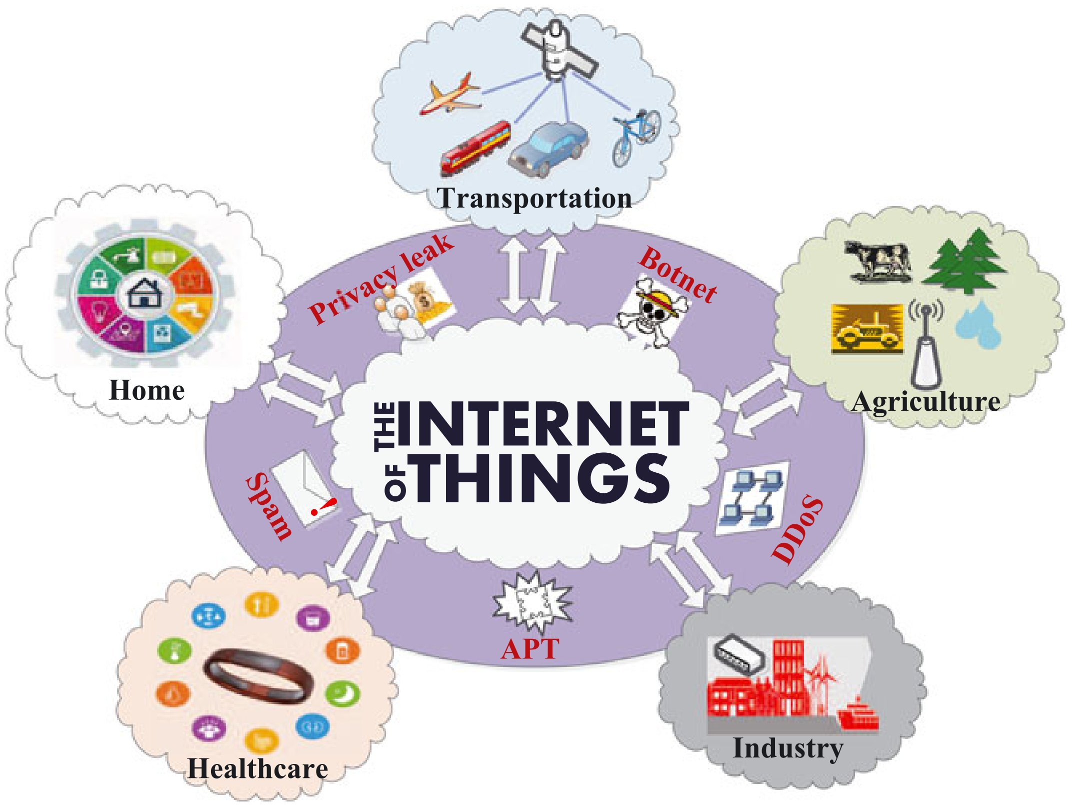

Figure 1. Applications and attacks in the Internet of Things. APT, Advanced persistent threat; DDoS, Distributed denial-of-service.

Hence, it is urgent to provide a trusted and one-stop security solution to take care of data security and privacy in those resource-constrained devices.

To defend against various malware, signature-based detection approach still plays an important role and takes up a large proportion after decades of development in both industry and academic research [5]. It is based on the theory that the crux of various malware, called signature, is generally unchangeable and can be detected at the early stage of propagation though the amount of malware samples is limited [6]. This approach is implemented by scanning and checking if a file contains the contents which match the known signatures. There are several commonly used and effective signature matching algorithms, such as Aho–Corasick [7] and Wu–Manber [8]. The efficiency of detection depends on several measures, such as the number of signatures, the accuracy and time consumption of matching, which are the primary motivations behind many of the researches done in this field.

1.1. Research challenges

Two primary kinds of signature-based anti-malware approach have been deployed according to their infrastructures in state-of-the-art technology. The first kind is host-based security system which installs detection agent in the users' devices and updates the signature database to ensure timely and complete security protection. ClamAV [9] is an open-source anti-virus system most widely used and many reformative works based on it are recently proposed, such as GrAVity [10]. However, these systems have become increasingly puffy with the development of malware attacks [11, 12] and do not suit for resource-constrained devices. The problems mainly embody in the following areas: (1) heavy resources consumption caused by the growing number of signatures which leads to low detection performance, such as memory, time, and database updating; (2) many mechanisms applied to improve detection performance, such as matching acceleration that benefits from the improvements in hardware speeds [10, 13], are paradoxical for resource-constrained devices. (3)

the characteristic of these systems that they must be updated at a very high frequency to provide up-to-date protection may give the malware attackers opportunity to compromise the hosts [14].

The other kind is cloud-based security system [15] which places different types of detection agents over the cloud servers and offers security as a service. Generally speaking, a user of the service can upload any type of file and receive a report about the malware that might be contained in the file [16–18]. This newly developed framework is lenitive and cost-saving for resource-constrained devices. However, most of existing cloud-based anti-malware technologies cannot solve all the following problems simultaneously: (1) security vendors are designed to directly expose or deliver the signature databases to the clients which is unwillingness for the vendors, and actually it is possible to ulteriorly lighten the consumption of clients, such as SplitScreen [5]; (2) users have to upload the whole file contents which may result in some important information (e.g., location and password) leakage, such as CloudAV [17]; (3) the communication consumption between the server and client should be further optimized which is significant for the improvement of detection efficiency, especially for the IoT devices. To conclude, it is hard to achieve high performance of security detection and data privacy protection simultaneously.

1.2. Research motivations

To address the problems earlier, the proposed system should be cloud-based and privacy preserving which are our main research motivations. In this paper, we propose CloudEyes, a cloud-based anti-malware system which utilizes the effective properties of reversible sketch to provide efficient security service and reliable data privacy protection for resource-constrained devices. Specifically, we make the following contributions:

(i) For the cloud server, a novel signature-based detection mechanism, called suspicious bucket cross-filtering, is proposed based on the structure of reversible sketch. It can provide retrospective and accurate orientations of malicious signature fragments which dramatically cut down the time and computation consumption in signature-based malware matching. To the best of our knowledge, no previous work has implemented similar endeavor.
(ii) For the client, a lightweight scanning agent is implemented to rapidly identify the suspicion of file content according to the digest of reversible sketch. It shows high applicability for resource-constrained devices with the advantage of sharp reduction of matching range and the capability of avoiding accurate matching of whole file content.
(iii) A balanced interaction mechanism is design to protect the data privacy for both client and server, and reduce the communication consumption. The client transmits the sketch coordinates of suspicious file segments, instead of the whole file content, to the cloud after fast scanning. As for the cloud server, modular hashing of malicious signature fragments are sent back to the client, rather than the signature database.

We analyse the accuracy of the proposed mechanism theoretically to prove its validity and veracity with appropriate parameters. Our implementation of CloudEyes consists of roughly 2.5K lines of C/C++ code for client and 4.5K for server which makes it easily applied to the resource-constrained devices. In addition, we evaluate the system by normal files and suspicious traffic captured from campus network with the number of signatures ranges from 460,000 to 3,700,000. Statistical results show that CloudEyes outperform ClamAV and SplitScreen with lower time consumption and smoother increment when scanning increasing number of samples. Moreover, the communication consumption in CloudEyes is on average 12.6 times smaller than that in SplitScreen.

The preliminary version of this paper has previously appeared in [19]. In this paper, we have substantially improved and extended the previous version. The most significant extensions include multiple hash functions adopted in each hash table for efficient detection (Section 4.2.1), modular hashing of signature fragments for reducing the memory cost in the cloud server (Section 4.2.2) and corresponding analysis of false positive it brings (Section 5.3.3), fast scanning and suspicious bucket cross-filtering for availably screen out the culprit signature (Section 4.3), and finally a series of performance studies for demonstrating how the proposed mechanisms enhance our system (Section 6). In addition, we specify the motivations and workflow of our system (Section 3).

The remainder of this paper is organized as follows: Section 2 introduces related work about signature-based malware detection. Section 3 presents the preliminaries and the architecture of our work. Section 4 gives a detail description about the signature-based detection mechanism, followed by accuracy discussion of the proposed mechanisms in Section 5. Section 6 shows the experimental results and analysis. Finally, we conclude the paper in Section 7.

2. RELATED WORK

Signature-based malware detection remains important and technically reliable after decades of development in anti-malware industry. Most researches can be divided into two kinds, host-based and cloud-based, according to where the main detection agents are placed.

ClamAV [9] is the most widespread and representative open-source host-based malware detection system. The latest database (main v.55 and daily v.19688) approximately contains 3,700,000 signatures consist of MD5 and regular expression signatures. Input files content are sequentially matched with the signature database when scanning. If a known signature is successfully matched, the file is claimed to be infected by malware. The matching algorithms adopted are primarily Aho–Corasick [7] and Wu–Manber [8].

Recently, several efforts to improve the detection performance based on host have been proposed. Hash-AV [13] proposes a malware scanning technique which aims to take advantage of improvements in CPU performance. It utilizes hashing functions that fit in L2 caches to speed up the exact pattern matching algorithms in ClamAV. GrAVity [10] is a massively parallel anti-malware engine which utilize the good performance of GPUs to accelerate the process of scanning. Hardware implementations provide better performance, but it is always impracticable for the resource-constrained devices, such as mobile phones and wearable devices. Deepak *et al.* [20] design a signature matching algorithm which is well suited in mobile device scanning, but its testing signatures are limited by fixed byte and the performance declines with the growth of signatures amount.

Cloud servers provide high-performance computation support to reduce the match consumption in malware scanning which is the main limitation of signature-based mechanism. Now, it is attracting lots of security vendors to start to deploy their cloud solutions, like Trend Micro, Panda Security and Kaspersky Lab.

CloudAV [17] first puts forward the notion of cloud-based malware scanning in academic research and the author apply their strategy to a mobile environment [21]. It runs a local cloud service consists of heterogeneous anti-virus engines running in parallel virtual machines and uses an end-user agent to transfer suspicious files to the cloud to be checked by all anti-virus engines. CloudAV achieves high detection rate, yet obviously, exposes the sensitive data which compromise users privacy. Similar researches like ThinAV [16] propose a lightweight anti-malware for android which utilize third party online malware scanning services, the users also need to submits size-restrained applications for scanning. CloudSEC [22] move the analysis and correlation of network alerts into network cloud which also consists of plenty autonomous anti-malware agents. Secloud [23] designs a generic framework for smartphone security that can be used to perform various powerful intrusion analysis solutions, but it suffers the problems as mentioned earlier of communication consumption and exposing the client to malware.

SplitScreen [5] designs a distributed anti-malware system based on ClamAV to speed up the malware scanning. SplitScreen design its first scanning mechanism based on Bloom filter [24] to perform slight comparisons with file data and reduce the size to be accurately matched. However, bloom filter is not reversible which is similar to sketch data structure because of the multiple-to-one nature of hashing functions, so it does not store any information about the fragments. Actually, the first scanning is so coarse-grained that the client still spends plenty of time and computation in exact pattern matching. Our study results show SplitScreen averagely spends 74.3 percent of its time in accurate pattern matching about 65 percent of pending files with small caches.

3. PRELIMINARIES AND SYSTEM ARCHITECTURE

In this section, we first introduce two main preliminaries of our work, then describe the architecture and workflow of our system, followed by the list of frequently used notations.

3.1. Preliminaries

Sketch structure is an aggregation method which maps diverse data streams into uniform vectors based on the Turnstile Model [25]. Let $I = \alpha_1, \alpha_2, \ldots,$ be a sequential input stream during a given time interval. Each item $\alpha = (\alpha_i, \mu_i)$ consists of a key $\alpha_i \in \{0, 1, \ldots, n-1\} \Leftrightarrow [n]$, and a value $\mu_i \in R$. The model assigns a time varying signal $U[\alpha_i]$ for each key $\alpha_i \in [n]$, and update $U[\alpha_i]$ with an increment of μ_i if a new item (α_i, μ_i) arrives.

Most researches based on sketch are applied to analysis of basic elements in flow, such as source and destination IP/Port. Reversible sketch (RS for short) [26] is the representative one and the infrastructure our system based on. It is based on the k-ary sketch data structure consists of H hash tables of size m, which k comes from the use of size k hash tables, that is, $m = k$. The hash functions adopted in the hash tables are chosen randomly from a universal hash-function family. The basic methodology of RS is detecting the heavy changes of data stream by summarizing the IP information into two schedule-based sketches and finding the suspicious keys whose updates cause the significant changes in these two time periods.

Our model design is inspired by RS structure whose methodology can be applied in identifying malicious data fragments from large amount of suspicious data. However, there are two different properties between the content and address element of data stream. Firstly, the size of content is distinct and various which results in more enormous quantity of keys. Our basic solution is to initialize the signatures and suspicious data to fragments with uniform length, and the digests of RS are utilized to achieve dramatic quantity reduction in scanning. The second and more important difference is distribution property. In RS, the distribution of IP addresses and ports are analyzed to reveal the heavy changes of data flows which are closely related to attacks, such as DoS and SYN flooding. While the distribution of the data fragments is less meaningful to reveal malware information which means the methodology of detecting heavy buckets needs proper modification. In our system, one k-ary reversible sketch and the corresponding digest are maintained to store the information of signatures which are the crux to define the data fragments to be malicious or not. The heavy buckets (suspicious reversible buckets in our work) are located by the result of fast scanning and the malicious keys (signature fragments) in the heavy buckets are selected by suspicious buckets cross-filtering.

Our goal is to design an efficient security system for resource-constrained devices, called CloudEyes, which achieves high performance of security detection and data privacy protection simultaneously, as we conclude in Section 1. The design of CloudEyes is inspired by SplitScreen [5], but differs from it on two significant aspects. First, we employ reversible sketch structure containing the information of suspicious signature fragments for malware detection. It is more efficient than Bloom filter structure because of needless to accurately match the whole contents of suspicious files. Second, we give consideration to the perspectives of both anti-malware vendors and end-users. Given the rapid incremental trend of signature volume and the security vendors' unwillingness of directly exposing malware signature databases which are their core profit and competitiveness, the system opts to transmit the sketch coordinates of file fragments and modular hashing of malicious signature fragments between the client and cloud server which cut down the communication consumption simultaneously.

3.2. System architecture: An overview

To break out of high time consumption, which is primarily caused by the enormous quantity of signatures, CloudEyes adopts the reversible sketch structure for effective representation and orientations of signatures. Additionally, it designs balanced interactive mechanism to protect the data privacy and reduce the communication consumption.

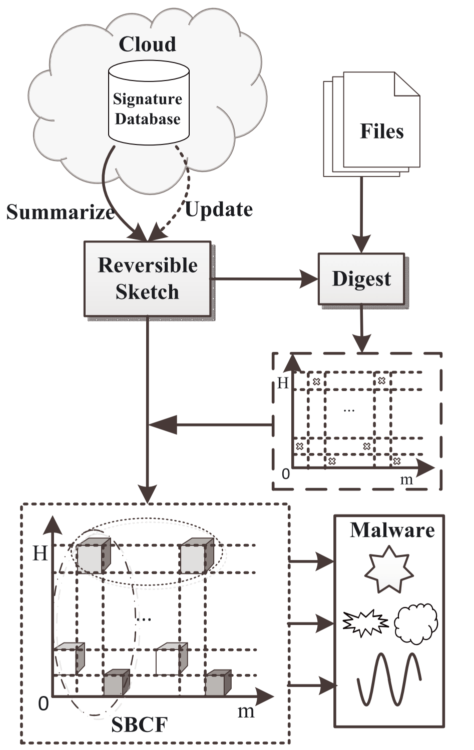

Figure 2. The system architecture of CloudEyes. SBCF, suspicious bucket cross-filtering.

We illustrate the system architecture of CloudEyes in Figure 2. The cloud server runs the signature database, summarizes the signatures into the reversible sketch. Meanwhile, the cloud generates a digest of the sketch which represents the existence of signatures. The digest is stored in the client when CloudEyes is firstly installed. The cloud updates the signature database and sketch periodically and sends the locations in the sketch where the changes take place to the client. The detail operations will be described in Section 4.2. As for file scanning, the client, rather than sends the whole file content to the cloud, first initializes the file content into the segments by the similarity method with the signatures (described in Section 4.1), then sifts out the unmatched segments with the latest digest. The matched ones are suspicious and need to be confirmed. We design the suspicious bucket cross-filtering (SBCF) mechanism for the cloud to locate the malicious file segments according to the sketch coordinates of suspicious segments sent from the client. The results which consist of modular hashing of malicious signature fragments are sent back to the client as a confirmed report according to which the client takes corresponding security measures. Figure 3 explicates the workflow during the communication between clients and cloud server in CloudEyes.

A list of frequently used notations is maintained in Table I.

4. DESIGN

In this section, we give a detail description about the signature-based detection mechanism via reversible sketch structure in CloudEyes.

4.1. Signature initialization

Let DB be the signature database managed in the cloud. Considering signatures do not have uniform length generally, we set a sliding window with size of w to scan the signatures in DB. For an arbitrary signature S of length l, there will be a set of segments of length $w\text{-}byte$ after initial scanning, namely, $S \rightarrow \{S_1, S_2, \ldots, S_{l-w+1}\}$. Moreover, we take account of the wildcards in specific signatures to map down multiple versions of a malware that originated from the same source.

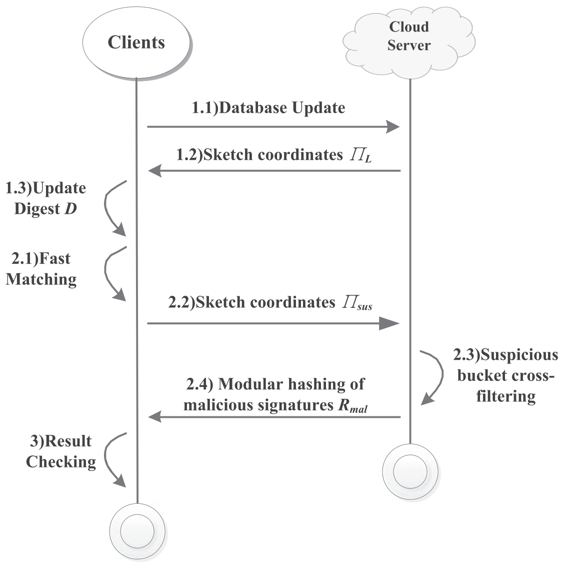
Clients
Cloud Server
1.1)Database Update
1.2)Sketch coordinates Π_L
1.3)Update Digest D
2.1)Fast Matching
2.2)Sketch coordinates Π_{sus}
2.3)Suspicious bucket cross-filtering
2.4) Modular hashing of malicious signatures R_{mal}
3)Result Checking

Figure 3. Workflow between clients and server in CloudEyes.

Table I. Frequently used notations.

H	number of hash tables
$m = k$	number of buckets per hash table
K	number of hash functions per hash table adopts
w	size of sliding window in signature initialization
X	a signature fragment after initialization with the length of w
q	number of words X is broken into, $X = \{X_1, X_2, \cdots, X_q\}$
N_S	number of signatures in database
$\overline{l}$	average length of the signatures
$L_i^j(X)$	sketch coordinate located by hash function result $h_i^j(X), i \in [H], j \in [K]$
$RB\left[L_i^j\right](X)$	the corresponding reversible bucket of $L_i^j(X)$
$D\left[L_i^j\right](X)$	the corresponding digest of $L_i^j(X)$, value is 0 or 1
$M(X)$	the modular hashing of X, $M(X) = \{mh_1(X_1)mh_2(X_2)...mh_q(X_q)\}$
FP_h, FP_{mh}, FP_c	three types of false positive: hashing, modular hashing and collision respectively

In a way, the initialization can be effective in handling polymorphic malware caused by wildcards [13]. However, it is still impractical to deal with all possibilities. In CloudEyes, the signatures with wildcard are roughly divided into two portions.

4.1.1. Fixed-size wildcard. It denotes the wildcards which contains numbered probabilities. For example, $''?''$ matches any byte, $''a|b|c''$ matches $''a''$ or $''b''$ or $''c''$. We adapt modulo (q) function in the wildcard signature initialization, which maps each string byte to a class between 0 to $q - 1$ (q is a random number smaller than 256),to support wildcard matching [27]. Therefore the matching space size is restricted because matching any value between the range of $[0, q - 1]$, instead of all possible values between 0 to 255, means successful hit. For instance, suppose a signature $''abcd?efgh''$ is initialized with $q = 4$ and $w = 9$. The initialization is processed by constructing four segments:$''abcd0efgh''$,$''abcd1efgh''$,$''abcd2efgh''$ and $''abcd3efgh''$. Similarly, $''abcd(x|y|z)efgh''$ is classified into three substrings: $''abcd0efgh''$,$''abcd1efgh''$ and $''abcd2efgh''$ because character x would be mapped to class 0 as $ASCII(x)\ mod\ q = 0$.

4.1.2. Variable-size wildcard. It denotes the wildcards with variable size, such as, $''*''$ matches any number of bytes, $''\{n\}''$ matches n bytes. Considering the large amount of probabilities lead to serious performance slowdown, we ignore these wilcards and initialize the rest part of signature. For instance, a signature $''abcdef * ghijkl''$ or $''abcdef\{200\}ghijkl''$ is initialized with $w = 6$, the corresponding substrings are $''abcdef''$ and $''ghijkl''$.

Additionally, if a signature does not contain a fixed fragment at least as long as the window size, the signature cannot be initialized. Small value of w cannot provide enough amount of unique fragments which raise the rate of collision to an unacceptable level during mapping. Alternatively, if the value is too large, there is not enough granularity to answer queries for smaller file fragments in detection. Study result of ClamAV's signature set for the 16-byte window size shows that the short-signature proportion is about 0.15% after initialization. This infrequence does not significantly reduce performance. For convenience, below we use X to represent a signature fragment after initialization.

4.2. Reversible sketch structure

4.2.1. Basic design. In our design, each element of hash table consists of a container called $bucket(RB)$ which stores the information of signature and a bit called $digest(D)$ which stands for the bucket is empty or not, with the value 0 or 1 respectively. Let $h_i^j, i \in [H], j \in [K]$ be $H * K$ functions randomly chosen from a class of 2-universal hash functions, each hash table adopts K independent functions respectively. Assume an arbitrary signature X with length of w-byte, that is $X = \{x_1, x_2, \ldots, x_w\}$. As we adopt modulo (q) function to deal with the signature contain fixed-size wildcards initially, each byte of X (or file content) needs to do the same modulo arithmetic to avoid false negative rate in detection, although it will bring slight false positive rate which is discussed in Section 5. Hence, the hashing result of X is $h_i^j(X) = h_i^j\,((x_1\ mod\ q), (x_2\ mod\ q), \ldots, (x_w\ mod\ q))$. Then, we can use $L(X) = \bigcup_{i=1}^{H} \bigcup_{j=1}^{K} L_i^j(X)$ which consists of $L_i^j(X) = \left(i, h_i^j(X)\right) (1 \leqslant i \leqslant H)$ to be the sketch coordinates of X, that means each signature fragment X has $H * K$ sketch coordinates. When summarizing X into RS, all the $L_i^j(X)$ in $L(X)$ are utilized to locate the corresponding reversible buckets $RB\left[L_i^j\right](X)$ and digests $D\left[L_i^j\right](X)$.

There are three operations related with RS:

(i) *Insert* $(X, L(X))$*:* Initially, RB contains no element and all the digests value is 0. For X which has not been mapped, $L(X)$ decides which buckets it belongs to. Then the sketch is updated as follows.

$$\begin{aligned} &\text{for each } L_i^j(X) \in L(X), i \in [H], j \in [K] \\ &RB\left[L_i^j\right](X) \leftarrow RB\left[L_i^j\right](X) \bigcup \{M(X)\} \\ &D\left[L_i^j\right](X) \leftarrow 1 \end{aligned}$$

Figure 4 illustrates the state of reversible sketch structure with $K = 2$ after inserting X_1 and X_2. The buckets labeled by sketch coordinates mean each contains at least one $M(X)$ and the rest stand for empties. And $M(X)$ in the operation denotes the modular hashing of X which is designed in storage mechanism (discussed in Section 4.2.2) for reducing the memory consumption.

(ii) *Delete* $(X, L(X))$*:* For the signature X that is proved to be incorrect or reduplicate for malware description, the servers call delete operations to get rid of X from the sketch with following steps:

$$\begin{aligned} &\text{for each } L_i^j(X) \in L(X), i \in [H], j \in [K] \\ &RB\left[L_i^j\right](X) \leftarrow RB\left[L_i^j\right](X) - \{M(X)\} \\ &D\left[L_i^j\right](X) \leftarrow 0, if\ RB\left[L_i^j\right](X) = \emptyset \end{aligned}$$

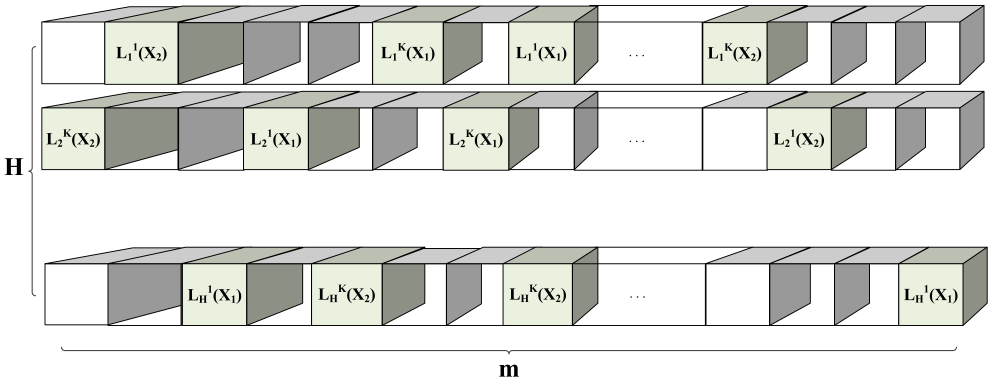

Figure 4. Reversible Sketch Structure after insertions.

(iii) *Update (Σ_X, Π_L,OP):* The cloud needs to periodically update the signature database with the increment of signature quantity. $\Sigma_X = \{X_1, X_2, \ldots, X_n\}$ is the set of signature fragments need to be updated, $\Pi_L = \{L(X_1), L(X_2), \ldots, L(X_n)\}$ is the set of sketch coordinates to locate the fragments and OP is the set of operations (1 or 0, stands for *Insert* or *Delete* respectively) corresponding to each signature. After the *Update* operation, the RB and D complete the similar changes with the two operations described earlier Additionally, the clients need to update their signature digests before file scanning. After receiving the update requests from clients, the cloud will send the Π_L and OP back. Hence, the clients and cloud keep synchronous in this way.

4.2.2. Storage mechanism. After summarizing the signature database into the reversible sketch, fundamental scanning about the database can be approximately answered very quickly according to the previous work [28, 29]. Generally speaking, the contents of signature should be stored in the structure in order to achieve the scanning veracity without the accurate scanning process like SplitScreen. Considering the amount of signatures is huge and growing, it is not scalable to store all the signature segments into the RS. Likewise, it is not applicable to assign each signature segment a unique number and store them, because the number of signatures is dynamic. To balance memory consumption and searching speed in the implementation, we design the storage mechanism based on modular hashing for signatures which accompanies with the RS operations.

In the stage of signature initialization, we utilize sliding window with size of w bytes and modulo (q) function. Hence, the basic element to be stored is the signature segment with $8w$ bits. Instead of directly hashing the entire segment in $[2^{8w}]$, we adopt modular hashing which divide the segment into q words, each word of size $8w/q$ bits. Then each word is hashed respectively by different hash functions which map from space $[2^{8w/q}]$ to $[2^{8w/q^2}]$. Figure5 illustrates the process of modular hashing. The 16-byte segment is divided into four words, each of 4 bytes, which are mapped by four independent hash functions from space $[2^{32}]$ to $[2^8]$. The hashing results of each word are concatenated to compose the final hashing result.

For a signature segment X, $M(X)$ denotes its modular hashing result. We adopt q independent hash functions $mh_1, ..., mh_q$ for every hash table, so $M(X) = \{mh_1(X_1)mh_2(X_2)...mh_q(X_q)\}$ is stored into the corresponding buckets in RS and all these buckets are arranged in the form of red black tree to achieve fast and dynamic operations. Suppose each hash function needs constant time to hash a value, modular hashing will slightly increase the operations discussed in the section above from $O(H * K)$ to $O(H * K + q)$ because the calculation of $M(X)$ simply executes once before the storage, while the memory consumption will be decreased by q times. Moreover, modular hashing permits the efficient execution of suspicious buckets cross-filtering and avoids the direct exposure of signatures during the communications between cloud server and clients (more details in Section 4.3.2)

More theoretical analysis about the accuracy of reversible sketch structure is discussed in Section 5 later and details about the performances are illuminated in Section 6.

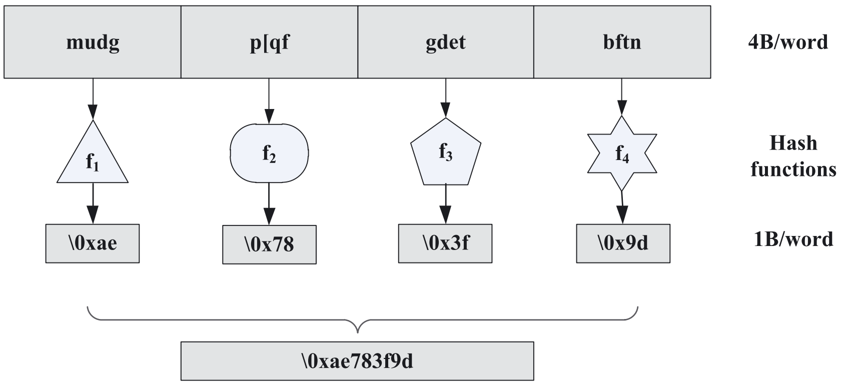

Figure 5. Modular hashing of signature segment ($w = 16, q = 4$).

4.3. Matching mechanism

The design of matching mechanism in CloudEyes is inspired by two purposes we desired: (1) taking the demands of both security vendors and clients into account and (2) ensuring high performance in the matching of file content. Hence, we divide the process of matching into two steps, fast scanning and suspicious bucket cross-filtering, for the client and cloud, respectively. Detail descriptions are listed below:

4.3.1. Fast scanning. In the CloudEyes system, the reversible sketch structure, which contains the reversible buckets and digest, is designed to store the summarization of signature and service for matching. The digest is the crux of fast scanning process which is stored in the client when the system is firstly installed. The files need to be initialized with w and q before scanning because of their diverse types and sizes, that is the file content should be incised into regular fragments and then do the modulo arithmetic like the initialization of signatures. Let F be the set of file fragments after initialization, the purpose of fast scanning is picking out the suspicious set of file fragments F_{sus} and the corresponding set of sketch coordinates Π_{sus} with the digest D.

Algorithm 1 Fast Scanning

```
Input: file fragments set $F$, digest $D$
Output: suspicious fragment set $F_{sus}$ and sketch coordinate set $\Pi_{sus}$
 1: $F_{sus}, \Pi_{sus} = \emptyset$;
 2: while each $f \in F$ do
 3:     calculate $M(f), L(f)$;
 4:     $clear = 0$;
 5:     for $i = 1$ to $H$ do
 6:         if $\sum_{j=1}^{H} D[L_i^j](f) < H$ then
 7:             $clear = 1$, break; //$f$ is not suspicious
 8:         end if
 9:     end for
10:     if $clear = 0$ then
11:         insert $M(f)$ into $F_{sus}$ and $L(f)$ into $\Pi_{sus}$;
12:     end if
13: end while
```

For each fragment in F, we calculate its sketch coordinates in the digest and check the corresponding value to estimate its existence. Only successful matching in all $H * K$ sketch coordinates make the fragment suspicious, the others are normal because the hash functions bring no false negative during signature summarization. That is to say, the file fragment f is checked to be suspicious if and only if all the values of $D\left[L_i^j(f)\right], i \in [H], j \in [K]$ are equal to 1. After all the fragments have been checked, the suspicious fragment set F_{sus} which consists of the modular hashing result of file fragments, and corresponding sketch coordinate set Π_{sus} are generated. Algorithm 1 presents details of fast scanning mechanism. This process is easy to be applied in the client because of its lightweight and can largely reduce the number of file fragments to be further confirmed. Considering the privacy protection of client, we only send the sketch coordinates of suspicious fragments to the cloud after fast scanning, which can cut down the communication consumption for the client simultaneously.

4.3.2. Suspicious bucket cross-filtering. This process aims at finding the set of culprit signature fragments according to the result of fast scanning. The basic idea is checking every reversible bucket according each sketch coordinate sent from the client to find the signature fragment which exists in all the H hash tables.

As we describe earlier, different types of signature need to be initialized into regular fragments. Let N_S be the total number of signatures in the DB (including the signatures with wildcards after initialized), and $\bar{l}$ be the average length of the signatures, w is the size of sliding window, m is the size of per hash table, and K is the number of hash functions each hash table adopts. So the number of fragments after initialization is $\left(\bar{l} - w + 1\right) \cdot N_S$, and each bucket averagely contains $t = \left(\bar{l} - w + 1\right) \cdot N_S \cdot K/m$ modular hashing results of signature fragments. The reversible buckets in RS corresponding with the set of Π_{sus} are treated as suspicious ones. For each f whose $M(f) \in F_{sus}$, there are K suspicious buckets in each of H hash tables.

The intuitionistic heuristic to find the target signature fragments is taking the intersections of all these buckets, but it is noteworthy that the values of K will make the filtering process different. For $K = 1$, each f relates to one suspicious bucket in each hash table. One possible way to achieve our goal is to take the union of the possible fragments of all suspicious buckets for each hash table and then take the intersections of these union. But it can lead to an enormous amount of fragments output that do not match and needless computation which called Reverse Sketch Problem [26]. We solve this instance in our previous work RScam [19] by building a filtering buffer to count the appearances of signature fragments in suspicious buckets. For $K \geqslant 2$, there are K buckets which contain the duplicates of arbitrary $M(f)$ which means the union operations do not work. Hence, the processing range should be shrink from all suspicious buckets in each hash table to the ones related to each f and the suspicious bucket filtering should be performed in cross way (row and column).

Algorithm 2 shows the process of suspicious bucket cross-filtering. T_{row} and T_{column} are the modular hashing filtering buffers based on red black tree structure in the row and column orientation, respectively. The affirmative precondition is that the suspicious buckets corresponding to the sketch coordinates Π_{sus} are not empty. In the row orientation, we want to pick out the signature fragments exist in all the K suspicious buckets. Considering the economization of memory and computation consumption, we insert the first word of fragments' modular hashing results contained in the targeted buckets into T_{row} and insert the ones whose count is K into T_{column}, which means the signature fragments appear in all the K suspicious buckets. After the filtering in all row orientations has finished, we do the similar filtering in the column orientation to screen out the H-count words and insert the corresponding $M(X)$ of signature fragments into the result set R_{mal} which can be utilized to claim the malice of file fragments in the client.

The cross-filtering scheme is running in the cloud server which possesses sufficient computation and memory power, but we still need to carefully choose the parameters to make our matching efficient. The filtering buffers we adopt are based on the red black tree structure which achieve the insertion and searching in $O(\log N)$. For arbitrary $L(f)$, there are $K * H$ corresponding buckets each of which contains t modular hashing results, so the total insertion and searching operations can be implemented roughly in $2K \cdot H \cdot O(\log t)$ (the operations of T_{column} can be ignored because each $L(f)$ corresponds to one word in each hash table theoretically). The H and K are assigned to be small constant. Hence, the total time complexity of suspicious bucket cross-filtering is equivalent to $O(N \log N)$. The analysis of memory cost is discussed in Section 6.2.

After the suspicious bucket cross-filtering, cloud server sends the result R_{mal} back to the client. The culprit signature fragments and short signatures should be compared with the suspicious file

Algorithm 2 Suspicious Bucket Cross-Filtering

```
Input: Sketch coordinates $\Pi_{sus}$, reversible buckets $RB$
Output: Set of malicious signature fragments $R_{mal}$
 1: $R_{mal} = \emptyset$;
 2: while each $L(f) \in \Pi_{sus}$ do
 3:   $T_{column} = \emptyset$;
 4:   for $i = 1$ to $H$ do
 5:     $T_{row} = \emptyset$;
 6:     for $j = 1$ to $K$ do
 7:       insert $mh_1(X_1) \in RB[L_i^j(f)]$ into $T_{row}$;
 8:     end for
 9:     for each $mh_1(X_1) \in T_{row}$ do
10:       if count$(mh_1(X_1)) = K$ then
11:         insert $mh_1(X_1)$ into $T_{column}$; // filtering in row orientation
12:       end if
13:     end for
14:   end for
15:   for each $mh_1(X_1) \in T_{column}$ do
16:     if count$(mh_1(X_1)) = H$ then
17:       insert the corresponding $M(X)$ into $R_{mal}$; // filtering in column orientation
18:     end if
19:   end for
20: end while
```

fragments in modular hashing form to make sure the veracity of matching mechanism. The modular hashing avoids the direct exposure of signature fragments during the communication with server and client to protect the privacy and profit of server. Moreover, the security vendors can choose some classical encryption algorithms to further ensure the secure communication which is beyond the scope of this work. The client will take some security measures, such as deletion or isolation, with the infected files after validate the matching results.

5. DISCUSSION

In this section, we discuss the accuracy of the reversible sketch structure which is measured based on the false negative and false positive rates generally. A false negative occurs when a fragment summarized into the RS earlier is asserted as clean when matching. While the false positive occurs when a query fragment not summarized into the RS is incorrectly stated as present. There are two types of false positives in CloudEyes. The first one is caused by the hash functions employed in the RS, which is called *hashing false positive*. Secondly, the modulo arithmetic adopted in the initialization brings the possibility of collision between two different fragments and modular hashing of signature fragments adopted in the storage mechanism. Here, we call it *fragment false positive*. In what follows we will conduct the theoretical and statistical analysis of these measurements.

5.1. Fasle negative

The false negative is caused by the initialization based on fixed-size slide window, rather than the hash functions. For example, suppose the signature *″abcdefg″* has been summarized into RS with window size of 6, which means two signature fragments are constructed and mapped into the RS: *″abcdef″* and *″bcdefg″*. Now, if we scan the file content *″bcdef″*, it will respond that the file was clean which is incorrect. It is remarkable that false negative in CloudEyes would occur only for the short file content whose length is less than w bytes. So it greatly depends on the length of the scanning content. However, this situation seldom takes place and is hard to be evaluated in prevalent security detection because sizes of files to be scanned are always larger than w bytes which we set in the evaluation. Alternatively, the calculation of false negative can be more comprehensible according to the number of short signatures produced by the initialization in the cloud server. We can adjust the value of w to minimize the false negatives of CloudEyes, while w also plays an important role in the false positives discussed later. Hence, it is worthwhile to give our careful consideration about the tradeoff (more details in Section 6.5).

5.2. *Hashing false positive*

The hash functions we use earlier are 2-universal which make the hash results are nearly randomized. Hence, the principle and accuracy of summarization is similar with the Bloom Filter. This type of false positive comes from the hash collisions which may lead to the conclusion that a specific fragment is suspicious when it is not. Alternatively, the false negative will never exist. We learn about the probability of false positive in a bloom filter can be calculated with following relation:

$$FP = \left(1 - \left(1 - \frac{1}{m}\right)^{kN}\right)^{k} \tag{1}$$

where m is the length of bloom filter, k is the number of used hash functions, and N is the amount of inserted elements. We can easily conduct the hashing false positive of a hash table in RS. As described earlier, each hash table uses K hash functions and $\left(\bar{l} - w + 1\right) \cdot N_S$ fragments are inserted into it. So the false positive of each hash table is

$$\alpha = \left(1 - \left(1 - \frac{1}{m}\right)^{(\bar{l}-w+1)\cdot N_S \cdot K}\right)^{K} \tag{2}$$

There are H hash tables built in RS which makes the hashing false positive reasonable if and only if collisions exist in all of the H ones. According to the relation (2), let FP_h be the hashing false positive of RS that is

$$FP_h = \left(1 - \left(1 - \frac{1}{m}\right)^{(\bar{l}-w+1)\cdot N_S \cdot K}\right)^{H \cdot K} \tag{3}$$

5.3. *Fragment false positive*

As we described in Section 4, the CloudEyes system adopts the modulo arithmetic to deal with the wildcards in specific signatures and cut down the storage consumption. However, this will introduce collisions between different fragments. Specifically, there are two distinct scenarios that lead to fragment collisions in the wildcards case which is discussed later, and the third one occurs in the storage mechanism which is independent of the others.

5.3.1. *Collision before summarization.*

This scenario occurs between two unsummarized fragments, that is, the hashing value of them is uniform. Suppose that S and S' are two different strings (signatures or files) with same length of $\bar{l}$. Assume that $S = s_1 s_2 \ldots s_{\bar{l}}$ and $S' = s'_1 s'_2 \ldots s'_{\bar{l}}$, and the number of classes by q, then the collision happens if each byte of string belongs to same class after modulo. Let F_1 be the false positive before summarization, which is calculated by

$$F_1 = \left(\frac{\lceil \frac{256}{q} \rceil}{256}\right)^{\bar{l}} \leqslant \left(\frac{1}{q} + \frac{1}{256}\right)^{\bar{l}} \tag{4}$$

5.3.2. Collision after summarization. This scenario occurs when the unsummarized file content is matched which is incorrect. Suppose that $S = s_1 s_2 \dots s_{\bar{l}}$ is initialized with the window length of w. As noted earlier, the number of w-byte fragments after initialization is $\left(\bar{l} - w + 1\right)$. The collision happens when all these fragments are wrongly resulted in suspicion. Let F_2 be the false positive after summarization, we can conclude the relation below according relation (4)

$$F_2 = \left(\frac{\lceil \frac{256}{q} \rceil}{256}\right)^{w \cdot (\bar{l} - w + 1)} \leqslant \left(\frac{1}{q} + \frac{1}{256}\right)^{w \cdot (\bar{l} - w + 1)} \tag{5}$$

Consequently, the probability of collisions are the sum of F_1 and F_2. However, we should negate the situation that all the bytes in the string are really equal. Moreover, the collision is directly related to the number of signatures summarized into the RS. Let FP_c be the collision false positive rate, then we have

$$\begin{aligned} FP_c &= \left[F_1 + F_2 - \left(\frac{1}{256}\right)^{\bar{l}}\right] \cdot N_S \\ &\leqslant \left[\left(\frac{1}{q} + \frac{1}{256}\right)^{\bar{l}} + \left(\frac{1}{q} + \frac{1}{256}\right)^{w \cdot (\bar{l} - w + 1)} - \left(\frac{1}{256}\right)^{\bar{l}}\right] \cdot N_S \end{aligned} \tag{6}$$

5.3.3. Modular hashing false positive. We adopt modular hashing in the storage mechanism to cut down the memory consumption which brings the false positive simultaneously. The q functions are randomly chosen from a class of 2-universal hash functions which ensure each part of signature segments are mapped independently and uniformly. Let FP_{mh} be the modular hashing false positive of any signature fragment X with the size of w-byte, then we have

$$FP_{mh} = \left(\frac{1}{2^{8w/q^2}}\right)^q \cdot \left(\bar{l} - w + 1\right) \cdot N_S = \left(\bar{l} - w + 1\right) \cdot N_S \cdot 2^{-8w/q} \tag{7}$$

In conclusion, the total false positive of CloudEyes FP can be computed by the relations (3), (6) and (7) as follows:

$$FP = FP_{mh} \cdot FP_h + FP_c \tag{8}$$

In Figure 6(a), the hashing false positive, denoted by FP_h, is illustrated in the left vertical axis with blue solid line; the modular hashing false positive, denoted by FP_{mh}, is illustrated in the right vertical axis with red dashed line. In Figure 6(b), the collision false positive, denoted by FP_c, is depicted by black dashed line. As observed, FP_h is much larger than FP_c with different number of signatures after initialization. And FP_{mh} is 80 times smaller than FP_h averagely. So FP_c is negligible compared with $FP_{mh} \cdot FP_h$, and the false positive of CloudEyes primarily lies on FP_h. It is reasonable that FP_h grows close to 1 when the number of signatures grows close to the size of hash table because empty reversible buckets get rare.

6. PERFORMANCE EVALUATION

In this section, we first introduce the experimental setup in detail, then we evaluate the performance of the CloudEyes system and make some comparison with the ClamAV and SplitScreen with different measurements.

6.1. Evaluation setup

We have implemented CloudEyes based on the file and signature identification model of ClamAV with approximately 7K lines of C/C++ code which consist of 4.5K for cloud server and the rest for client. CloudEyes clients and cloud server are connected with each other via Transmission Control Protocol sockets. And, we adopt the current version of ClamAV (0.98.5) and implement SplitScreen based on it for latter comparisons.

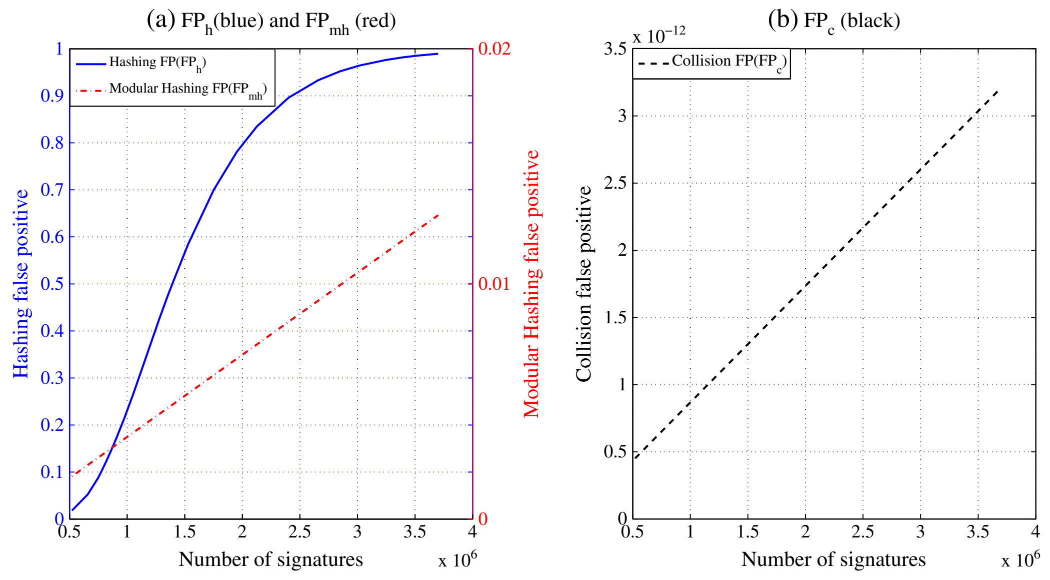

Figure 6. Three types of false positive in CloudEyes with $m = 2^{24}, w = 16, \bar{l} = 30, H = 4, K = 2, q = 4$ and different number of signatures between 460,000 to 3,700,000. (a) is hashing false positive FP_h (depicted by blue line) and modular hashing false positive FP_{mh} (red line); (b) is collision false positive FP_c (black line).

The signature databases which originate from the ClamAV open source platform contain two types of signatures: whole file or segment MD5 signatures and regular expression signatures. We employ ten versions from Nov. 2008 to Nov. 2014, which the number of signatures ranges from 460,000 to 3,700,000. About 98% of all signatures are MD5 signatures with uniform size of 16 bytes each in the latest database which means one MD5 signature can be treated as a signature segment directly. Thus, we fix the value of w to be 16 while dealing with MD5 signature initialization and scanning. If unspecified, we use $m = 2^{24}, q = 4, H = 2, K = 2, \bar{l} = 20$ for the RS in our experiment and $w = 16$ for regular expression signatures as well. We implement the evaluation with the latest database (main v.55 and daily v.19688) and show the average results over 20 runs.

Our total 36GB suspicious data set consists of about 240,000 unique samples named by MD5 hash, which are captured by specific IDS from the campus network. The clean files come from the install of common applications. And experiments are performed on a CentOS 5.6 virtual cloud server (8 cores, 32-GB memory, and 2.53 GHz) and a common open research network emulator based on OpenVZ which provides different types of virtual machines and distributed network. Figure 7 depicts the topological architecture of our experiments.

6.2. *Memory analysis*

As described earlier, we adopt the reversible sketch structure in the cloud server. Each bucket averagely contains $t = \left(\bar{l} - w + 1\right) \cdot N_S \cdot K/m$ signature segments, so the entire memory cost is at least $(w \cdot t \cdot m \cdot H)/q$ bytes theoretically. We utilize the dynamic red black tree structure to store these segments and prune the reduplicate ones after initialization. This process takes up a period of time, but we do not count it in the performance of CloudEyes because it performs only once at the starting of evaluation.

We first practically analyse the numbers of signature segments and filled buckets in RS after initialization with different number of signatures. Table II lists these numbers and the average memory cost of the cloud server. The number of signature item listed in the table indicates how many signatures are contained in different versions of ClamAV database. For example, 460K means the database (main v.49 and daily v.8683) contains 460 thousands signatures, 3.7M means the database (main v.55 and daily v.19688) contains 3.7 million signatures, et cetera. The item of Signature Seg-

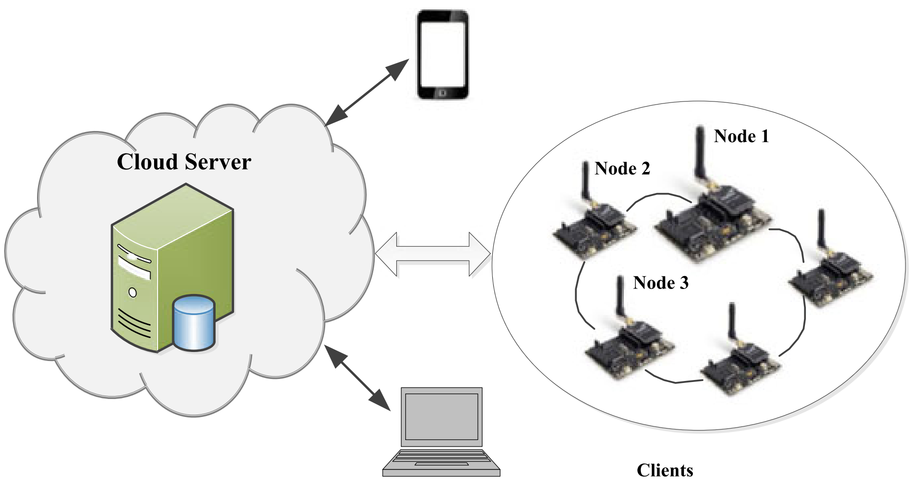

Figure 7. The experimental topological architecture.

Table II. Memory analysis of cloud server.

	The number of signature						
	460K	530K	860K	1M	2M	3M	3.7M
Signature segments	7,642,123	7,717,276	8,355,560	8,774,970	9,567,772	10,733,552	11,917,850
RS Buckets	4,862,216	5,031,482	5,565,858	6,427,696	7,737,428	8,624,520	9,798,166
Cloud Server(MB)	102	149	195	262	328	417	488

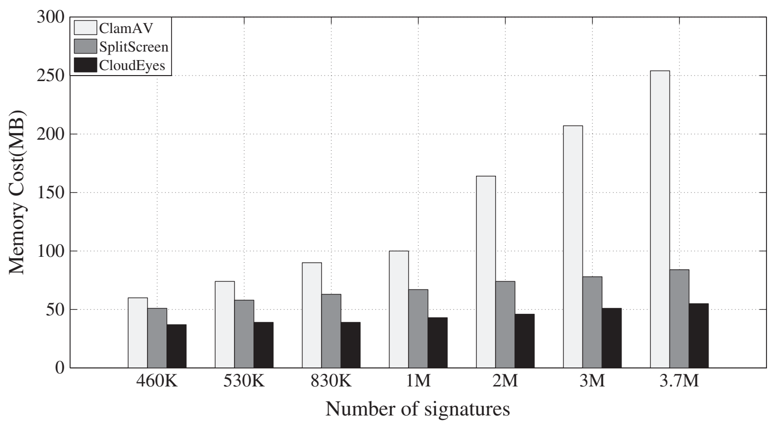

Figure 8. Memory cost of ClamAV, the client of SplitScreen and CloudEyes with different numbers of signatures.

ments in the table means the sum of original segments after signature initialization, and RS Buckets item includes all the buckets which contain at least one segment in H hash tables. As observed, the memory cost of cloud server in CloudEyes mounts up with the growth of signatures. However, it is acceptable for security vendors. The commercial cloud products are abundant to achieve the storage in memory and ensure high accessing speed.

Figure 8 lists the average memory cost of the client with various number of signatures after we adjust from different versions when scanning 900 suspicious samples (total 600MB). Unlike the cloud server, memory cost of client does not grow with the number of signatures. We compare the memory cost of SplitScreen client and ClamAV with same environments which are also illustrated in Figure 8. The numbers indicate that our client appropriator the least memory, which means

CloudEyes is more applicable than SpliltScreen because the latter calls the accurate scanning of ClamAV after its first scanning.

6.3. Time analysis

In this section, we evaluate the performance of our system from several perspectives. First, we test the scanning time cost of CloudEyes in the virtual machine as a resource-constrained client with 350MB memory, 256KB L2 cache, and 1GHz CPU, and the bandwidth between the cloud and client is 1MB/s. The testing data are the samples randomly chosen from our data set. The average size of each sample is 2MB. Meanwhile, we make comparisons with the system of ClamAV and SplitScreen in the same environment. Figure 9 shows the details of the time cost. We implement this with 1MB signature database (main v.54 and daily v.13810) because ClamAV exhausts the system memory when running with larger signature databases. As observed, CloudEyes outperform the others with lower time consumption and smoother increment. We can conclude that small cache volume slows down the detecting speed of SplitScreen distinctly. In some condition, SplitScreen even runs slower than ClamAV.

Moreover, we are concerned about the composing of the time cost illustrated in Figure 10 which reveals the effect of our matching mechanism. The mean percentage of accurate scanning of SplitScreen is 74.3% while that of CloudEyes is 19.7%. The fast scanning takes account of all the file fragments which matched in the digest to avoid the accurate scanning of whole file content,

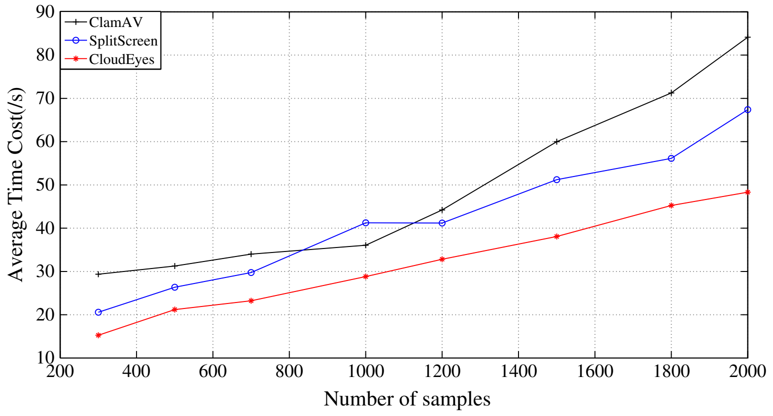

Figure 9. Time performance of ClamAV, SplitScreen and CloudEyes using different number of samples.

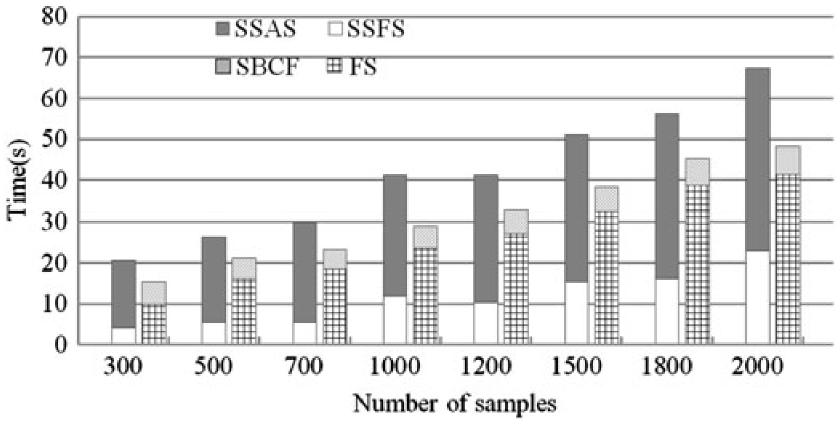

Figure 10. The composing of time cost of CloudEyes and SplitScreen. SSAS and SSFS stand for the accurate and first scanning of SplitScreen, respectively. SBCF and FS stand for suspicious bucket cross-filtering and fast scanning of CloudEyes.

while the fast scanning of SplitScreen only reserve the first matched file fragment to label the file to be accurately scanned. In this way, we cut down a mass of computation and time. Hence, we can confirm that the matching mechanism based on the reversible sketch structure can largely improve time performance.

Besides, we try to analyse the time costs of three systems from the hardware execution perspective. We hypothesize that the primary time consumption of these systems are cache misses which consist of instruction fetches and loads that miss the different level caches. Figure 11 illustrates the total number of cache misses for ClamAV, SplitScreen, and CloudEyes as the number of signatures increases. We achieve the statistic and comparison after servers of SplitScreen and CloudEyes are steadily running. The trendlines of increment which tally with the time performance confirm our hypothesis. The lower number of cache misses indicates the less time consumption and this disparity grows as the number of signatures increases.

Finally, we test the throughput performance of CloudEyes with three different resource-constrained nodes assigned by the network emulator: Node 1 (512KB L2 cache, 1GHz CPU, 1GB RAM); Node 2 (256KB L2 cache, 1GHz CPU, 768MB RAM) and Node 3 (256KB L2 cache, 512MHz CPU, 768MB RAM). Comparisons with SplitScreen and ClamAV are made using latest signature database and 210 MB suspicious data samples. Figure 12 shows these results which indicate CloudEyes performs better than the others in three nodes. In the third node with lower frequency processor, the performance is affected because more time is needed in the fast scanning.

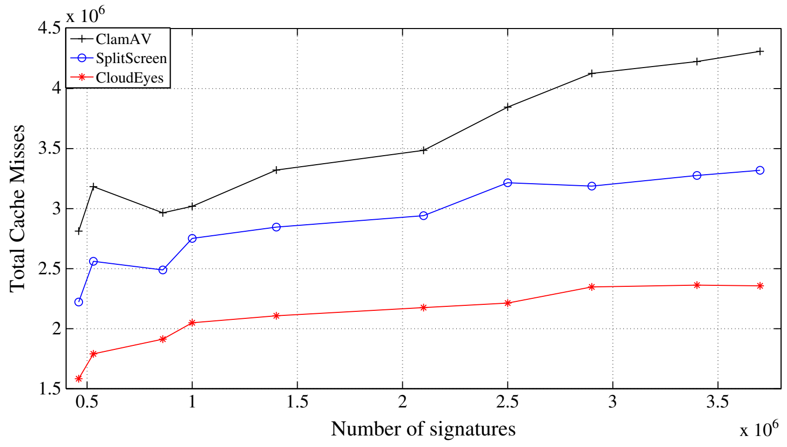

Figure 11. Total cache misses of ClamAV, SplitScreen and CloudEyes using different number of signatures.

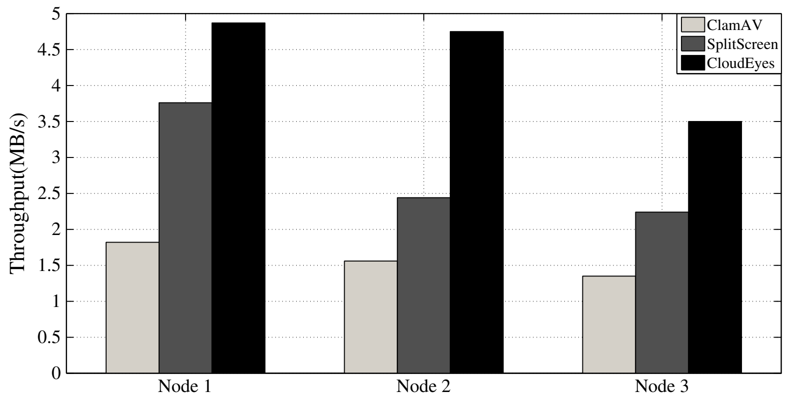

Figure 12. Throughput performance for three different resource-constrained nodes.

6.4. *Communication consumption analysis*

Another important inspiration of our design is data privacy protection with slight amount of communication consumption between the client and server. We achieve this through the communication mechanism labored earlier. The client sends the sketch coordinates of suspicious file fragments to the server, and the server send the short signatures and transformation of malicious signature segments back to the client.

Figure 13 illustrates the average communication consumption between the client and server with different number of signatures in CloudEyes and SplitScreen when scanning 2GB suspicious samples. The other experiment parameters are same with Section 6.2, besides the client and server are connected with Transmission Control Protocol protocol.We do not take ClamAV into consideration to make this analysis not so far-fetched as the infrastructure of universal ClamAV is designed to be host-based. As observed, the communication consumption in CloudEyes is averagely 12.6 times smaller than that in SplitScreen, and stand smooth with the growth of signatures. The communication bandwidth of CloudEyes during scanning is averagely 36.7 KB/S which is acceptable for the resource-constrained clients, such as mobile phones and pads.

6.5. *Practical accuracy*

We discuss the accuracy of the reversible sketch structure in Section 5 and conclude that it should be carefully balanced with false positive and false negative. Moreover, we give a practical test of the accuracy in detecting 5972 clean PE files (totally 1.42GB) with different window size under the latest signatures database. There is no need to repeat the evaluation in anterior versions of database because they are successive. Table III lists the details of the practical accuracy. The false positive of CloudEyes is calculated by the number of suspicious file fragments divided by the total number of file fragments. As mentioned earlier, for MD5 signatures, we fix the value of w to be 16, the other variable values are for regular expression signatures. The false negative is calculated by the number of short signatures divided by the total number of signatures. Small window size cannot provide enough possibilities for the large amount of signature fragments which caused high false positive.

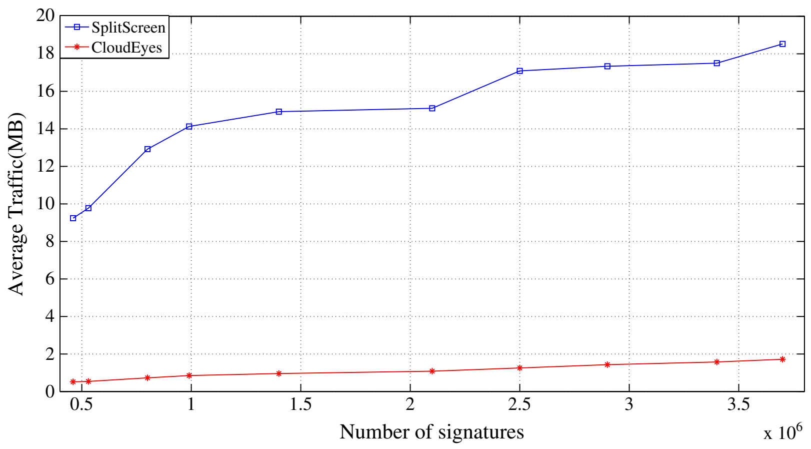

Figure 13. The communication consumption between the client and server with different number of signatures.

Table III. Practical accuracy of CloudEyes.

Window size	Fasle Positive (%)	Short Sigs	False Negative (%)
$w = 12$	7.861	3467	0.092
$w = 16$	5.726	5741	0.152
$w = 20$	3.380	7676	0.203
$w = 24$	2.371	10929	0.289

While large window size will produce many short signatures which bring high false negative and not be enough fine-grained. Hence, we can ensure the high accuracy of CloudEyes with considered window size and 20 seems to be the moderatest value.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed CloudEyes, a cloud-based anti-malware system which provides security service with high-performance detection and data privacy protection for resource-constrained devices. In this CloudEyes work, we designed suspicious bucket cross-filtering, a novel signature-based detection mechanism based on the reversible sketch structure which dramatically reduce the scanning range and provide retrospective and accurate orientations of malicious data fragments. And we implemented a lightweight scanning agent which utilizes the digest of signature fragments to sharply reduce accurate matching range. Meanwhile, we design the balanced interaction mechanism to protect the data privacy and reduce the communication consumption for both the clients and security vendors. Performance evaluation in suspicious campus networks and normal files shows that the system is able to achieve efficient malware detection and trusted protection of data privacy with slight traffic and acceptable memory requirement.

As part of our future work, we are planning to address several challenges. The detection performance can be ulteriorly improved by better algorithms. For example, private set intersection allows two parties to compute the intersection of private sets while revealing nothing more than the intersection itself. This property can be utilized to reinforce the suspicious bucket cross-filtering to protect the data privacy in the cloud. Moreover, some new effective methods applied in payload attribution to provide large data reduction rates and support efficient payload queries, such as Winnowing Block Shingling and Winnowing Multi-Hashing, can be utilized in the process of signature initialization to optimize the storage and matching performances.

ACKNOWLEDGEMENTS

This research is supported in part by the project of National Science Foundation of China (NSFC) under grant No.61303264, No.61202488; Guangxi cooperative innovation center of cloud Computing and big data No.YD16505; Distinguished young scientist promotion of National University of Defense Technology; the program of Changjiang Scholars and Innovative Research Team in University (No. IRT1012).

REFERENCES

1. Borgia E. The internet of things vision: Key features, applications and open issues. *Computer Communications* 2014; **54**(1):1–31.
2. Gubbi J, Buyya R, Marusic S, Palaniswami M. Internet of things (IoT): A vision, architectural elements and future directions. *Future Generation Computer Systems* 2013; **29**(7):1645–1660.
3. *Mcafee threats report: fourth quarter 2014*. Available from: http://www.mcafee.com/us/mcafee-labs.aspx.
4. Nguyen KT, Laurent M, Oualha N. Survey on secure communication protocols for the internet of things. *Ad Hoc Networks* 2015; **32**(2):17–31.
5. Cha SK, Moraru I, Jang J, Truelove J, Brumley D, Andersen DG. Splitscreen: enabling efficient, distributed malware detection. *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*, USENIX Association, Berkeley, CA, USA, 2010; 12–25.
6. Aho AV, Corasick MJ. An information-theoretic view of network aware malware attacks. *IEEE Transactions on Information Forensics and Security* 2009; **4**(3):530–541.
7. Chen Z, Ji. C. Efficient string matching: An aid to bibliographic search. *Communications of the ACM* 1975; **18**: 333–340.
8. Wu S, Manber U. A fast algorithm for multi-pattern searching. *Technical Report TR-94-17*, University of Arizona, 1994.
9. *Clamav*. Available from: http://www.clamav.net.
10. Vasiliadis G, Ioannidis S. Gravity: A massively parallel antivirus engine. *International Symposium on Recent Advances in Intrusion Detection.* RAID, Springer, Ottawa, Ontario, Canada, 2010; 79–96.
11. AV-comparative. On-demand detection of malicious software. *Technical Report AV-comparative*, AV-comparatives: Innsbruck, Austria, 2010. Available from: www.av-comparatives.org.

12. Yan W, Ansari N. Why anti-virus products slow down your machine? *Proceedings of IEEE International Conference on Computer Communications and Networks*, San Francisco, CA, USA, 2009; 1–6.
13. Erdogan O., Cao P. Hash-av: Fast virus signature scanning by cache-resident filters. *International Journal of Security and Networks* 2007; **2**:50–59.
14. Min B, Varadharajan V, Tupakula U, Hitchens M. Antivirus security: Naked during updates. *Software: Practice and Experience* 2014; **44**:1201–1222.
15. *Mcafee saas endpoint protection suite*. Available from: http://www.mcafee.com/us/products/saas-endpoint-protection-suite.aspx.
16. Jarabek Chris, Barrera David, Aycock John. Thinav: Truly lightweight mobile cloud-based anti-malware. *Proceedings of the 28th Annual Computer Security Application Conference (ACSAC'12)*, ACM: New York, NY, USA, 2012; 209–218.
17. Oberheide J, Cooke E, Jahanian F. Cloudav: N-version antivirus in the network cloud. *Proceedings of the 17th Conference on Security.* USENIX Association: Berkeley, CA, USA, 2008; 91–106.
18. Tang Y, Xiao B, Lu X. Signature Tree Generation for Polymorphic Worms. *IEEE Transactions on Computers* 2011; **60**(4):565–579.
19. Sun H, Wang X, Su J, Chen P. Rscam: Cloud-based anti-malware via reversible sketch. *Proceedings of International Conference on Security and Privacy in Communication Networks.* SecureComm'15: Dallas, Texas, USA, 2015.
20. Venugopal D, Hu G. Efficient signature based malware detection on mobile devices. *Mobile Information Systems* 2008; **4**(1):33–49.
21. Oberheide J, Veeraraghavan K, Cooke E, Flinn J, Jahanian F. Virtualized in-cloud security services for mobile devices. *Proceedings of the first workshop on virtualization in mobile computing*, 2008; 31–35.
22. Xu J, Yan J, He L, Su P, Feng D. Cloudsec: A cloud architecture for composing collaborative security services. *2nd IEEE International Conference on Cloud Computing Technology and Science*, 2010; 703–711.
23. Zonouz S, Houmansadr A, Berthier R, Borisov N, Sanders W. Secloud: A cloud-based comprehensive and lightweight security solution for smartphones. *Computers & Security* 2013; **37**:215–227.
24. Bloom BH. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, Vol. 13. ACM, New York, USA, 1970; 422–426.
25. *Data streams: Algorithms and application*, Vol. 1. Foundations and Trends in Theoretical Computer Science, 2005.
26. Schweller R, Li Z, Chen Y, Gao Y *et al.* Reversible sketches: Enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Transactions on Network* 2007; **15**(5):1059–1072.
27. Haghighat MH, Tavakoli M, Kharrazi M. Payload attribution via character dependent multi-bloom filters. *IEEE Transactions on Information Forensics and Security* 2013; **8**(5):705–716.
28. Wang F, Wang X, Su J, Xiao B. Vicsifter: A collaborative ddos detection system with lightweight victim identification. *Proceedings of the 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, 2012; 215–222.
29. He Ming, Gong Zhenghu, Chen Lin *et al.* Securing network coding against pollution attacks in p2p converged ubiquitous networks. *Peer-to-Peer Networking and Applications* 2015; **8**(4):642–650.