The Computer Journal Advance Access published April 15, 2009

doi:10.1093/comjnl/bxp026

A Digest and Pattern Matching-Based Intrusion Detection Engine

ZHONGQIANG CHEN[1,*], YUAN ZHANG[2], ZHONGRONG CHEN[3] AND ALEX DELIS[4]

[1]*Yahoo! Inc., Santa Clara, CA 95054, USA,*
[2]*Department of Mathematics, Florida State University, Tallahassee, FL 32306, USA,*
[3]*ProMetrics Inc., King of Prussia, PA 19406, USA,*
[4]*Department of Informatics and Telecommunications, University of Athens, Athens, 15784, Greece*
[*]*Corresponding author: zqchen@yahoo-inc.com*

Intrusion detection/prevention systems (IDSs/IPSs) heavily rely on signature databases and pattern matching (PM) techniques to identify network attacks. The engines of such systems often employ traditional PM algorithms to search for telltale patterns in network flows. The observations that real-world network traffic is largely legitimate and that telltales manifested by exploits rarely appear in network streams lead us to the proposal of *Fingerprinter*. This framework integrates fingerprinting and PM methods to rapidly distinguish well-behaved from malicious traffic. *Fingerprinter* produces concise digests or fingerprints for attack signatures during its programming phase. In its querying phase, the framework quickly identifies attack-free connections by transforming input traffic into its fingerprint space and matching its digest against those of attack signatures. If the legitimacy of a stream cannot be determined by fingerprints alone, our framework uses the Boyer–Moore algorithm to ascertain whether attack signatures appear in the stream. To reduce false matches, we resort to multiple fingerprinting techniques including *Bloom–Filter* and *Rabin–Fingerprint*. Experimentation with a prototype and a variety of traces has helped us establish that *Fingerprinter* significantly accelerates the attack detection process.

Keywords: pattern matching engine of IDSs/IPSs; multi-pattern matching algorithms; fingerprinting and digesting techniques; intrusion detection process

Received 5 October 2008; revised 14 March 2009
Handling editor: Alison Bentley and Florence Leroy

1. INTRODUCTION

Intrusion detection/prevention systems (IDSs/IPSs) are now ubiquitously deployed to shield intranets from attacks and protect the confidentiality, integrity and availability of involved computer systems [1]. IDSs/IPSs typically use pattern matching (PM) and anomaly analysis [2,3] to accomplish their objective. The PM techniques in IDSs/IPSs search for telltale patterns unique to known attacks while anomaly analysis seeks to detect network connection behavior that statistically deviates from normal activities. Such PM methods do help with the detection of known attacks and reduce the overall noise level of intrusions in the Internet although they appear rather ineffective when it comes to zero-day exploits [4]. Nevertheless, the ongoing strong exploitation of existing vulnerabilities and exposures is so prevalent in the globe today that the effective and well-managed use of PM methods is considered absolutely critical [5].

Security loopholes frequently emanate from common design and development mistakes in a multitude of applications including Web and Email [4,6]. Moreover, the easy access to 'kiddy scripts' and attack tools not only simplifies the exploitation process for known vulnerabilities [5], but also offers opportunities to intruders for synthesizing new strains of attacks. We should point out that the population of security flaws reported has consistently grown. The common vulnerabilities and exposures (CVEs) dictionary [6] listed 1565 loopholes in 1999 while it accrued 6832 and 6259 new species in 2006 and 2007, respectively, resulting in a total of 30648 collected CVEs in the past decade. In light of the rapid evolution of security loopholes and the ever-increasing complexity of applications, patching vulnerable computer systems promptly has become a real challenge [7]. It is thus vitally important to detect and prevent existing attacks from recurrence [3]. In this context, PM-based intrusion detection methods are the key mechanism

for contemporary IDSs/IPSs to detect and prevent the majority of intrusions in the Internet [8].

With attacks expressed as character sequences [2], PM-based IDSs/IPSs typically treat intrusion identification as a pattern search process. There is evidently a need to not only organize all such telltale patterns manifested by exploits in an effective signature or rule database within the IDS/IPS, but also to augment its content with specially crafted signatures for the ever-expanding attacks population. As the legitimacy of a network stream is determined by matching it against all telltales in the signature database, the performance of an IDS/IPS is dominated by the string matching operations conducted in the PM-engine (PME) [9]. There is clearly a trade-off between voluminous signature databases and attack coverage from one side and demanding CPU-intensive PM operations on the other. Also, the rapidly expanding network bandwidth and diversified applications further exacerbate the IDS/IPS performance problem as PMEs are forced to inspect much more massive Internet flows. For instance, it is reported that the open-source *Snort* IDS/IPS spends ∼31% of its processing time on PM when subjected to real-world traffic and its CPU utilization reaches 80% when dealing with Web services due to the bulky Web-specific signatures in its database [9]. Emerging attacks also have the potential to trap PMEs into their worst-case working conditions [9]. It is therefore absolutely essential to 'accelerate' the intrusion detection process in PMEs and subsequently improve the overall performance of IDSs/IPSs.

Evasion techniques that include modification of telltale patterns, TCP segmentation or IP fragmentation, and message encryption [4,10] often make up a contemporary option for intruders. Such attacks could be mitigated should IDSs/IPSs dissect incoming traffic into a sequence of messages according to TCP/IP protocols before conducting their PM operations. To this end, stateful and layer-7 inspection on the ensued application messages has to be carried out [1,10]. The realization of the above counter-measures requires sophisticated PM process and expansion of signature databases. PM algorithms routinely used by PMEs such as the Aho–Corasick [11] and Boyer–Moore [12] are computationally intensive when it comes to generating positive verdicts for attack streams and negative verdicts for legitimate flows. Although a number of heuristics have been proposed to improve PM performance, they are often effective only with lengthy patterns [13]. The latter is not always possible as for instance, 90% of the patterns defined in the signatures of *Snort* are <16 bytes rendering heuristics rather ineffective. The sophistication of modern intrusions necessitates that IDSs/IPSs work with multiple-patterns to characterize a single attack and so a number of PM-iterations are needed to handle a single attack signature [9]. To improve detection accuracy, IDSs/IPSs may also specify complex relationship among multiple patterns in a signature with respect to their occurrence order, distances and positions [2], putting additional CPU burden on the PME.

Intrusion detection could also be facilitated by multi-pattern matching (MPM) algorithms that are typically derived by modifying single-pattern PM techniques to simultaneously search for multiple patterns with the help of rather complex data structures [14]. To this end, the Wu–Manber algorithm organizes telltale patterns using hash tables [9,14] but its memory footprint for a *Snort* implementation is about 29.1 MB when the signature database involves only about 1500 patterns [3]. Automata-based approaches such as the Aho–Corasick can also be extended to conduct MPM at the expense of very large memory footprints and high-dimensional state spaces [11,15]. In particular, the implementation of the Aho–Corasick algorithm in *Snort* requires 1 KB on an average for each character appearing in the patterns of the signature database and thus demands a time-consuming state traversal process [3]. Clearly, such memory consumption levels disqualify automata-based methods in light of populous signature databases. One alternative option to boost performance would be to implement MPM operations in hardware such as application-specific integrated circuits (ASICs) or field programmable gate arrays (FPGAs) [16]. Similar to their software counterparts, however, hardware accelerated PM methods can achieve only limited success due to a number of characteristics unique to the intrusion detection process:

(i) PMEs subject both legitimate and malicious network flows to the same CPU-intensive procedures heavily penalizing attack-free traffic. In this regard, a negative verdict for a legitimate stream requires inspection of the entire signature database. The fact that the vast majority of the Internet traffic is legitimate [17] points to an unnecessary punishment for benign connections.

(ii) Heuristics developed for most MPM techniques are efficient only for long sought-for patterns but do suffer performance-wise when short telltales are the search targets in incident detection [18].

(iii) Characters appearing in telltale patterns of attack signatures are not uniformly distributed but indeed scatter in the entire *ASCII* code range (that is $[0, 0xFF]$) implying extremely large memory footprints for automata-based MPM algorithms. When sizeable attack signature databases are present, such techniques are of limited value [9].

To address the above issues, we propose the *Fingerprinter* framework, which attempts to significantly accelerate the attack detection process by exploiting unique characteristics of Internet traffic and harnessing multiple fingerprinting methods. We represent each IDS/IPS attack signature with a concise fingerprint derived from the signature's telltale patterns. Such a digesting process essentially allows us to transform the complex space formed by attack signatures into a condensed space of signature fingerprints. By initially matching the digest of an incoming stream against the fingerprints instead of the actual attack signatures, our *Fingerprinter* can quickly identify

attack-free traffic. The PM algorithm Boyer–Moore is invoked only when the traffic stream under examination shares the same fingerprints with some attack signatures. To conduct the fingerprint matching (FM) as well as PM operations efficiently, the *Fingerprinter* is designed to work in two distinct stages:

- *programming phase*: signatures in the IDS/IPS database are fingerprinted off-line and represented with short digests using multiple summarization strategies including the *Bloom–Filter* and *Rabin–Fingerprint* methods. Compared with original signatures, fingerprints are much shorter, and as a result the space defined by the signature database is effectively compressed into its digest counterpart that assumes lower dimensionality.
- *querying phase*: fingerprints from incoming traffic are computed and matched against the digests of signatures created during the programming phase. Traffic receives a negative verdict and is declared as legitimate if it fails to match any fingerprint from the signature database; thus, no further PM processing is required. If the *FM* yields a positive identification, *Fingerprinter* executes the Boyer–Moore algorithm to ascertain that the connection in question indeed contains the exact patterns in the signature whose fingerprint resulted in the preliminary match.

The acceleration in detecting exploits stems from the observation that malicious telltales appear only infrequently. The majority of attack-free traffic should produce negative verdicts in the *FM* operation conducted over the space of signature fingerprints without the explicit assistance of PM algorithms. The lightweight *FM* process of our *Fingerprinter* is expected to offer significant performance gains compared with what the IDSs/IPSs system offers at this time. The combination of short-length telltale patterns and diverse attack types may yield false matches in the course of the *FM* procedure as streams and signatures could share the same digests but actually are different. The frequency of such occurrences termed *false match rate* affects the efficiency of *Fingerprinter* as every false match leads to the invocation of a CPU-intensive PM algorithm. Our *Fingerprinter* reduces this false match rate by integrating a variety of fingerprinting strategies including the *Bloom–Filter* and *Rabin–Fingerprint* methods into the digest generation.

To provide flexibility and attain high performance, we standardize the interfaces of *Fingerprinter* so that new fingerprinting methods can be activated in a plug-and-play fashion. We have implemented the *Fingerprinter* along with the proposed fingerprinting methods as a PME in *Snort* and evaluated its performance by employing a trace-driven testing methodology [19] with a variety of traffic traces. We have experimentally established the operational acceleration offered by *Fingerprinter* to the incident detection process. We showed that our prototype consistently outperforms *Snort* in conventional settings with an up to 2-fold speedup rate. The rest of the paper is organized as follows: Section 2 outlines related IDS/IPS PM techniques and Section 3 deals with our design and discusses our fingerprinting methods. In Section 4, we provide key findings of our evaluation while conclusions and future work are found in Section 5.

2. RELATED WORK

Single-pattern matching (SPM) techniques, which seek the occurrence of a pattern P in a text T, have been widely applied in information retrieval, automatic document classification and pattern recognition [11,12,20]. As SPM methods use character comparison to be their key functional element, their complexity is dominated by the number of comparisons involved having a lower bound of $(n - m + 1)$, where n is the length of T and m the size of P [21]. Efforts reported in [12,13,22–24] have investigated the reduction in the number of such comparisons. For instance, the Boyer–Moore algorithm exploited the *bad character* and *good suffixes* heuristics to eliminate unnecessary comparisons [12]. It has been shown [23,25] that Boyer–Moore incurs $O(n + \gamma m)$ comparisons with γ being the number of P's occurrences in T. We should point out that the effectiveness of heuristics in the Boyer–Moore method heavily depends on the length of P. Evidently, a long pattern does provide opportunities for large shifts when mismatching occurs. Comparisons in Boyer–Moore can be further reduced by using more sophisticated heuristics and/or higher memory consumption [13,22,23,26]. The Aho–Corasick SPM method follows an entirely different approach as it first constructs a finite automaton able to recognize P and then handles T sequentially [11]. Its computational complexity is $O(n+|\Sigma|m)$ where $|\Sigma|$ is the size of the character set used by both P and T. As Σ is often large in intrusion detection, Aho–Corasick-based IDS/IPS implementation calls for a sizeable memory footprint.

Compared with Boyer–Moore, the Aho–Corasick method can be readily extended to carry out MPM. The latter searches for the occurrence of any pattern from a set $S = \{P_i, i = 0, 1, \ldots\}$ in text T [14]. MPM Aho–Corasick tracks multiple matching patterns with the help of respective states and transitions. In a way similar to SPM, the MPM Aho–Corasick algorithm also demands a large amount of memory to accommodate the automaton structure despite the fact that initially patterns are clustered according to their shared prefixes in an attempt to lower memory consumption [9]. The MPM Aho–Corasick implementation in *Snort* needs an average 1 KBytes per pattern character [9], a challenging requirement to address in real settings.

The SPM Boyer–Moore method and its variants have also been redesigned to address the MPM problem [8,9,13]. To this end, the Boyer–Moore–Horspool (BMH) algorithm organizes patterns in a trie so that the Boyer–Moore can be applied to a set of patterns that share common prefixes [9,27]. In contrast, the AC–BM algorithm [8] uses suffix trees to organize patterns according to their shared suffixes, and consequently executes the Boyer–Moore on all patterns 'held' by the same suffix tree.

The Wu–Manber method [14] employs a hash-based table that controls the movement of the patterns when mismatches occur, and it may yield better performance than its MPM Boyer–Moore and Aho–Corasick counterparts for diverse PM tasks [28].

The attack detection through PM by its nature is an MPM problem as incoming traffic should be simultaneously examined against multiple signatures, each of which may contain more than one telltale patterns [8,9]. Although the MPM Boyer–Moore, Aho–Corasick and Wu–Manber approaches have been individually implemented and optimized for the open-source *Snort* IDS/IPS, it appears that there is no clear winner as far as diversified traffic types, variable population of attack signatures and different network environments are concerned [9,28]. To enhance performance, a number of proposals exist that either integrate multiple MPM techniques or select different MPM methods according to traffic types and characteristics in the attack signature databases involved [9]. For instance, it is a good choice to employ Boyer–Moore when the traffic type is Web services and Aho–Corasick for other streams. This is necessitated by the fact that the extremely large set of patterns in *HTTP*-specific attacks would render the memory-intensive Aho–Corasick an unworkable choice [8,9].

The real-time requirements for IDS/IPS operations have led to the hardware implementation of MPM methods [29,30]. Algorithms including the MPM Boyer–Moore and Aho–Corasick methods have been realized with dedicated processors, *ASICs* or *FPGAs* [30–32]. In [33], a multi-pattern matching system is proposed that de-multiplexes a traffic flow into several streams and spreads the load over parallel matching units that perform a string search using deterministic finite automata. In [34,35], the integration of pre-decoded wide parallel inputs with automata implementations offers high throughput rates. Similar pre-decoding techniques are also employed to design IDSs for Gigabit networks [36,37]. An IDS equipped with content addressable memory (CAM) is used to match traffic against attack telltales in a brute force manner [38]. The hardware-based pattern match engine in [39] uses a Bloom filter to construct a hash-table for attack patterns. To improve space and time efficiency, optimizations such as pattern alignments and redundancy elimination are under intense examination [40–42].

The exclusion-based PM approach relies on the observation that pattern P cannot appear in T if any portion of P is not found in T [17]. This technique represents every attack signature with a set of n-grams derived from the patterns of the exploit. By tokenizing incoming traffic into a set of n-grams, the method matches the n-grams of the input against those of signatures and declares the input as non-malicious if the intersection of the two n-gram sets is empty [17]. Should the n-gram set of an attack signature be indeed subsumed by that of the input traffic, the Boyer–Moore algorithm is invoked to scan the entire input for exact telltale patterns. Unfortunately, the effectiveness of the exclusion-based PM algorithm diminishes quickly as the number of signatures exceeds 1000 due to the increasing false match rate [32].

Content filtering techniques could be an alternative to PM when exact matches in the given text are not required [20,43]. In this context, *Bloom–Filter* based methods can quickly determine whether a given input T contains any pattern from a set S [44]. However, *Bloom–Filter* cannot pinpoint the exact matching location if there is one and may cause false positives due to the fact that entries in the fingerprint vector corresponding to a specific input may be marked separately by different patterns in S. *Bloom–Filter* techniques have been applied in hardware-based IDSs/IPSs as they can readily organize attack telltale patterns in a way that legitimate traffic can be quickly identified [20]. This is accomplished at the expense of computational overheads and large memory consumption [20,39]. As the *Shingle* and *Rabin–Fingerprint* digesting techniques have been successfully used to detect document similarity and establish plagiarism [45–48], they have also been adopted in the MPM procedure of IDSs/IPSs [18]. In contrast to prior efforts that use either filtering or fingerprinting techniques in isolation and fabricate them in hardware to detect attacks, our *Fingerprinter* is a pure software-based framework that integrates filtering, fingerprinting and PM methods, thus treating them as indispensable elements of the PMEs in IDSs/IPSs. In this regard, incoming streams are first fingerprinted and matched against those of signatures before an exact PM algorithm actually takes place. Moreover, the concurrent use of multiple fingerprinting methods helps reduce false matches that occur in fingerprint comparison and assists in significantly accelerating the attack identification process.

3. OUTLINE OF *FINGERPRINTER*

To avoid expending CPU-cycles for reaching negative verdicts, *Fingerprinter* subjects network traffic to a fingerprinting process before launching any other 'expensive' exact pattern matching operation. During its *programming phase*, *Fingerprinter* creates a fingerprint repository for attack signatures designed to help make quick decisions as far as the existence of exploits in input streams is concerned. While in its *querying phase*, *Fingerprinter* may rapidly deduce the legitimacy of a flow if there are no matches with digests found in its repository. If in doubt, the framework proceeds with an exact-PM Boyer–Moore algorithm to issue authoritative and final verdicts. In contrast to traditional PMEs of IDSs/IPSs, *Fingerprinter* only imposes a light-weight *FM* on benign network flows.

3.1. *Fingerprinter* design rationale

Fingerprints are short and compact tags that represent large and often complex objects such as raw data and documents [48]. A fingerprinting process 'transforms' objects into a space created by their fingerprints [18]. This transformation makes it possible to shift the operations conducted on the original data to processes taking place on concise signatures. This enhances

system performance as long as fingerprints of objects 'collide' with very low probability [49]. Fingerprints are usually generated by a digesting function defined as $f : \Omega \rightarrow \{0, 1\}^k$, where Ω is the set of objects whose fingerprint length is k. To obtain a compact space of fingerprints, the parameter k is typically small compared with the original representation of the objects in Ω. This inadvertently leads to fingerprint collisions—distinct objects with the same fingerprints—due to information loss in the digesting process. It is therefore desired that for any n distinct objects $S = \{s_i | s_i \in \Omega, i = 0 \ldots n-1\}$, the number of distinct fingerprint values $f(s_i), i = 0 \ldots n-1$ is equal to n with a high probability; at the same time, for any two objects s_i and s_j $(i \neq j)$, it should be highly unlikely that $f(s_i) = f(s_j)$.

By treating streams and/or signatures of exploits as complex objects that consist of character sequences, *Fingerprinter* represents both traffic and attack signatures with their fingerprints, so that it can predominantly operate on the 'reduced' space of fingerprints to detect intrusions. In this regard, *Fingerprinter* tokenizes its input text—either a network packet P or an attack signature R—into a series of shingles. To help manage overheads, *Fingerprinter* considers shingles having only fixed length of l bytes. Thus shingles, also termed l-grams, can be generated quickly by sweeping through the input with a sliding window of l bytes. The produced l-gram set for an input packet P can be described as $S_P = \bigcup_{i=0}^{|P|-l} \{P[i, (i + l - 1)]\}$, where $|P|$ is the length of packet P, $P[i, j]$ is the byte-sequence in P between positions i and j, and $\bigcup$ is the union set.

The same tokenization process is applied to telltale patterns of signatures that make up the IDS/IPS rule-bases. For instance, with the help of a 4-byte sliding window, *Snort* signatures depicted in column *Signature* of Table 1 can be tokenized to obtain bags of shingles. Signature *sid-1002* defines a single pattern *cmd.exe* with keyword *uricontent* and its set of 4-grams is {cmd., md.e, d.ex, .exe} as shown in column *Token set*. Table 1 depicts the sets of shingles derived for other signatures as well.

Should we apply a digesting function f on the shingle set of an attack signature R, we can obtain the fingerprint of R as $F_R = \bigcup_{i=0}^{|R|-l} \{f(R[i, (i + l - 1)])\}$. By assuming that f is a naive *XOR* function defined as $f(s) = (\bigoplus_{i=0}^{|s|-1} s[i])$ with $\bigoplus$ being the *exclusive-OR* operation and $s[i]$ the *ASCII* code for the character at position i of the input s, we may compute the fingerprints for the *Snort* signatures depicted in Table 1. For instance, the fingerprint of the 4-gram '*cmd.*' in the shingle set for Signature *sid-1002* can be calculated as $f(cmd.) = (0x63 \oplus 0x6D \oplus 0x64 \oplus 0x2E) = 0x44$. By repeating the digesting process on other three 4-grams of Signature *sid-1002*, we can derive its fingerprint as the set of {44, 42, 57, 56}. Similarly, the above digesting can be used to fingerprint the payload of packets. For instance in the Nimda attack traffic depicted in Table 2 consisting of 12 packets, only three packets have TCP payloads: packets 4 and 6 from attacker and packet 8 from victim. Packet 4 of the Nimda traffic in question contains the string '*cmd.exe.*' It thus features all 4-grams of *Snort* signature *sid-1002* in its set of shingles and subsequently subsumes the latter's fingerprints.

With fingerprints of both a packet P and a signature R at hand, we can quantify their similarity via a homology metric defined on respective fingerprint sets F_P and F_R as follows: $\sigma(P, R) = |F_P \cap F_R|/|F_P \cup F_R|$, where $\cap$ and $\cup$ are the intersection and union of the sets involved. The range of $\sigma(P, R)$ is $[0 \ldots 1]$ and $\sigma(P, R) = 1$ should P and R be identical. Similarly, we quantify the '*containment*' relationship between P and R as: $\pi(P, R) = |F_P \cap F_R|/|F_R|$; P contains R should $\pi(P, R) = 1$. We can readily see that the fingerprints of Nimda and signature *sid-1002* indicate containment as $\pi(P_4, sid\text{-}1002) = 1$. This is however not the case for *sid-1735* as its token '*file*' does not appear in any fingerprint sets of the Nimda packets yielding $\pi(P_i, sid\text{-}1735) < 1$ for all $i = 1 \ldots 12$.

By identifying attack signatures that are contained in packets emanating from network connection C, we can compile a set of candidate signatures, $R' = \{R_i | \pi(P_j, R_i) = 1, P_j \in C\}$. If R' is empty, *Fingerprinter* can fast determine the legitimacy of traffic from C. Otherwise, the exact-PM method is invoked to render the definitive verdict. For instance, the traffic depicted in the right column of Table 2 yields an empty R' set indicating a non-malicious flow. In contrast, the non-empty $R' = \{sid\text{-}1002\}$ for the Nimda traffic of Table 2 points out a stream that may comply with signature *sid-1002*; the latter is confirmed through the subsequent execution of the Boyer–Moore algorithm.

It is worth pointing out that IDSs/IPSs designate complex constraints that intrusions are expected to demonstrate in order to improve their detection accuracy. For example, the *Snort* signature *sid-1002* not only specifies the pattern '*cmd.exe*' as the telltale in the Nimda attack, but also requires that the telltale should appear in packets originated from the client side of an established TCP connection. Along these lines, to identify the exploit in the buffer overflow existing in *IIS*-servers, the *Snort* signature *sid-2572* specifies that the distance between patterns '*txtusername=*' and '|0A|' should be larger than 980 character positions. It is therefore conceivable for a network session to be declared attack-free even though it features a non-empty set of candidate signatures and may further contain exact telltale patterns specified in signatures. A false match occurs when there is a non-empty set of candidate signatures for a network flow which receives a positive verdict in the *FM* process but the flow in discussion fails the examination during the PM inspection. We term the frequency of false match occurrences as the *false match rate*.

Telltale patterns do appear at arbitrary positions either within protocol headers or the payload of packets. Besides examining traffic for the occurrence of telltales, IDSs/IPSs attempt to verify the correctness of locations as well as spatial relationships among patterns involved. The *FM* operation of the *Fingerprinter* may help accelerate the work of the PM process if it can pinpoint the positions of input substrings that match signature fingerprints; this information can substantially reduce the search space for the PM algorithm. The two phases, under which

TABLE 1. Tokenizing and fingerprinting some sample signatures in the rule-base of *Snort.*

SID	Signature	Token set	Fingerprint with XOR
1002	tcp $EXTERNAL_NET any → $HTTP_SERVERS $HTTP_PORTS (msg: "WEB-IIS cmd.exe access"; flow:to_server,established; uricontent: "cmd.exe"; nocase;)	cmd. md.e d.ex .exe	|44 42 57 56|
1735	tcp $EXTERNAL_NET $HTTP_PORTS → $HOME_NET any (msg: "WEB-CLIENT XMLHttpRequest attempt"; flow: to_client, established; content: "new XMLHttpRequest|28|"; content: "file|3A|//"; nocase;)	*new_ ew_X w_XM* ... *LHtt ttpR ... est|28|* *... le|3A|/ e|3A|//*	|5C 6A 42 79 11 ... |04 38 22 ...| |... 5A 1C 5F|
978	tcp $EXTERNAL_NET any → $HTTP_SERVERS $HTTP_PORTS (msg: "WEB-IIS ASP contents view"; flow: to_server, established; content: "%20"; content: "&CiRestriction=none"; nocase; content: "&CiHiliteType=Full"; nocase;)	*&CiR CiRe* ... *&CiH CiHi* ... *... =Ful Full*	|5E 1D 2D ... | |44 0B 24 ... | |... 6B 62 33|
1986	tcp $HOME_NET any <> $EXTERNAL_NET 1863 (msg: "CHAT MSN file transfer request"; flow: established; content: "MSG "; depth:4; content: "Content-Type|3A|"; distance:0; nocase; content: "text/x-msmsgsinvite"; nocase; distance:0; content: "Application-Name|3A|"; content: "File Transfer"; distance:0; nocase;)	*MSG_ Cont* ... *text ext/* ... *Appl ppli* ... *... nsfe sfer*	|79 36 10 ...| |1D 46 5B ...| |2D 05 16 ...| |... 1A 1E 02|
2572	tcp $EXTERNAL_NET any → $HTTP_SERVERS $HTTP_PORTS (msg: "WEB-IIS SmarterTools SmarterMail login.aspx buffer overflow attempt"; flow:to_server, established; uricontent: "/login.aspx"; nocase; content: "txtusername="; isdataat: 980, relative; content:!"|0A|"; within: 980; nocase;)	/log logi ogin gin. in.a n.as .asp aspx txtu xtus tuse user sern erna rnam name	|4B 0D 0F 4E 48 52 4C 1A| |0D 0A 17 11 0A 18 10 07 54|

Fingerprinter operates—*programming* and *querying*—are the outcome of the quest for efficient detection of exploit signatures. Figures 1 and 2 show the respective architectural lay-outs for the two stages, respectively.

In the *programming phase*, attack signatures are tokenized and digested so that the fingerprint space can be constructed. For an exploit signature R that defines a pattern r, the module *Input Scanner* of the *Fingerprinter* tokenizes r and builds its shingle set $S_R = \bigcup_{i=0}^{|r|-l}\{r[i, (i+l-1)]\}$, where $|r|$ is the length of pattern r and l is the size of the sliding window. A shingle set of a signature with multi-patterns is obtained by combining shingles from all its patterns, that is, $S_R = \bigcup_{j=0}^{|R|-1}\bigcup_{i=0}^{|r_j|-l}\{r_j[i, (i+l-1)]\}$, where $|R|$ is the number of patterns in signature R. For instance, the 4-grams from the five patterns defined by keyword *content* in signature *sid-1986* contribute to its shingle set, which contains token "*Cont*" from the second pattern and "*File*" from the last telltale. Subsequently, the *Fingerprint Generator* module selects a subset of l-grams from the shingle set for each signature R, and computes the *Rabin–Fingerprint* of the selected l-grams; in addition, k hash codes are also derived for each l-gram with the help of the *Bloom–Filter* techniques. The digest of R—its Rabin fingerprint and k hash values—are stored in the fingerprint table that is searched with the Rabin fingerprint as its access key. Given that the attack signatures recognized by the PME are known in advance, the programming phase is performed off-line once and gets updated only when changes in the signature database occur.

The *Fingerprinter* carries out its normal operation in real-time during its *querying phase*. For every incoming packet P, *Input Scanner* generates its shingle set with a sliding window of l bytes as shown in Fig. 2. After the creation of P's shingle set $S_P = \bigcup_{i=0}^{(|P|-l)}\{P[i, (i+l-1)]\}$, the *Rabin-Fingerprinter* module computes P's digest $F_P = \bigcup_{i=0}^{(|P|-l)}\{f(P[i, (i+l-1)])\}$. Signatures contained in F_P are then identified by the *Fingerprint Manager* that helps form the set of candidate signatures, $R' = \{R_i \mid \pi(P, R_i) = 1\}$. The latter is used by *Verdict Generator* to decide upon the legitimacy of packet P. Obviously, a signature R is a member of the set R' only if its Rabin fingerprint is contained in packet P. To further reduce the false match rate, the *Fingerprinter* also compares the k hash codes of the signature R against P before R is placed into R'. If a signature R is included in R', the sliding window used by *Input Scanner* helps locate the input substring that shares the common fingerprint with R. This facilitates the work of the Boyer–Moore algorithm, which follows and commences at the current sliding window instead of scanning the entire input stream. Moreover, the set of candidate signatures R' is constructed efficiently in our *Fingerprinter* with the help of optimized data structures that maintain the associations between signatures and their digests. We derive Rabin fingerprints of l-grams for an input packet in a

TABLE 2. Traffic for a Nimda attack and packet flow for a normal Web service.

No.	Dir	Payload	Token set
	Nimda: attacker (A) – 10.80.8.183/32872; victim (V) – 10.80.8.221/80		
1	A→V	(SYN)	
2	V→A	(SYN|ACK)	
3	A→V	(ACK)	
4	A→V	GET /scripts/..%255c../winnt/system /cmd.exe?/c+dir HTTP/1.1	*GET_, ET_/ T_/s, ...*
5	V→A	(ACK)	
6	A→V	|0D 0A|	
7	V→A	(ACK)	
8	V→A	HTTP/1.1 400 Bad Request|0D 0A| Date: Thu, 17 Jul 2003 ...	*HTTP, TTP/ TP/1, ...*
9	V→A	(FIN|ACK)	
10	A→V	(ACK)	
11	A→V	(FIN|ACK)	
12	V→A	(ACK)	
	Normal: client (C) – 10.80.8.183/4569; server (S) – 10.80.8.221/80		
1	A→V	(SYN)	
2	V→A	(SYN|ACK)	
3	A→V	(ACK)	
4	A→V	GET /cgi_bin/library/picturebase /execmd.eps HTTP/1.1	*GET_, ET_/ T_/c, ...*
5	V→A	(ACK)	
6	A→V	|0D 0A|	
7	V→A	(ACK)	
8	V→A	HTTP/1.1 200 OK|0D 0A| Server: Microsoft-IIS/4.0 ...	*HTTP, TTP/ TP/1, ...*
9	V→A	(FIN|ACK)	
10	A→V	(ACK)	
11	A→V	(FIN|ACK)	
12	V→A	(ACK)	

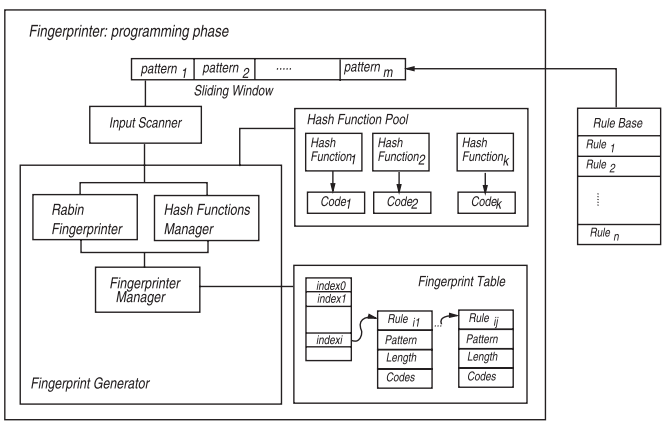

FIGURE 1. The *Fingerprinter* in the *programming phase*.

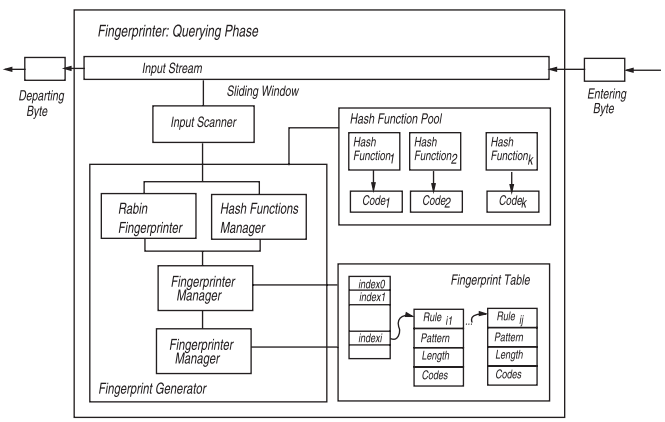

FIGURE 2. The *Fingerprinter* in the *querying phase*.

rolling manner so that the fingerprints of previous shingles can be reused in the computation of subsequent tokens. In addition, hash codes for an input are generated on-demand by the *Hash Function Manager* module and they are materialized only if the input matches the Rabin fingerprints of certain signatures. The stateful inspection of *Snort* allows *Fingerprinter* to generate a verdict for a network session and declares it as attack-free if all its constituent packets obtain negative verdicts. We provide additional detailed description for the functions of modules involved in Figs 1 and 2 in the remainder of this section.

3.2. Programming and querying the *Fingerprinter*

The tokenization process creates shingles for exploit signatures in the programming phase and for input traffic in the querying phase. The efficiency of tokenization is determined by the size l of the sliding window as: first, size l determines the uniqueness of the produced tokens with respect to attack telltales. Apparently, tokens derived from malicious telltales with large l are still unique with high probability to specific and infrequent exploits. Second, the computational complexity of fingerprint calculation on a l-gram linearly depends on l; small-sized sliding windows are naturally favored in this regard. Finally, parameter l also affects the *Fingerprinter* false match rate as small l generates short tokens that may destroy the information context of the original patterns and consequently weaken the framework's differentiation capability [18]. For instance, if $l = 4$ bytes, the *Snort* signature *sid-1002* has a four-element set of 4-grams, {*cmd., md.e, d.ex, .exe*}. With the same size of the sliding window $l = 4$, the tokenization process creates a set of shingles for packet 4 of Nimda in Table 2, which has a subset of {*cmd., md.e, d.ex, .exe*}; the set of 4-grams for packet 4 of the attack-free traffic presented in the right column of Table 2 contains elements of {*exec, xecm, ecmd, cmd.*}. Clearly, packet 4 of Nimda contains all the shingles of Signature *sid-1002*. By reducing l to 2, we have the token set for signature *sid-1002* to be {*cm md d. .e ex xe*} with the token set for packet 4 in the traffic from the second half of Table 2 including all the 2-grams of Signature *sid-1002*, resulting in a false match.

Based on our experiments, a window size in the range of [4 . . . 8] bytes achieves superb performance for various network traffic types and mixture of benign/malicious streams. Thus, we use a sliding window of $l = 4$ bytes as default in the *Fingerprinter* operation. The efficiency of *Fingerprinter* also depends on the penalties imposed by the digest computation in the fingerprinting process for both signatures (off-line) and input packets (in real-time). In this regard, the fingerprint derivation for an l-gram is definitely affected by the length l of the token (determined by the sliding window size). The fingerprinting process is also influenced by the digesting function f used. The latter could be either CPU-intensive as it may entail expensive manipulations including divisions and multiplications or lightweight as it would involve bit-shifting functions and *XOR*s that are now efficiently carried out by modern OSs. Furthermore, the false match rate caused by the *FM* operations does significantly affects the performance of *Fingerprinter* as it increases the frequency with which exact-PM Boyer–Moore algorithm is triggered without yielding positive identifications.

Algorithm 1 The *programming process* in *Fingerprinter*.

1: initialize the fingerprint table F with size of m entries to be zero;
2: register k functions to establish the hash pool H;
3: **while** (there is unprocessed signature s in the rule base S) **do**
4: find the longest pattern p in patterns of s;
5: **if** (signature s defines no telltale pattern) OR (pattern p is shorter than sliding window's size l) **then**
6: mark s as "non-fingerprintable", put s into the set of R, and process next signature;
7: **end if**
8: extract the first l bytes of p to obtain substring p';
9: compute the Rabin's fingerprint of p' to get c; $c \leftarrow (c \text{ modulo } m)$;
10: materialize entry c of fingerprint table F with c as the key and signature s as the value;
11: linked-list is used to organize multiple signatures when collision occurs;
12: **for** (each hash function h in the hash pool H) **do**
13: compute hash code $d = h(p')$; store d in the fingerprint table $F[c]$;
14: **end for**
15: **end while**
16: the construction of Rabin hash table H is completed; H is switched to work in "querying" mode;

The *Fingerprinter* reduces overheads and the false match rate by integrating multiple efficient fingerprinting techniques, namely:

- *Efficient fingerprint computation*: in tokenizing incoming flows with an l-byte sliding window, the l-gram in the current window shares $(l - 1)$ characters with the just previous l-gram. This implies that the *Rabin–Fingerprint* techniques employed in our *Fingerprinter* accelerate the fingerprinting process by reusing the computational results from previous tokens in the derivation of fingerprint for the current token.
- *Determination of candidate signatures*: for traffic that shares the same Rabin fingerprints and Bloom-hash codes with certain attack signatures, the set of candidate signatures can be identified quickly with the help of optimized data structures. This reduces the invocation of the PM algorithm from matching packets against the entire signature database to the candidate signatures only.
- *Identification of matching point in traffic*: any time an FM occurs, the position of the match in the incoming traffic can be located accurately with the help of the sliding window. Thus, subsequent PM searches for exact patterns may commence from specific locations instead of scanning the entire flow.

Depending on the location of their appearance, *Snort* defines telltales for exploits with keywords *uricontent* and *content*. The former expects the specified patterns to occur in the field of universal resource identifier (URI) in Web traffic and the latter searches for patterns in IP payloads. Telltale patterns delineated with the above two keywords (*uricontent* and *content*) necessitate the invocation of Algorithm 1 twice as the two individual

searches work on unrelated locations. Algorithm 1 outlines the work of the programming phase: it first extracts the longest telltale pattern from each attack signature, and then computes the Rabin fingerprint based on the l-byte prefix of the longest pattern. In addition, a set of k hash codes are also generated with the help of a hash function pool in the Bloom-filter of the *Fingerprinter*. The Rabin fingerprint and Bloom hash codes are stored in a fingerprint table. In case a collision occurs on the Rabin fingerprint, linked-list helps organize all 'colliding' signatures.

As mentioned before for efficiency reasons, the shingle computation for Rabin fingerprints is done in a rolling manner. The k hash values of shingles for exploit signatures are always computed in Algorithm 1 and so, the overhead in question does not affect the operation of the *Fingerprinter*'s subsequent querying phase. Should we assume that Rabin fingerprints in attack signatures are uniformly distributed, the requisite table should have $M = 2^{|l|+8}$ entries to accommodate any possible Rabin fingerprint code. Clearly, the *Fingerprinter* memory-footprint increases exponentially with l. To achieve as compact as possible memory requirements l has to be small on the one hand, while on the other hand a large l decreases fingerprint collisions improving overall performance. Signatures marked as 'non-fingerprintable' in Algorithm 1 are processed separately with the *Snort* conventional PME.

Algorithm 2 The *querying process* in *Fingerprinter*.

```
1:  align the l-byte sliding window on packet P at index i = 0;
    initialize the set R of candidate signatures;
2:  while (sliding window completely covers the packet P) do
3:     compute the Rabin's fingerprint c of input substring in the sliding window;
4:     if (fingerprint table entry keyed by c is empty) then
5:        advance sliding window by one byte (i.e., i ← (i + 1)); process next substring;
6:     end if
7:     compute k hash codes for current substring with functions in hash pool H;
8:     for (each signature s in fingerprint table entry F[c]) do
9:        if (s shares same k hash codes as current substring) and (longest pattern of s is
          in P starting at current window) then
10:          insert signature s into R along with position of current window;
11:       end if
12:    end for
13:    advance sliding window by one byte (i.e., i ← (i + 1));
14: end while
15: if (candidate signature set R is empty) then
16:    assign a negative verdict for P;
17: else
18:    invoke boyer-moore algorithm on R and P; return its verdict for P;
19: end if
```

Algorithm 2 delineates the operational work of *Fingerprinter* in its querying phase. The algorithm uses the l-bytes sliding window to yield a set of l-grams for the corresponding input stream. The Rabin fingerprint for a l-gram is computed and is used as key to access the fingerprint table established in the programming phase. Should an empty entry be encountered, no further PM inspection is required. Otherwise, Algorithm 2 computes the k hash codes of the l-gram by using the function pool in the Bloom-filter. The signature residing in the entry of the table is not a candidate if its k hash values differ from those of the current l-gram. The same process is repeated for every signature in the entry if the latter accommodates multiple signatures. It is rather unlikely for a shingle from an attack-free traffic to share both the Rabin fingerprint and k hash codes with a malicious telltale pattern. Thus, we expect that most shingles should hit empty slots in the fingerprint table asking for no further action.

If a token indeed lands on a non-empty entry of the fingerprint table and has simultaneously the same k hash codes with the signature in the entry, additional action is taken by Algorithm 2 to further verify that the input starting at the current window actually matches the signature's longest pattern, and the signature is considered as a candidate only if the verification is positive. Evidently, the Boyer–Moore algorithm is only executed when the above-described *FM* operations yield a non-empty set of candidate signatures. Moreover, the information on matching positions in the input stream recorded by Algorithm 2 during the *FM* process also facilitates the performance of the Boyer–Moore as the latter's search space is substantially reduced. Algorithm 2 involves light-weight operations for most legitimate traffic as tokens from the latter result in an empty set of candidate signatures with high probability.

3.3. Digest functions and fingerprint computation

The main digesting method of the *Fingerprinter* is the Rabin's fingerprinting approach that treats each l-gram of a given input string as an l-digit number in a certain base b. In this context, an l-gram with byte sequence of $A_1 = \{a_1, a_2, \ldots, a_l\}$ can be represented as an $(l-1)$-degree polynomial with base b : $A_1(b) = a_1 b^{l-1} + a_2 b^{l-2} + \cdots + a_l$, and its Rabin fingerprint can be computed as $f(A_1) = A_1(b) \bmod P(b)$, where $P(b)$ is an irreducible polynomial of k degree. Suppose that the above l-gram is only a substring of a given long input stream, and the next immediate l-gram has the byte sequence of $A_2 = \{a_2, a_3, \ldots, a_{l+1}\}$, obviously, A_1 and A_2 share the same $(l-1)$ characters. Subsequently, the computation of the Rabin's fingerprints for A_2 can use a portion of the result from the previous l-gram A_1. This is what we have earlier termed the rolling computation of fingerprints. In particular, the Rabin fingerprint of the l-gram A_1 can be expressed as $f(A_1) = (a_1 b^{l-1} + a_2 b^{l-2} + \cdots + a_l) \bmod P(b) = r_1 b^{k-1} + r_2 b^{k-2} + \cdots + r_k$, while the fingerprint of the l-gram A_2 can be derived as $f(A_2) = ((f(A_1) - a_1 b^{k-1})b + a_{l+1}) \bmod P(b)$. As b^{k-1} is a constant given that base b and degree k of P are fixed, $f(A_2)$ can be obtained from $f(A_1)$ efficiently with only two additions and two multiplications. It is known that among n randomly chosen strings with degree l, the probability for a Rabin fingerprint collision of two distinct strings is less than $(nl^2/2^k)$. Hence, the collision rate can be controlled by manipulating parameters k, l and n according to application environments [48]. To reduce its computational complexity, *Fingerprinter* sets base b to two so that multiplications in Rabin fingerprint calculation can be replaced with left-shift operations. We further decrease computational penalties by representing

each signature with the Rabin fingerprint derived from the l-byte prefix of its longest telltale pattern instead of all its shingles.

The *Fingerprinter* also resorts to *Bloom–Filter* methods to efficiently settle membership of a given input string in a set of pre-specified patterns [20,44]. Consisting of k hash functions and an m-bit fingerprint vector, a Bloom filter operates in two phases. In the programming mode, the Bloom filter attempts to remember a set of pre-specified objects by materializing the fingerprint vector with the help of object digesting. More specifically, a set of k hash codes is first generated for each object through the hash function pool and then each resulting hash value acts as an index into the fingerprint vector to set the corresponding bit. In the querying mode, the task of the Bloom filter is to determine if a piece of input is identical to any of the objects that are 'cognizant' by the filter. To this end, the Bloom filter inspects the bits in the fingerprint vector corresponding to the k hash values of the input generated with the same hash function pool. A positive verdict for the input is created if all k bits are marked. Otherwise, a negative verdict is issued.

A Bloom filter always correctly identifies non-member objects as any unmarked fingerprint bit excludes a non-member under test. We have to point out that the membership of an object may not perfectly match a known object as the k fingerprint bits associated with the input object could be checked off separately by different pre-specified objects leading to false positives. By either deploying much larger hash function pools or using orthogonal hash functions that generate codes independently the number of false alarms can be diminished.

Although it appears that Bloom filtering methods can be readily applied to the *FM* process, in general their application in IDS/IPS intrusion identification presents a number of challenges:

- The ever-expanding IDS/IPS signature databases make the set of patterns to be recognized by a Bloom filter very large creating massive main-memory requirements. Multi-pattern attack signatures also present challenges as far as the compact derivation of their fingerprints is concerned.
- The diversification in the length of telltales implies that the membership of a given string in the signatures can only be determined by inspecting all its substrings whose sizes are in the range of [*LSP, LLP*], where *LSP* and *LLP* are the length of the shortest and longest patterns of all signatures, respectively. This evidently is computationally intensive. Moreover, the relationships among telltale patterns such as overlap and containment call for complex structures to help organize and efficiently maintain signatures.
- The false positive rate may deteriorate along with the ever-expanding signature database due to increased collisions. Furthermore, the inability to exactly pinpoint the corresponding telltale patterns matched by the input traffic does hurt the overall performance of Bloom filters.

In light of the above, Bloom-based methods may be applied in IDSs/IPSs when hardware and parallelization options are used so that multiple Bloom filters can be deployed simultaneously to process input [43]. We overcome these challenges in *Fingerprinter* by representing each attack signature with the l-byte prefix of its longest pattern. In this way, signatures are organized with a single filter. Instead of programming all telltale patterns into a single fingerprint vector, *Fingerprinter* stores hash codes of each signature separately within entries of the fingerprint table using the Rabin fingerprint as key. As depicted in Figs 1 and 2, the relations among signatures, telltale patterns and their hash codes are maintained with the fingerprint table facilitating the identification of candidate signatures encountered in network flows. By subjecting shingles of the input to the Bloom-filter only when these shingles match the Rabin fingerprints of certain signatures, we further reduce computation overheads as Bloom-filters are only executed on-demand.

Similar to Rabin's fingerprinting method, the performance of the Bloom filter method in the *Fingerprinter* is affected by parameter l as the latter determines the differentiation capability of the Bloom filter. Obviously, a large l improves the uniqueness of the prefixes extracted from telltales, which are programmed into the Bloom filter. Larger l values also generate fewer tokens for an input stream and help accelerate the detection process. Small l values can help substantially reduce the processing required for hash code generation. The performance of the Bloom filter also depends on the size of its hash function pool and the footprint of its corresponding fingerprint vector. A large pool of hash functions could decrease the false positive rate but at the cost of high computational intensity. Conversely, a sizeable fingerprint vector does reduce the hash code collisions at the expense of high memory consumption. Finally, the choice of hash functions does affect the performance of *Fingerprinter* and thus, we use hash functions that are both flexible to process input with arbitrary length and are adaptive to diverse types of traffic. In this regard, the operations on hash code computation are designed to be lightweight so that they offer a viable overall software implementation.

When multiple hash functions are configured into the pool of the Bloom-filter, the *Fingerprinter* selects functions that generate hash codes independently with diversified operations. This is done so that no close correlation exists in the resulting codes helping reduce hash code collisions. Each hash function is thoroughly tested and fine tuned to ensure that the generated hash codes are non-linear with input. Moreover, we consider functions that are complete; this requires that a flip in a single input bit affect most output bits and the derivation of every output bit depend on all input bits [50]. The hash functions in discussion are mainly constructed from one-way functions typically used in cryptography and digital signatures [51], some of which are shown in Table 3. Each hash function has its identifier listed in column *Function*, and column *Operations* outlines key computational operations on each input byte. For instance, the naive hash function *XOR* applies the operation $(c \oplus P[i])$ to the ith byte of the input P with c being the

TABLE 3. Some of the hash functions used by *Fingerprinter*.

No.	Function	Operations	Explanation
Input: $P[0..(n-1)]$; Output: c is the hash code generated by corresponding hash function			
1	XOR	$c = (c \oplus P[i])$	c is hash code with initial value zero; $\oplus$ is the exclusive *OR*
2	RS	$c = (c \ll 7 + P[i]) + (c \ll 3)$	$\ll$ is left shift operator
3	PJW	$c = ((c \ll 4) + P[i])$ $c = (c \oplus (c \gg 24))$ if high byte of c is not empty	$\oplus$ and & are exclusive *OR* and *AND*; $\ll$ and $\gg$ are left and right shifts
4	ELF	$c = ((c \ll 4) + P[i])$; $c = ((c \gg 24) \oplus c)$	$\oplus$ is exclusive *OR* operator; $\ll$ and $\gg$ are left and right shifts
5	BKDR	$c = (c \ll 5) - c + P[i]$	$\ll$ is left shift operator
6	JS	$c = (c \oplus ((c \ll 5) + P[i] + (c \gg 2)))$	hash code c is initialized to 131; $\ll$ and $\gg$ are left/right shifts
7	DJB	$c = ((c \ll 5) + c) + P[i]$	hash code c is initialized to 5381; $\ll$ is left shifts
8	DEK	$c = ((c \ll 5) \oplus (c \gg 27)) \oplus P[i]$	hash code c is initially zero; $\ll$ and $\gg$ left and right shifts
9	AP	$c = (\neg(c \ll 11) \oplus P[i] \oplus (c \gg 5))$	"¬" is negation operation; $\ll$ is left shifts
10	SDBM	$c = P[i] + (c \ll 6) + (c \ll 16) - c$	c is initialize to zero; $\ll$ is left shifts

accumulated hash code generated by previous input bytes and $\oplus$ being the exclusive-*OR*. The hash code for an input string P with length of n bytes can be expressed as $c = (P[0] \oplus P[1] \oplus \cdots \oplus P[n-1])$. It is evident that the hash function *XOR* generates only a small hash code space as its hash values are limited in the range of [0, 255]. In contrast, other functions in Table 3 have much larger code space, which helps reduce collision rates. For example, by applying the hash function *DJB* to the *Snort* signatures of Table 1 to generate 16-bit hash codes, we can obtain the fingerprint for signature *sid-1002* as {*40AE, 1625, C518, A365*}. The fingerprint for signature *sid-1735* contains the subset of {*A300, 6058, 0B4D, 69EC,* ...}. By and large, Table 3 shows that key operations in the hash functions are bit shifts, exclusive-*OR*, and modulo, all of which are efficiently implemented in modern computer systems.

4. EXPERIMENTAL EVALUATION

We have implemented the *Fingerprinter* as a PME of the open-source IDS/IPS *Snort*. Figure 3 depicts the architecture of the system, which can either intercept live network streams or extract packets from traces stored in files. *Snort* can be configured to work in two modes: in passive detection, *Snort* monitors network traffic and generates alarms whenever a malicious connection is detected; while in pro-active prevention, *Snort* may further act with countermeasures such as packet drop or session termination on identified malicious streams. In both modes, incoming traffic is first rearranged by the component '*TCP/IP Protocol Dissector*' into a sequence of packets according to TCP/IP specifications before it gets subjected to security inspection. To improve attack detection accuracy, the *Snort* also resorts to layer-7 protocol analysis with the help of its module '*Application Layer Analyzer*'.

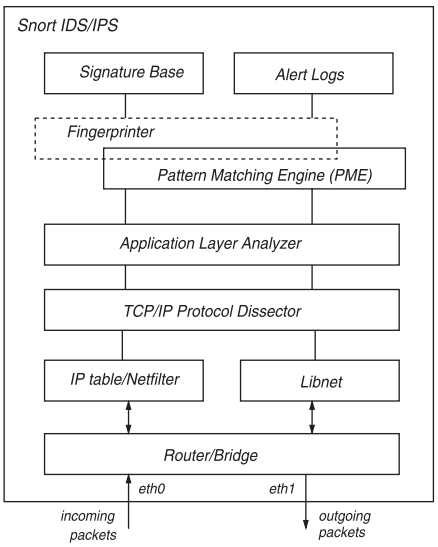

FIGURE 3. Key *Snort* components with *Fingerprinter* serving as PME.

For instance, the URI in *HTTP* protocols is identified and normalized by the module so that telltale patterns in URIs can be readily located. In this context, the Nimda attack is discovered by searching in URIs for the pattern *cmd.exe* as demonstrated by *Snort* signature *sid-1002* of Table 1. At the same time, the module attempts to decode certain types of data streams including simple network management protocol (SNMP) messages encoded in abstract syntax notation one (ASN.1). For network traffic including instant messengers, *P2P* protocols and Backdoors that masquerade or encrypt their communications, the module may be able to decrypt if the cryptographic algorithms and keys are available so that a PM process can be carried out in plaintext instead of ciphertext.

The stateful inspection is provided in *Snort* by maintaining the correlation between the bidirectional traffic streams for each connection and tracking the progress of data transmissions. For instance, input is declared as a Nimda attack only if the sought-for pattern *cmd.exe* appears in the packets from the client-end of an established TCP connection. The *Snort* heavily relies on PM techniques for intrusion detection and its performance is dominated by the PM algorithm in its PME. A number of PM algorithms including Boyer–Moore and Aho–Corasick have thus been implemented as part of the *Snort* module '*Pattern Matching Engine* (PME)'. The default PME is the Lowmem method that is a set-wise multi-pattern Boyer–Moore algorithm [9]. Lowmem clusters attack signatures into different groups according to the protocol types and network ports involved. It further organizes telltale patterns of signatures within each group with a prefix tree, and invokes Boyer–Moore on the input packet if the latter matches a pattern in any prefix tree [9]. The modularized design of the *Snort* helps in the ease of replacement of '*Pattern Matching Engine* (*PME*)' module should this be desired. We developed the *Fingerprinter* to function as a *Snort* PME; this is depicted as '*Fingerprinter*' in Fig. 3. The rationale for this design decision was to have *Fingerprinter* seamlessly integrate with other *Snort* components by replacing the module '*Pattern Matching Engine* (*PME*)' of the official release.

We conducted experiments on a 2.80 GHz CPU, 1 GB main memory machine running *RedHat* Linux OS. The test machine was equipped with *Snort v.2.6.0.2* whose signature database was released on 24 September 2008. The 4637 signatures in the database are crafted to identify the majority of contemporary attacks. We assumed the 'out-of-the-box' *Snort* configuration and so all attack signatures were activated. Experiments were conducted on multiple traffic traces following a trace-driven test methodology [19]. By configuring *Snort* to use different PMEs including Lowmem as well as our proposed *Fingerprinter* and its variants, and feeding it with traffic traces at the maximum throughput allowed by the test machine, we evaluated *Snort*'s performance mainly in terms of processing time and memory consumption.

To ensure that various PMEs deliver the same and correct behavior, we compared the respective alert logs generated to verify their consistency and ascertain the same attack detection capability. Whenever a different experiment setting was materialized, we restarted the test machine to minimize the interference from previous experiments. To further reduce noise, we conducted multiple experiments for every single configuration and evaluated the outcome based on the best performance attained.

4.1. The *Fingerprinter* and its variants

To facilitate the *Fingerprinter* evaluation and quantify its contributions to the intrusion detection process, we designed and modularized its components in a way that they can be integrated into the framework in a plug-and-play manner. This flexibility enables the easy creation of a variety of PMEs that function with different fingerprinting methods. In particular, we derive three *Fingerprinter* variants: *Bloom–Filter PM*, *Shingling PM* and *Rabin–Fingerprint PM*. For convenience, we name the PME depicted in both Figs 1 and 2 as the *Hybrid PM* method as it activates and employs multiple fingerprinting methods.

We obtain the *Bloom–Filter PM* by enabling the *Fingerprinter* Bloom-filter related components. Figures 4 and 5 show the respective *programming* and *querying phases*. During its programming phase, the *Bloom–Filter PM* represents each attack signature with the l-byte prefix of its longest telltale pattern and programs it into a Bloom-filter consisting of k hash functions and a fingerprint vector with m bits. The k hash codes obtained by applying k hash functions to the l-byte prefix of the longest pattern for an attack signature serve as indices to the fingerprint vector to mark the corresponding bits. The fingerprint vector can be treated as a two-dimensional array accessed with key pair (byte, bit), which is computed for a hash code c with the help of formulas byte $= (c \gg 3)$ and bit $= (1 \ll (c \bmod 8))$,

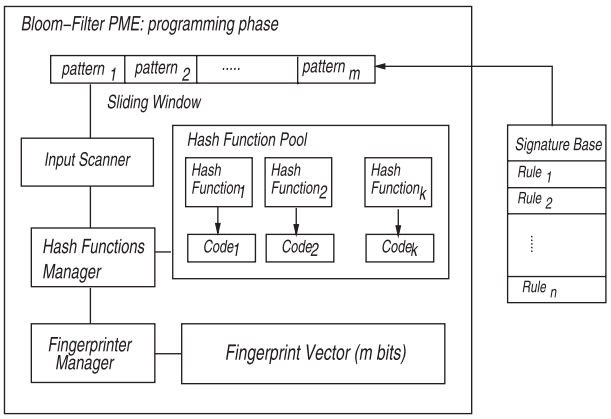

FIGURE 4. *Bloom–Filter PM*: programming phase.

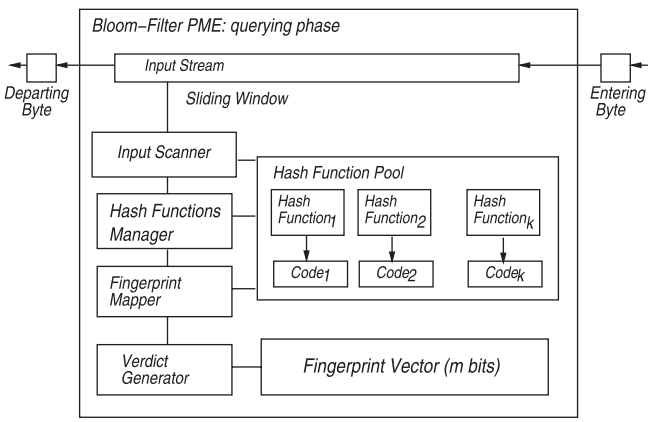

FIGURE 5. *Bloom–Filter PM*: querying phase.

where $\gg$, $\ll$ and mod are right-shift, left-shift and modulo operations, respectively. Once in the querying phase, the *Bloom–Filter PM* tokenizes the input into a series of l-grams with an l-byte sliding window, and computes k hash codes for each l-gram with the same hash function pool as that used in its programming phase. The k bits in the fingerprint vector associated with the k hash values of the l-gram are checked to determine whether the l-gram is recognized by the Bloom filter. If at least one l-gram of the input traffic is recognized by the Bloom filter, the Boyer–Moore algorithm is invoked to generate the final verdict.

As all signatures in the *Bloom–Filter PM* share the same fingerprint vector, it is impossible for the filter to identify which signature has created a match with the input stream. This requires the invocation of the Boyer–Moore algorithm against all possible signatures in the IDS/IPS rule base. Also, the Bloom filter could raise false positives on the membership of an l-gram as its corresponding k bits in the fingerprint vector could be marked independently by patterns from different signatures. Consequently, the *Bloom–Filter PM* false match rate may be significant when either a large signature database is used or the memory is limited. Obviously, the performance of the *Bloom–Filter PM* can be tuned by manipulating the size of the hash function pool, length of the sliding window and the footprint of its fingerprint vector.

The acceleration on the intrusion detection process achieved by the *Bloom–Filter PM* comes from its ability to rapidly distinguish legitimate input from malicious traffic. However, its inability to identify specific signatures whose fingerprint is matched by the input impairs its performance. It is thus vital as far as efficiency is concerned to maintain associations between signatures and their fingerprints. This is the rationale for *Shingling PM* whose two phases are depicted in Figs 6 and 7. In the programming mode, each attack signature is represented by the digests of the fixed-length shingles—sequence of characters—derived from its telltale patterns. The digests are organized with a signature table which stores telltales and their fingerprints in linked-list data structures as described in Fig. 6. While in the querying phase, the input traffic is tokenized into its l-grams and their digests are used as indices into the input fingerprint vector to mark the corresponding bits. The candidate signatures are detected by traversing the signature table sequentially and matching each signature's digest against the input fingerprint vector. An empty set of candidate signatures guarantees the legitimacy of the input stream; otherwise, the Boyer–Moore algorithm is invoked. It is evident from Fig. 7 that the *Shingling PM* materializes the input fingerprint vector with all shingles of the input before the signature table is traversed, making it impossible to locate the input substrings that result in FMs. Therefore, whenever a digest match occurs in the *Shingling PM*, the Boyer–Moore should be executed to search the entire input stream.

The *Shingling PM* improves the attack detection process by maintaining the associations between signatures and their

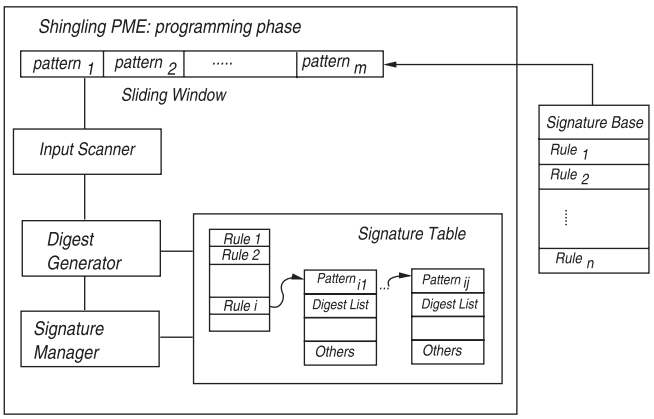

FIGURE 6. *Shingling PM*: programming phase.

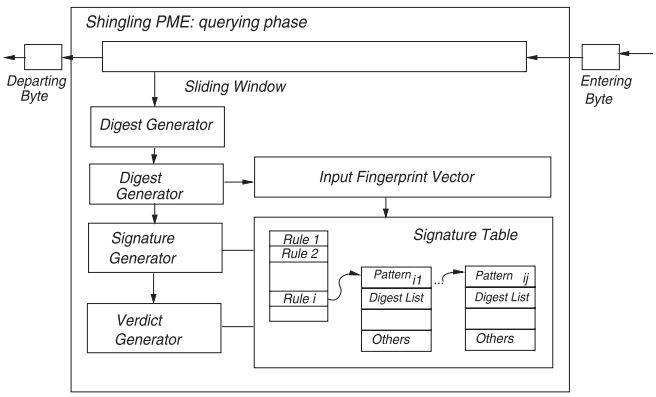

FIGURE 7. *Shingling PM*: querying phase.

fingerprints; however, it is not able to locate input substrings that lead to digest matches. In contrast, the *Bloom–Filter PM* can quickly pinpoint the matching input substrings but provides no mechanism to identify matched signatures. It is by now clear that the *Shingling PM* and *Bloom–Filter PM* are complementary in their roles and both help accelerate the intrusion detection process.

Figures 8 and 9 depict the architectural choices for *Rabin–Fingerprint PM* which actually tracks input with the sliding window and at the same time maintains the relation between signatures and their fingerprints. In its programming phase, the Rabin's fingerprint of the l-byte prefix of each signature's longest telltale pattern is computed and organized with a fingerprint table shown in Fig. 8. Moreover, linked-lists are used to accommodate colliding signatures that share the same fingerprints. While in the querying phase, the input packet is swept with an l-byte sliding window, and the Rabin fingerprint of the l-gram within the sliding window acts as the access key to the fingerprint table. Signatures within the landing entry of the fingerprint table form the set of candidate signatures for the current l-gram, and Boyer–Moore may be invoked if necessary.

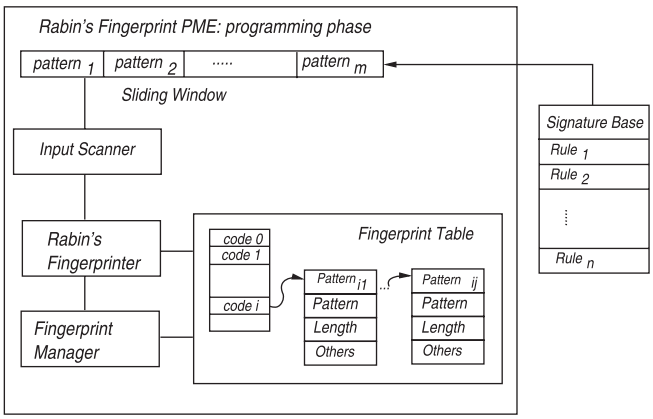

FIGURE 8. *Rabin–Fingerprint PM*: programming phase.

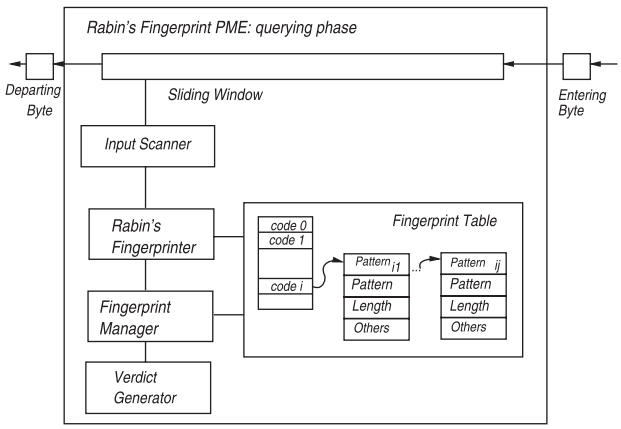

FIGURE 9. *Rabin–Fingerprint PM*: querying phase.

In summary, the hash function pool in the *Bloom–Filter PM* helps identify legitimate traffic quickly while the associations between signatures and their fingerprints in the *Shingling PM* facilitate the identification of matched signatures. By maintaining the associations between fingerprints and signatures with a fingerprint hash table and tracking input substrings with a sliding window, the *Rabin–Fingerprint PM* could further improve attack detection. While the full-fledged *Fingerprinter* activates simultaneously multiple fingerprinting techniques including *Rabin–Fingerprint* and *Bloom–Filter* to achieve better performance than its counterparts.

4.2. Data sets used in the *Fingerprinter* evaluation

Characteristics of wide-area Internet traffic has been studied for many years [52,53] and recently, the make-up and patterns of intranet streams have also been under investigation [54, 55]. The characterization of such traffic publicly available is predominantly based on LANs typically hosting a small number of systems [56] and often focuses on specific characteristics such as services and host communities that share common interests [53,54]. In general, it is challenging to thoroughly investigate Internet/Intranet traffic due to technical difficulties on collecting network activities when it comes to the coordination of multiple choke-points around the globe [55]. Considerations on privacy and intellectual property further exacerbate the problem by making publicly accessible traffic traces with full payloads required for IDS/IPS evaluation a scarcity [19]. Moreover, it has been established that traffic observed at different sites varies significantly and network payloads also evolve over time [57]. Consequently, the very notion of 'typical' traffic for Internet/Intranet network streams is not well defined [55]. For instance, peer-to-peer file sharing is widespread in some environments [58], but are rarely observed in other organizations [59]. We therefore use multiple traffic traces from various sites captured at different time periods to assess the effectiveness and efficiency of *Fingerprinter*.

The first set of traffic traces, known as *KDD CUP'99* data set and depicted in Table 4, was collected by the MIT Lincoln Laboratory for IDS testing and performance benchmarking [60]. The traffic was captured within three weeks of 1999 and each trace was generated by merging packet flows from simulated military networks serving as background noise and streams containing attacks as the foreground traffic. Each data set is assigned an identifier shown in column *ID* in addition to its descriptive name in the column '*Data set name*'. The volume for each traffic trace presented in the column *Size* indicates that daily network activities vary dramatically. The mixture of network protocols in each trace, mainly connection-oriented TCP and connectionless UDP, also differs significantly as demonstrated by the columns *TCP* and *UDP*. For instance, the data set *1999 Train Week2 Wednesday* comprises 89.55% TCP and 1.28% UDP connections, respectively; in contrast, the TCP/UDP ratio in the data set *1999 Train Week3 Friday* is 95.78/0.60. A trace with a heavy TCP/UDP protocol mixture does force IDSs/IPSs to spend much more CPU cycles and expends more main memory on tracking network connections and their communication progress when stateful inspections are in place.

The total number of packets in a trace does affect the IDS/IPS performance. For example, there are different expectations when an IDS/IPS deals with the data set of *1999 Train Week2 Wednesday* having only 888139 packets than when dealing with the set *1999 Train Week1 Thursday*, which consists of 1807060 packets. We compute the average packet size in each trace (column *Avg pkt*) as the ratio of *Size* and *Total pkts*. We should point out that the count of the column *Total pkts* includes packets that may have no payload at all (for example TCP ACKs). In this respect, column *Avg pkt* may underestimate the average packet size. Ten traces are collected in weeks 1 and 3 and labeled as attack-free while all data sets in week 2 contain malicious activities such as denial-of-service attacks, buffer overflow exploits and port scans. By feeding all traces to *Snort* with its out-of-the-box configuration and all its signatures enabled, we observe the generation of alerts for all traces including those

TABLE 4. Statistics on traces in the *KDD* data set collected by MIT Lincoln Laboratory in 1999.

ID	Data set name	Size (bytes)	TCP	UDP	Total pkts	Avg pkt	Attacks	Density
1	1999 Train Week1 Monday	323832360	1247366 (91.525%)	59679 (4.379%)	1362869	237.61	40242	0.0295
2	1999 Train Week1 Tuesday	325395277	1069409 (92.403%)	15661 (1.353%)	1157328	281.16	6244	0.0054
3	1999 Train Week1 Wednesday	368776477	1465529 (90.649%)	92291 (0.915%)	1616713	228.10	73989	0.0458
4	1999 Train Week1 Thursday	517042040	1719843 (95.174%)	21313 (1.179%)	1807060	286.12	2390	0.0013
5	1999 Train Week1 Friday	284774805	1262782 (93.565%)	15750 (1.167%)	1349635	211.00	2928	0.0022
6	1999 Train Week2 Monday	329322084	1244932 (93.060%)	17029 (1.273%)	1337777	246.17	3158	0.0024
7	1999 Train Week2 Tuesday	375798588	1374855 (94.554%)	16755 (1.152%)	1454035	258.45	3601	0.0025
8	1999 Train Week2 Wednesday	145698730	795287 (89.545%)	11395 (1.283%)	888139	164.05	151	0.0002
9	1999 Train Week2 Thursday	330867682	1325282 (93.816%)	10240 (0.725%)	1412645	234.22	1829	0.0013
10	1999 Train Week2 Friday	273295370	1151616 (91.952%)	14139 (1.129%)	1252412	218.22	5232	0.0042
11	1999 Train Week3 Monday	371123625	1454465 (94.286%)	12768 (0.828%)	1542614	240.58	557	0.0004
12	1999 Train Week3 Tuesday	334280722	1283146 (93.358%)	13747 (1.000%)	1374431	243.21	2512	0.0018
13	1999 Train Week3 Wednesday	540109859	1678816 (95.341%)	9010 (0.512%)	1760859	306.73	207	0.0001
14	1999 Train Week3 Thursday	183158763	1017062 (92.742%)	10559 (0.963%)	1096660	167.02	1358	0.0012
15	1999 Train Week3 Friday	495948059	1471954 (95.784%)	9164 (0.596%)	1536736	322.73	289	0.0002

that are attack-free. It is interesting that *Snort* in fact generates more alarms (column *attacks*) for some 'attack-free' traces than for other malicious traffic streams. For example, the 'attack-free' data set of *1999 Train Week1 Wednesday* triggers 78989 alarms while the attack-inflicted data set of *1999 Train Week2 Wednesday* generates only 151 alerts. Evidently, IDSs/IPSs may generate false positives and thorough tests on the attack detection accuracy of *Snort* have established that false alarms are unavoidable considering the complexity of contemporary intrusions [19]. On the other hand, it has also been shown that the background traffic in the data set is not thoroughly validated and therefore may not be completely attack-free [61]. Attack densities, which is the ratio between number of attacks and total packets, shown in the column *Density* are rather low with the highest value set at 0.046 attacks per packet corroborating the fact that the majority of traffic is legitimate.

The second set of traces we use in our experimentation has been captured by Lawrence Berkeley National Laboratory (LBNL) in 2005 [59] and is depicted in Table 5. The collected data set spans more than 100 hours and records network activity from 8000 internal to *LBNL* and 47000 external hosts [55]. Compared with the TCP-dominated *KDD* set, the *LBNL* traces demonstrate diversified protocol mixture. For instance, the ratio of TCP/UDP in the trace *lbl-internal.20050106-1626.port002.dump.anon* is 97.99/1.59, while in *lbl-internal.20050107-0858.port029.dump.anon*, the ratio reverses to 3.95/95.10. Most *LBNL* traces are much larger than the *KDD* traces; for example, the largest trace *lbl-internal.20050106-1323.port025.dump.anon* has nearly four times the volume of the largest *KDD* set. Packets in the *LBNL* demonstrate a much higher average packet size; for instance, traffic in *lbl-internal.20050107-1225.port022.dump.anon* has an average packet length of 920 bytes in comparison with only 323 bytes in the *KDD* set. The column *Hosts* shows the number of unique IP addresses appearing in respective traces; the host distribution is non-uniform across different

TABLE 5. Statistics on traces in the data set of LBNL enterprise traffic captured in 2005.

ID	Data set name	Size (bytes)	TCP (%)	UDP (%)	Total pkts	Avg pkt	Hosts
1	lbl-internal.20050106-1323.port025.dump.anon	2074090057	2090240 (70.409)	859760 (28.961)	2968708	682	2555
2	lbl-internal.20050106-1423.port026.dump.anon	974274825	1363285 (75.922)	416939 (23.220)	1795635	526	2977
3	lbl-internal.20050106-1626.port002.dump.anon	847491207	2272550 (97.990)	36846 (1.589)	2319177	349	903
4	lbl-internal.20050106-1827.port006.dump.anon	1182342280	2246342 (94.940)	113887 (4.813)	2366053	483	1780
5	lbl-internal.20050107-0255.port023.dump.anon	1274953496	992269 (71.648)	385841 (27.860)	1384913	904	253
6	lbl-internal.20050107-0356.port024.dump.anon	132155826	407601 (81.600)	84956 (17.008)	499511	248	783
7	lbl-internal.20050107-0858.port029.dump.anon	1022085285	125076 (3.948)	3012951 (95.097)	3168280	306	2183
8	lbl-internal.20050107-1225.port022.dump.anon	1106793460	1143243 (96.754)	34793 (2.945)	1181593	920	1243
9	lbl-internal.20050107-1323.port026.dump.anon	1151521843	2111751 (90.226)	212653 (9.086)	2340519	475	3081
10	lbl-internal.20050107-1625.port029.dump.anon	763165383	532123 (16.382)	2677774 (82.438)	3248237	218	1786

LBNL subnets. The *LBNL* set is essentially attack-free and the small number of alerts *Snort* raises are attributed to the artifacts of the traffic collection process, which occasionally drops packets.

4.3. The *KDD* set under *Fingerprinter* and variants

While experimenting with the *KDD* traces of Table 4, we set out to evaluate the performance delivered for the following PMEs:

- Lowmem *method*: the baseline obtained by using the official *Snort* with its default Lowmem PM method and its out-of-the-box configuration.
- *Bloom–Filter PM method*: its default configuration has a fingerprint vector of 4 KB, and its hash function pool consists of four functions, *JS, PJW, SDBM* and *DJB* selected from Table 3.
- *Shingling PM method*: this technique by default employs *JS* as its digesting function and the footprint of its input fingerprint vector is 4 KB.
- *Rabin–Fingerprint PM method*: the Rabin's fingerprint function acts as the digesting function of the method and the fingerprint table contains 4 K entries by default.
- *Hybrid PM method*: the Rabin's fingerprint function is the primary digesting function used and hash function *JS* is built into the Bloom filter. Similar to the *Rabin–Fingerprint PM*, the fingerprint table of the *Hybrid PM* has 4 K entries in its default configuration.

In all of the above, we use a sliding window with size $l = 4$ bytes.

We evaluate the performance of each PME at hand by the following procedure termed *top speed feeding*: (a) each trace of the data set is fed with the tcpreplay utility [62] into the PME n times in a row ($n = 10$) as fast as the test machine can go, and the processing time is recorded; (b) the above step is repeated m times ($m = 10$), and the best processing time achieved within the m iterations is obtained and (c) the average running time on each trace of the data set is computed as its best processing time divided by n. We present the best processing times by PMEs in Fig. 10. Here, the x-axis depicts the trace identifier (that is column '*ID*' of Table 4) and the y-axis shows the processing time in seconds.

Figure 10 clearly shows that the PME processing time is not proportional to the size of the trace. For example, the Lowmem spends about 21.64 and 16.64 seconds, respectively, on the *1999 Train Week1 Thursday* and *1999 Train Week3 Wednesday* traces even though the former is much smaller. For all data sets, the *Hybrid PM* outperforms its counterparts by significant margin, rendering it the best accelerator for intrusion detection. Using the processing time of Lowmem as the baseline, we compute the speedup rates of the *Fingerprinter* and its variants with the results depicted in Fig. 11. The *Fingerprinter* and its variants achieve better performance than Lowmem for the majority of traces with the best rate of 1.75 attained by the *Hybrid PM* on the *1999 Train Week1 Monday* set. The *Rabin–Fingerprint PM* nearly always eclipses both the *Bloom–Filter PM* and the

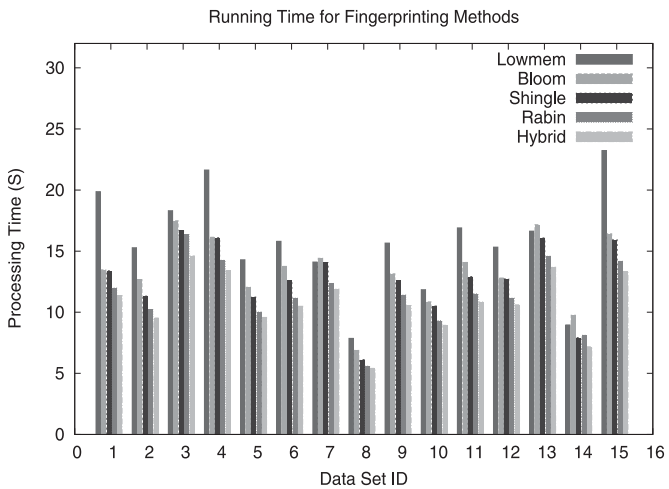

FIGURE 10. Processing times by various PMEs on *KDD* set.

Shingling PM, while the *Shingling PM* fares better than the *Bloom–Filter PM* method. Occasionally the *Bloom–Filter PM* is even slower than Lowmem indicating that it is only suitable for certain types of traffic.

The processing time of Fig. 10 is obtained by feeding PMEs the traces in the *KDD* set with the highest sustainable speed of the test machine, therefore it measures the best performance attained by PMEs. In this regard, the ratio between the size of the trace and its processing time depicted in Fig. 10 can be used to approximate the supported network bandwidth by a PME as outlined in Fig. 12. It can be observed that the highest bandwidth is 394.77 Mbps achieved by the *Hybrid* method on the trace *1999 Train Week3 Wednesday*, while the lowest bandwidth 162.97 Mbps is delivered by Lowmem on the trace *1999 Train Week1 Monday*. The average bandwidth on traces of the *KDD* set accomplished by Lowmem is 219.85 Mbps. In contrast, it is 317.34, 298.31 and 269.51 Mbps, for the *Hybrid*, *Rabin–Fingerprint* and *Shingle*, respectively. Obviously, the *Hybrid PM* could support about 50% more traffic than Lowmem under the same hardware/software setting.

To evaluate the memory consumption of *Fingerprinter* and its variants, we measure the total amount of resident non-swapped physical memory occupied by all PMEs under study through the following *maximum resident memory* procedure: (a) feed each trace to the PME n times in a row ($n = 10$); (b) during the execution of the PME, we measure its memory footprint by sampling once per t seconds ($t = 2$ s—default value) with the help of the profiler top; and (c) once the PME terminates, we compute the maximum memory footprint among all the measurements obtained. Figure 13 depicts the outcomes of the above procedure. One key observation is that the resident memory varies only slightly among different traces; similarly, for the same trace, the memory consumption by various PMEs differs only marginally. For instance, the average memory footprint is 40.07 MB for Lowmem, and it is 41.37, 41.64, 41.66 and 40.48 MB for *Hybrid*, *Rabin–Fingerprint*, *Shingle*, and *Bloom–Filter*, respectively.

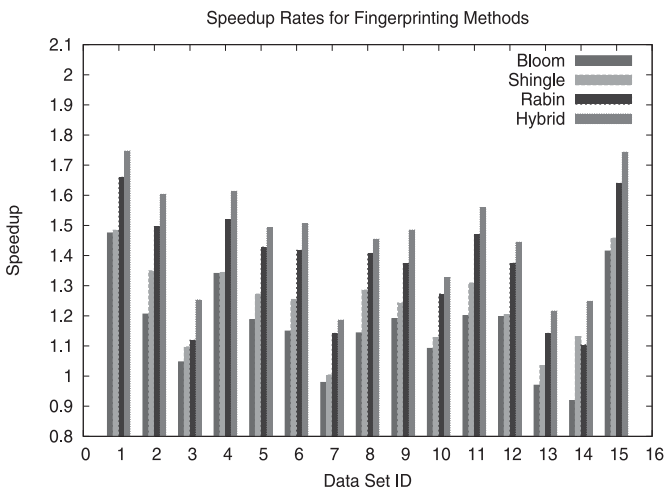

FIGURE 11. Speedup rates of various PMEs on *KDD* set.

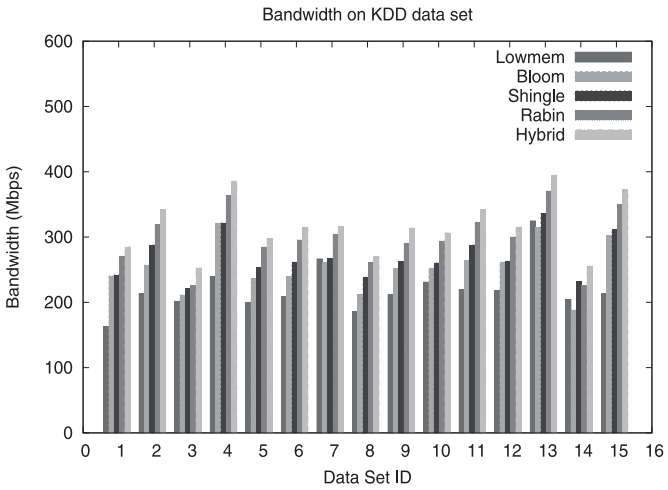

FIGURE 12. Bandwidth supported by PMEs on *KDD* traces.

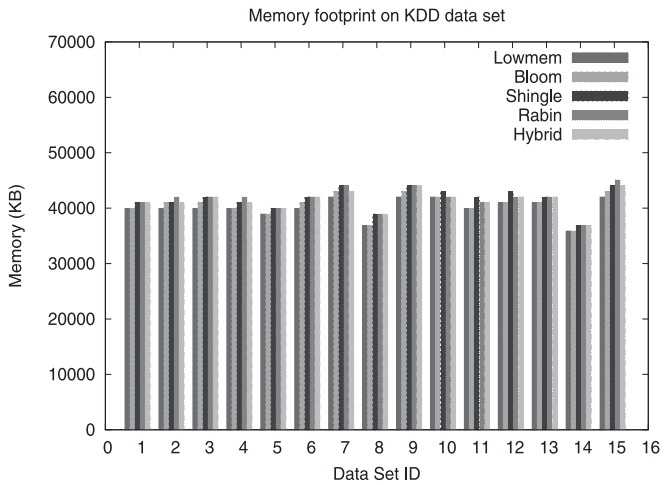

FIGURE 13. Memory footprints required by PMEs on *KDD* traces.

4.4. The *LBNL* set under *Fingerprinter* and variants

The *LBNL* set is released with anonymized IP addresses in the traces and without packet payloads to avoid information leakage and possible privacy infringement [59]. Unfortunately, traces with complete packet contents are absolutely necessary for meaningful IDS/IPS evaluation. We thus generate packet payloads for this set of traces using the following three methods:

- *randomization*: payloads are randomly generated.
- *replacement*: payloads are filled with contents captured from the internal network of *Fortinet* [63] using the testbed discussed in [19]. The captured traffic is cleansed by removing all sessions that are reported as attacks by *Snort*.

- *contamination*: the traces obtained by the *replacement* method are further 'polluted' with attack traffic by randomly selecting a portion of connections and substituting the respective data flows with attacks randomly selected from the exploit database provided by *Fortinet* and containing more than 80000 intrusions such as *Slammer, Nimda* and *Sasser*. The configurable attack density is set to 0.01% by default.

We term the above three generated data sets as *random*, *real* and *attack payload*, respectively.

For the *random LBNL payload*, we evaluate the performance of each PME at hand by following the procedure *top speed feeding* described in Section 4.3. Figure 14 shows the outcome of the experiment with the *random LBNL payloads*. The *Hybrid* consistently outperforms the *Rabin–Fingerprint* approach and in this respect we ascertain that it delivers consistent performance on both *KDD* and *LBNL* sets. In contrast to its relatively poor performance with the *KDD* set, the *Shingling PM* improves its processing times on several traces including *lbl-internal.20050107-0255.port023.dump.anon* and *lbl-internal.20050107-1225.port022.dump.anon*. Our investigation reveals that the distribution of marked bits in the input fingerprint vector of the *Shingling PM* for such *LBNL* traces are much more uniform, reducing the false match rate. It is worth noting that the *Bloom–Filter PM* has a performance inferior to that of the baseline Lowmem approach on numerous *LBNL* traces mainly due to the large average packet sizes that increase its false match rate.

Figures 15 and 16 show the processing times of various PMEs with the *real* and *attack payload*. Obviously, the behavior of PMEs on traces with *real* payloads is quite different from that with *randomized* content. For instance, Lowmem spends about 41 s on randomly filled trace *lbl-internal.20050106-1323.port025.dump.anon* but consumes more than 60 s if the trace is packed with real-world content. The comparison between Figs 15 and 16 clearly indicates that PMEs demonstrate similar processing times on traces with *real* and *attack payloads*, implying that PMEs are only marginally affected by the relatively light attack density. The average running time by the Lowmem, *Bloom–Filter*, *Shingle*, *Rabin–Fingerprint*, and *Hybrid* on traces of the data set *random payload* can be computed as 26.59, 30.93, 23.29, 23.55 and 21.87 s, respectively; in contrast, it is 34.78, 39.36, 25.39, 24.52 and 22.50 s for the *attack payload* set. Clearly, traces with real-world payloads consume much longer processing times than traces with synthetic content due to the fact that randomized payloads are less likely to share common patterns with attack signatures.

The speedup rates attained by various PMEs with respect to the baseline Lowmem method on the *LBNL* set using

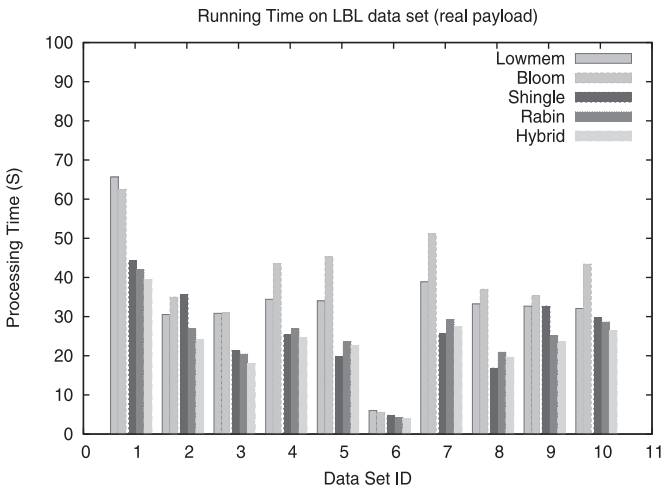

FIGURE 15. Processing times of various PMEs on *LBNL* set of Table 5 with *real payload*.

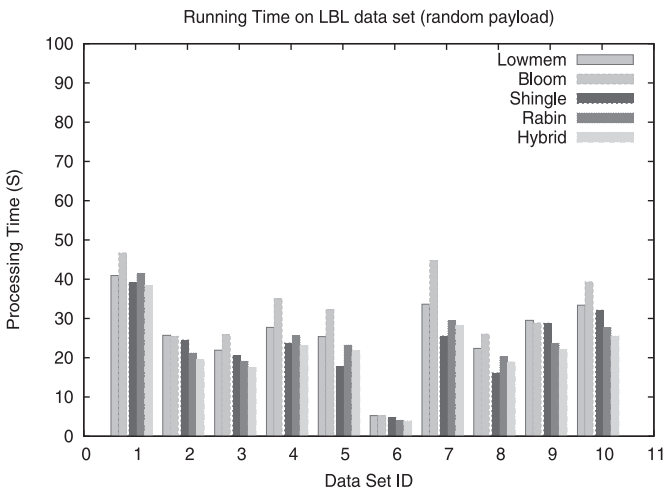

FIGURE 14. Processing times of various PMEs on *LBNL* set of Table 5 with *random payload*.

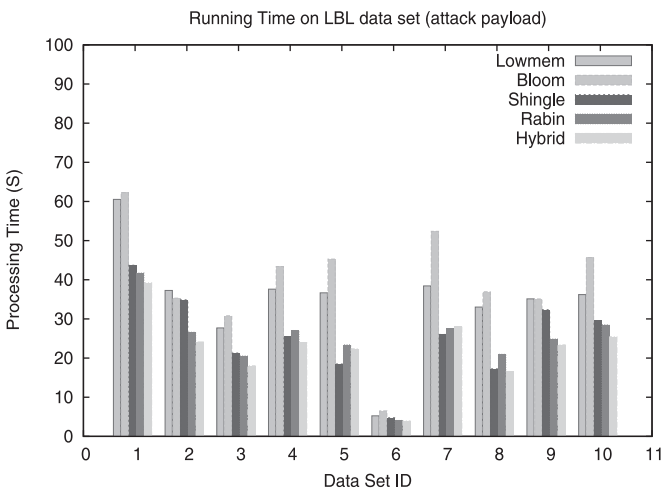

FIGURE 16. Processing times by various PMEs on *LBNL* set of Table 5 with *attack payload*.

attack payload are furnished in Fig. 17. The best speedup rate of 2.00 is delivered by the *Hybrid PM* on the trace *lbl-internal.20050107-1225.port022.dump.anon* coming much higher than the corresponding 1.75 best rate achieved on the *KDD* set. The average speedup rates for the *LBNL* set with *attack payload* are 1.55, 1.41, 1.40 and 0.88 for *Hybrid*, *Rabin–Fingerprint*, *Shingle*and *Bloom–Filter*, respectively. Apparently, the *Bloom–Filter* engine does not help the intrusion detection process much. The standard deviation values on the speedup rates indicate that the *Rabin–Fingerprint PM* behaves more consistently compared with the *Shingling PM* (that is standard deviation 0.10 vs 0.31). Despite the fact that the *KDD* and *LBNL* sets have been captured in different time periods and network environments, they both help establish the efficiency of the *Fingerprinter*.

As pointed out previously, the processing time of Fig. 16 is measured by replaying traces in the *LBNL* set with *attack payload* at the highest speed supported by the test machine. Thus, it can be considered as the best performance achieved by PMEs. To this end, we estimate the sustained network bandwidth on a trace by a PME to be the ratio between the size of the trace and its processing time depicted in Figure 16. The results are shown in Figure 18. The significant fluctuation of bandwidth among various traces does indicate the dependency of bandwidth delivered by PMEs on various types and protocol mixtures of traffic. The average bandwidth on traces of the *LBNL* set with *attack payload* delivered by the Lowmem method is 296.34 Mbps. In comparison, it is 462.90, 421.34 and 423.10 Mbps, for the *Hybrid*, *Rabin–Fingerprint* and *Shingle*, respectively. Obviously, the *Hybrid PM* is able to yield about 50% more bandwidth than Lowmem under the same hardware/software setting.

The memory consumption of *Fingerprinter* and its variants can be measured with the help of the *maximum resident*

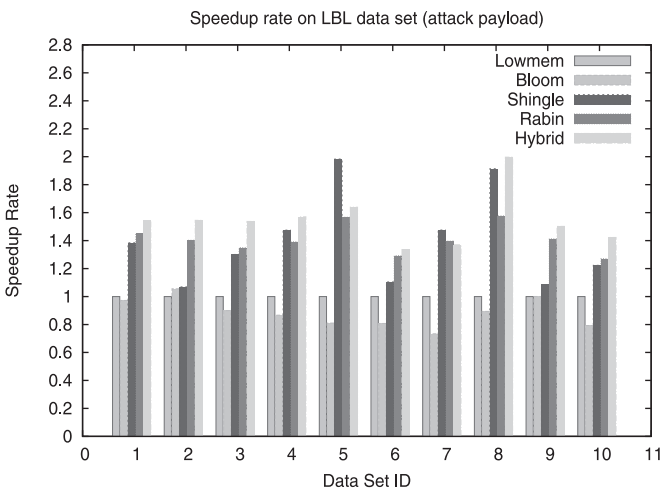

FIGURE 17. Speedup rates by various PMEs on *LBNL* set of Table 5 with *attack payload*.

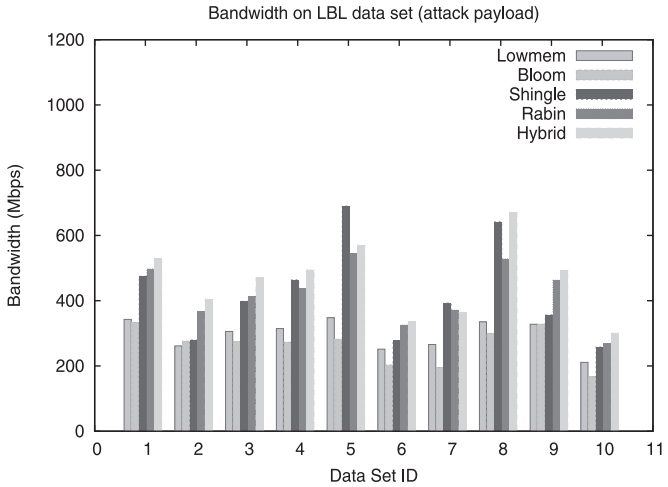

FIGURE 18. Bandwidth supported by various PMEs on *LBNL* data set of Table 5 with *attack payload*.

memory procedure outlined in Section 4.3. Figure 19 depicts the outcomes of the above procedure. Similar to its *KDD* counterpart, traces in the *LBNL* data set have comparable memory footprint; meanwhile, for the same trace, the memory consumption by various PMEs differs only slightly. For example, the average memory footprint is 38.42 MB for Lowmem, while it is 40.67, 40.50, 40.66 and 38.10 MB for *Hybrid*, *Rabin–Fingerprint*, *Shingle* and *Bloom–Filter*, respectively.

4.5. Performance of the *Bloom–Filter PM* method

The key parameters in the *Bloom–Filter PM* are the size of its hash function pool and the footprint of its fingerprint vector. To evaluate the impact on the speedup rate by the size of the hash function pool in the *Bloom–Filter PM*, we

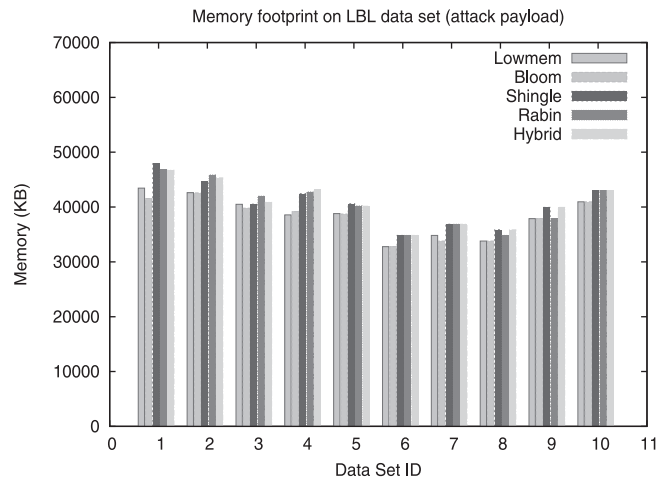

FIGURE 19. Memory footprints required by various PMEs on *LBNL* data set of Table 5 with *attack payload*.

design a sequence of experiments by configuring the *Snort* to employ the *Bloom–Filter PM* as its PME, while the fingerprint vector of the *Bloom–Filter PM* is 2 KB and the trace to be processed is *1999 Train Week1 Monday*. In each different experiment setting, we construct a new set of hash functions with size $k \in [2, 8]$ and its elements are extracted from the set $H = \{\text{JS, PJW, SDBM, DJB, DEK, RS, BKDR, AP}\}$, constructed from Table 3, in the presented order. For instance, when $k = 2$, the hash function pool is {JS, PJW}, and it is {JS, PJW, SDBM} when $k = 3$. We repeat the experiment with the same configuration $n = 10$ times and record the best processing time we can obtain.

By using the running time of the Lowmem method as the baseline, we compute the speedup rates of the above described experiment settings and present the results in Fig. 20; here, the x-axis depicts the size of the hash function pool while the y-axis represents the speedup rate. In Fig. 20, we also depict the false match rates generated by the *Bloom–Filter PM* under different settings; in this respect, the y-axis represents the ratio between the number of false matches created by the experiment setting in question and that by the *Bloom–Filter PM* with hash function pool of {JS, PJW}. The non-linear relationship between the speedup rate and the size of the hash function pool can be easily observed from Fig. 20, and the highest speedup rate is achieved when three hash functions are used. It is also noticeable that the speedup rate deteriorates monotonically as the size of the hash function pool increases beyond 3, indicating that the computational intensity in the hash code calculations become significant when $k > 3$. In contrast, the false match ratio decreases steadily when the size of the hash function pool increases, implying that extra hash functions in the pool can improve the Bloom filter's differentiation capability.

We also conduct a series of experiments to investigate the relationship between the speedup rate and the size of the

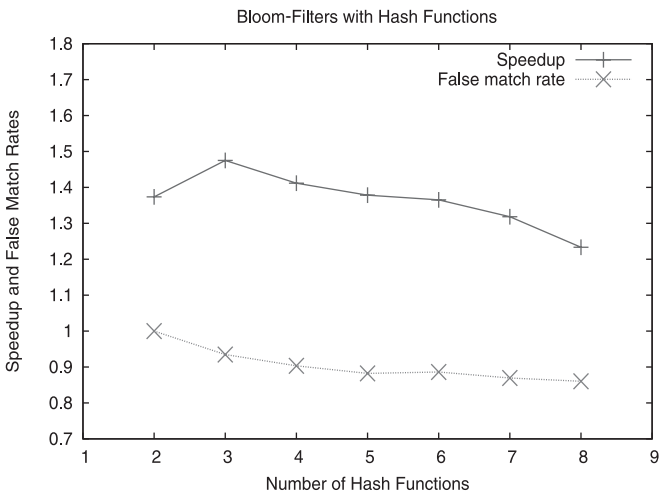

FIGURE 20. Speedup, number of hash functions and false match rate in *Bloom–Filter PM*.

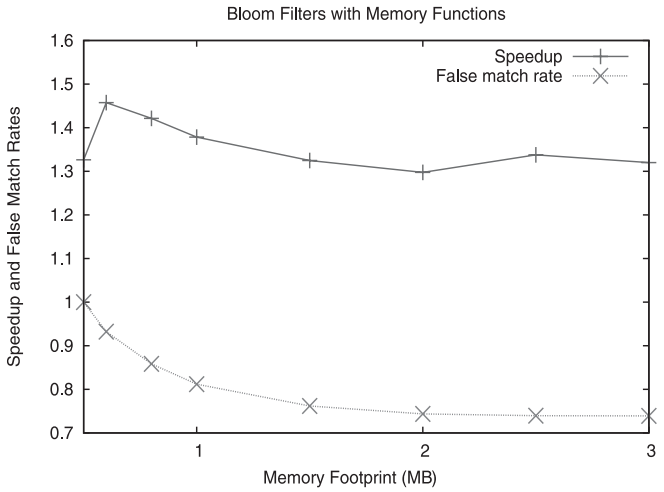

FIGURE 21. Speedup, memory footprint and false match rate in *Bloom–Filter PM*.

fingerprint vector in the *Bloom–Filter PM* method. We control the memory footprint m of the fingerprint vector so that the ratio (m/n) varies in the range of [0.5, 3] (in bytes); here, n is the number of signatures in the *Snort*. In all settings, we fix the hash function pool to be {JS, PJW, SDBM, DJB} and the *1999 Train Week1 Monday* trace is used, so that the performance of the *Bloom–Filter PM* is only affected by (m/n). The speedup rates attained by the above described experiments are presented in Fig. 21 along with the false match rates. It is interesting to observe that increases in the footprints of the fingerprint vector do not necessarily improve the speedup rate, implying that memory-related aspects such as the locality of memory access, page faults/swaps and memory management, all affect the speedup rate. The false match rate gradually decreases as more memory is allocated to the fingerprint vector, providing a much larger hash code space and, consequently, reducing the fingerprint collision rate.

4.6. Evaluation of the *Rabin–Fingerprint PM* method

The performance of the *Rabin–Fingerprint PM* method is heavily affected by the size of its fingerprint table as the latter determines the fingerprint collision probability. The relationship between the number of entries M in the fingerprint table and the size of the sliding window l can be expressed as $M = 2^{(l+8)}$. Consequently, we expect that the speedup rate attained by *Rabin–Fingerprint PM* can be improved by increasing the size of the sliding window l. To investigate the relationship between speedup rate and memory consumption, we configure the *Snort* to use *Rabin–Fingerprint PM* method as its PME and work with the traces of Table 4. By varying the size of the sliding window l in the range of [3, 6] bytes to control the footprint of the fingerprint table, we define a family of experiments termed *Rabin-l*; for example, *Rabin-3* refers to the *Rabin–Fingerprint PM* with a 3-byte sliding window. Figure 22 presents the

speedup rates obtained by the above experiments. Clearly, rates on the same trace can be steadily improved by increasing the size of the sliding window. We also note that the *Rabin-4* curve comes much closer to that of *Rabin-5* instead of *Rabin-3* for most data sets, indicating that a more generous improvement can be attained by changing the window from 3 to 4 than from 4 to 5. Similarly, the relatively narrow gap between the curves *Rabin-5* and *Rabin-6* points out that only insignificant gains are feasible by increasing l beyond 5 bytes.

The formula ($M = 2^{(l+8)}$) clearly indicates that the memory consumption by the *Rabin–Fingerprint PM* increases exponentially with the sliding window size. However, the improvement on the speedup rate is approximately linear to the sliding window size as Fig. 23 demonstrates. For instance, when we change the sliding window sizes in the range of [3, 6] bytes and employ the *Rabin–Fingerprint PM* to process the trace *1999 Train Week3 Friday*, the speedup rates obtained are 1.43, 1.64, 1.81 and 1.94, respectively, which clearly form a straight line in Fig. 23. Similar observations can also be drawn from other traces. In contrast, the fingerprint table contains 1, 2, 4 and 8 K entries when the sliding window sizes range between 3 and 6 bytes. Obviously, a trade-off exists between the speedup rate and memory consumption in the *Rabin–Fingerprint PM* method: the intrusion detection process could be improved with a large fingerprint table that reduces fingerprint collision rate at the cost of exponentially increasing memory footprint.

4.7. Impact of *Hybrid PM* on intrusion detection

The full-fledged *Fingerprinter* that employs the *Hybrid PM* method integrates multiple fingerprinting techniques to improve the performance of the PME in IDSs/IPSs. The *Hybrid PM* actually performs three tasks on each shingle of the input in its *FM* process: (1) the Rabin's fingerprint of the input shingle acts as the access key to the fingerprint table. The shingle is declared benign and no further inspection is required unless it

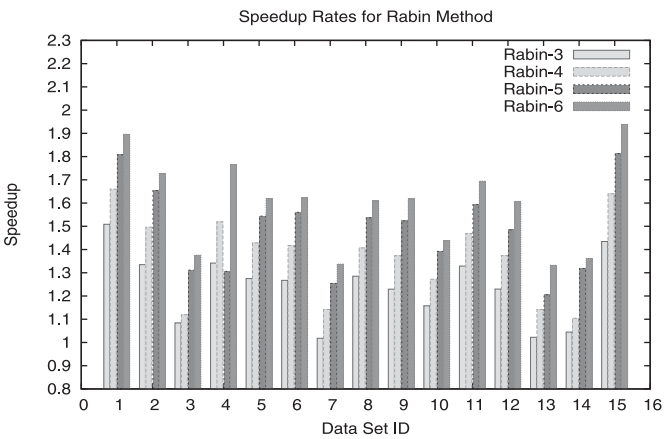

FIGURE 22. Relationship between speedup and sliding window size in *Rabin–Fingerprint PM*.

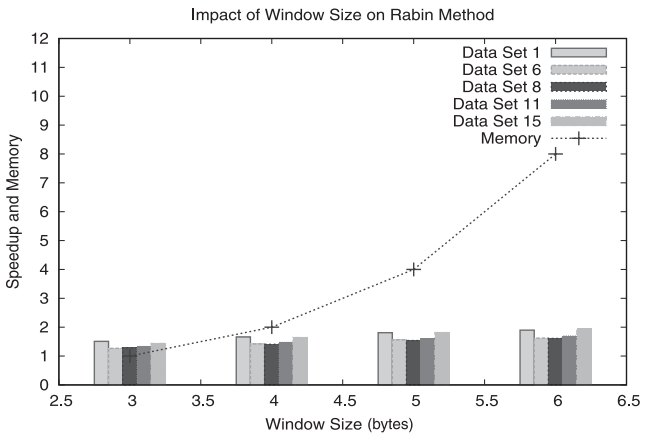

FIGURE 23. Relationships between speedup, memory footprint and sliding window size in *Rabin–Fingerprint PM*.

hits a non-empty entry of the fingerprint table; (2) in case that a non-empty entry in the fingerprint table is landed, k hash codes are generated for the input shingle and compared with those of the signature in the hitting entry. The shingle is declared attack-free as long as a mismatch occurs in the comparison and (3) the input stream starting at the current sliding window is further matched against the longest pattern of the signature to ensure that the latter is indeed contained by the input. It is expected that input shingles from legitimate traffic rarely pass the above-described inspections, helping improve the system performance.

By defining the screening rate of a fingerprinting method as the ratio between the number of negative verdicts it generates for shingles over the total shingles it processes, we can evaluate the differentiation capabilities of the fingerprinting method in question. With the definition of the screening rate at hand, we can evaluate the contributions on the system performance by the above-described three tasks conducted by the *Hybrid PM* method. We first configure the *Hybrid PM* method to use a hash function pool with a single element *DJB* to process the data sets in Table 4. We then repeat the experiments by replacing the hash function pool of the *Hybrid PM* method with the two-element set {*DJB, SDBM*}. For comparison, we also conduct an experiment with the *Rabin–Fingerprint PM* method as *Snort's* PME. The resulting speedup rates are presented in Fig. 24.

It is clear from Fig. 24 that *Hybrid PM* with one or two hash functions outperforms the *Rabin–Fingerprint PM* method, demonstrating the advantage of multiple fingerprinting strategies. The fact that the *Hybrid PM* with a single hash function actually attains better performance than the *Hybrid PM* with two hash functions indicates that the computation intensity resulted from two hash functions is significant and that it occasionally may deteriorate the overall performance. The contributions by the two hash functions can also be evaluated based on their screening rates presented in Fig. 25. The screening rate achieved by the first hash function is typically

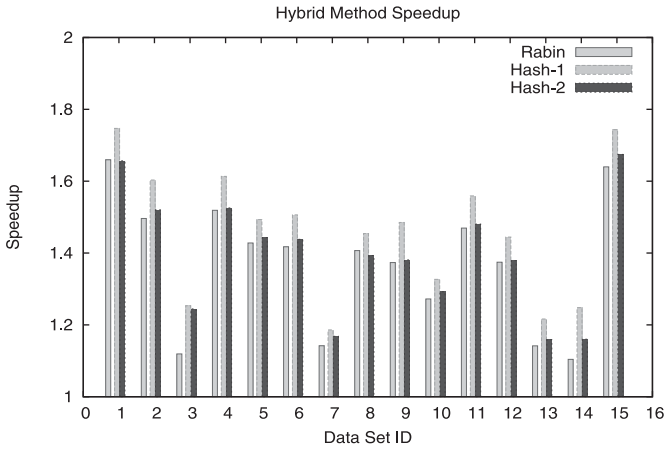

FIGURE 24. Relation between speedup and hash function pool in *Hybrid PM*.

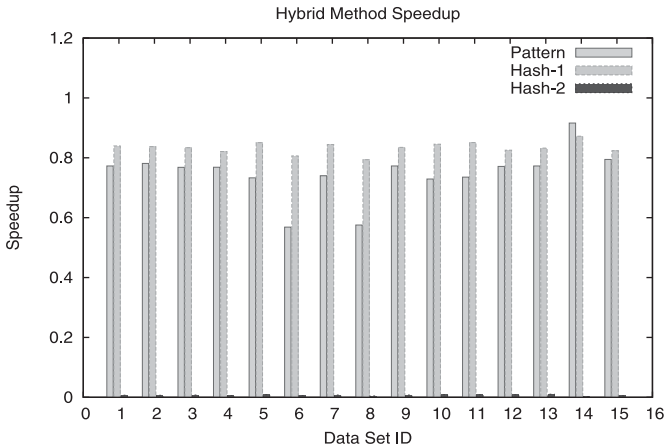

FIGURE 25. Screening capabilities of hash functions and string comparison in *Hybrid PM*.

around 0.85, rendering its strong capability to differentiate legitimate from malicious traffic. Similarly, the exact PM operation by the third task described above also attains a high screening rate, implying that most input shingles leading to FMs are in fact false matches. On the contrary, the screening rate of the second hash function is extremely low, signifying that shingles passing the examination of the first hash function are in fact prefixes of telltale patterns in signatures.

5. CONCLUSIONS AND FUTURE WORK

To protect intranets and computer systems from being compromised, IDSs/IPSs employ PM techniques to identify intrusions often with the help of an attack signature database. By matching the incoming streams against each signature with exact PM algorithms–such as Boyer–Moore and Aho–Corasick-an IDS/IPS generates a positive verdict if a match occurs and a negative verdict otherwise. Clearly, a positive verdict can be delivered by scanning on average half of the signatures while a negative one necessitates the involvement of the entire signature database. The latter is obviously more computationally intensive and consequently legitimate traffic gets heavily penalized.

In this paper, we propose the *Fingerprinter* whose aim is to accelerate the attack identification process of IDSs/IPSs based on the observations that the vast majority of the Internet traffic is legitimate and telltale patterns in signatures are often only unique to attacks. The *Fingerprinter* integrates fingerprinting and PM techniques to generate negative verdicts very quickly for attack-free streams. At first, the framework develops a concise and compact fingerprint for each attack signature. Then, it transforms the incoming traffic into the fingerprint space and matches its digest against those derived from the signatures. Traffic is exploit-free if no FM exists. Otherwise, *Fingerprinter* resorts to the Boyer–Moore method to ascertain that the input indeed satisfies conditions specified in the signatures with matching fingerprints. We combine multiple fingerprinting approaches such as *Bloom–Filter* and *Rabin–Fingerprint* in order to reduce false matches that occur when the input shares the same fingerprints with signatures but fails to match the exact patterns specified by the signatures.

We have implemented the *Fingerprinter* as a PME in the open-source IDS/IPS *Snort* and experimentally evaluated with a number of traces. The modularized design of *Fingerprinter* enables component activation in a plug-and-play manner. In this way, a variety of PMEs with diversified fingerprinting methods can be readily derived and their contributions to the overall system performance can be quantified. Our experiments clearly demonstrate that the *Bloom–Filter* method alone can rapidly identify legitimate traffic only for certain types of network streams. We also establish that the *Shingling PM* typically outperforms the baseline–Lowmem in the official *Snort*–by quickly pinpointing candidate signatures with the help of the specifically maintained associations between attack signatures and their fingerprints. The intrusion detection process can be further accelerated through the *Rabin–Fingerprint* method as the computation of fingerprints can be performed in a rolling manner. Compared with the performance delivered by the Lowmem method in *Snort*, our purely software-based *Fingerprinter* attains an up to 2-fold speedup for IDS benchmark traffic in the *KDD* data set as well as *LBNL* traces with the help of integrated fingerprinting and PM techniques.

In the future, we aim at pursuing a number of different directions: first, we intend to automate the synchronization between our framework and the releases of the *Snort* code and signature database, so that *Fingerprinter* remains updated with developments in the field. Second, we plan to develop more advanced fingerprinting methods that may use pools of updatable Bloom filters and coordinate their actions through a multi-threading paradigm so that attack signatures can be added or deleted dynamically. It is also part of our ongoing work to examine the viability of applying the summarization methods

developed for multimedia and digital documents to IDSs/IPSs. Finally, we wish to implement the *Fingerprinter* in hardware such as FPGA or ASIC and empower it with regular expression functionalities to accommodate complex signatures.

ACKNOWLEDGEMENT

We are grateful for the comments of the anonymous reviewers that helped us significantly improve the presentation of our work. We are also indebted to Peter Wei and Gary Duan for providing valuable comments on key aspects of the *Fingerprinter* as well as Fushen Chen and Ping Wu for their help while we evaluated the *Fingerprinter*.

FUNDING

This work was partially supported by a European Social Funds and National Resources Pythagoras Grant and the University of Athens Research Foundation.

REFERENCES

[1] Bos, H. and Huang, K. (2006) Towards software-based signature detection for intrusion prevention on the network card. *Lect. Notes Comput. Sci.*, Seattle, WA, USA, **3858**, 102–123.

[2] Roesch, M. (1999) Snort–Lightweight Intrusion Detection for Networks. *Proc. 13th USENIX Conf. Systems Administration–LISA'99*, Seattle, Washington, pp. 229–238. USENIX Association, Berkeley, CA, USA.

[3] Tuck, N., Sherwood, T., Calder, B. and Varghese, G. (2004) Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. *Proc. IEEE Infocom and Twenty-Third Annual Joint Conf. IEEE Computer and Communications Societies*, Hong Kong, March, pp. 2628–2639. IEEE.

[4] Handley, M., Paxson, V. and Kreibich, C. (2001) Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. *Proc. USENIX Security Symp.*, Washington, DC, USA, January, pp. 115–134.

[5] Browne, H.K., Arbaugh, W.A., McHugh, J. and Fithen, W.L. (2001) A trend analysis of exploitations. *Proc. 2001 IEEE Symp. Security and Privacy*, pp. 214–231.

[6] MITRE Organization (2008). *Common vulnerabilities and exposures*. http://cve.mitre.org/. (Last accessed April 6, 2009).

[7] Arora, A., Krishnan, R., Nandkumar, A., Telang, R. and Yang, Y. (2004) Impact of Vulnerability Disclosure and Patch Availability—An Empirical Analysis. *Proc. Workshop on the Economics of Information Security* (*WEIS*), Minneapolis, MN, pp. 1–20.

[8] Coit, C.J., Staniford, S. and McAlerney, J. (2001) Towards Faster Pattern Matching for Intrusion Detection, or Exceeding the Speed of Snort. *Proc. 2nd DARPA Information Survivability Conf. Exposition* (*DISCEX II*), Anaheim, CA, USA, June, pp. 367–373.

[9] Fisk, M. and Varghese, G. (2002) An Analysis of Fast String Matching Applied to Content-Based Forwarding and Intrusion Detection. Technical Report, CS2001-0670 (updated version). University of California, San Diego.

[10] Tan, L. and Sherwood, T. (2005) A High Throughput String Matching Architecture for Intrusion Detection and Prevention. *Proc. 32nd Annual Int. Symp. Computer Architecture*, Washington, DC, USA, June, pp. 112–122. IEEE Computer Society.

[11] Aho, A. and Corasick, M. (1975) Fast pattern matching: an aid to bibliographic search. *Commun. ACM*, **18**, 333–340.

[12] Boyer, R. and Moore, J. (1977) A fast string searching algorithm. *Communi. ACM*, **20**, 762–772.

[13] Gusfield, D. (1997) *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. University of California Press.

[14] Wu, S. and Manber, U. (1994) A Fast Algorithm for Multi-Pattern Searching. Technical Report TR-94-17, University of Arizona.

[15] Walter, B.C. (1979) A String Matching Algorithm Fast on Average. *Proc. 6th Int. Colloquium On Automata, Languages, and Programming*, London, UK, pp. 118–132. Springer.

[16] Song, H., Sproull, T., Attig, M. and Lockwood, J. (2005) Snort Offloader: A Reconfigurable Hardware NIDS Filter. *Proc. 15th Int. Conf. Field Programmable Logic and Applications* (*FPL*), Tampere, Finland, August, pp. 493–498.

[17] Anagnostakis, K.G., Markatos, E.P., Antonatos, S., and Polychronakis, M. (2003) E2xB: A Domain-Specific String Matching Algorithm for Intrusion Detection. *Proc. 18th IFIP Int. Information Security Conf.* (*SEC2003*), May, pp. 217–228. Kluwer.

[18] Ramaswamy, R., Kencl, L. and Iannaccone, G. (2006) Approximate Fingerprinting to Accelerate Pattern Matching. *Proc. 6th ACM SIGCOMM on Internet Measurement Conf.* (*IMC'06*), Rio de Janeiro, Brazil, October, pp. 301–306. ACM Press.

[19] Chen, Z., Delis, A. and Wei, P. (2008) A pragmatic methodology for testing intrusion prevention systems. *Comput. J.*, online, 1–30.

[20] Attig, M., Dharmapurikar, S. and Lockwood, J. (2004) Implementation Results of Bloom Filters for String Matching. *Proc. 12th Annual IEEE Symp. Field Programmable Custom Computing Machines* (*FCCM*), Napa, CA, April, pp. 322–323. IEEE.

[21] Rivest, R.L. (1977) On the worst-case behavior of string-searching algorithms. *SIAM J. Comput.*, **6**, 669–674.

[22] Apostolico, A. and Giancarlo, R. (1986) The Boyer–Moore–Galil string searching strategies revisited. *SIAM J. Comput.*, **15**, 98–105.

[23] Stephen, D.L. (1994) String searching algorithms. *Lect. Notes Series Comput.*, **3**.

[24] Crochemore, M., Hancart, C. and Lecroq, T. (2003) A unifying look at the Apostolico–Giancarlo string matching algorithm. *J. Discrete Algorithms*, **1**, 37–52.

[25] Knuth, D., Morris, J. and Pratt, V. (1977) Fast pattern matching in strings. *SIAM J. Comput.*, **6**, 323–350.

[26] Galil, Z. (1979) On improving the worst case running time of the Boyer–Moore string searching algorithm. *Commun. ACM*, **22**, 505–508.

[27] Horspool, R. (1980) Practical fast searching in strings. *Softw.—Pract. Exp.*, **10**, 501–506.

[28] Kruegel, C., Valeur, F., Vigna, G. and Kemmerer, R. (2002) Stateful Intrusion Detection for High-Speed Networks. *Proc. 2002 IEEE Symp. Security and Privacy*, Washington, DC, USA, May, pp. 285–294. IEEE Computer Society.

[29] Hutchings, B., Franklin, R. and Caraver, D. (2002) Assisting Network Intrusion Detection with Reconfigurable Hardware. *Proc. 10th Annual IEEE Symp. Field Programmable Custom Computing Machines* (*FCCM*), Napa, CA, April, pp. 111–120. IEEE.

[30] Schuehler, D.V., Moscola, J. and Lockwood, J.W. (2003) Architecture for a Hardware Based TCP/IP Content Scanning System. *Proc. 11th Symp. High Performance Interconnects*, Stanford, CA, August, pp. 89–94.

[31] Sugawara, Y., Inaba, M. and Hiraki, K. (2004) Over 10 Gbps String Matching Mechanism for Multi-Stream Packet Scanning Systems. *Proc. 14th Int. Conf. Field Programmable Logic and Application* (*FPL*), Antwerp, Belgium, August, pp. 484–493. Springer.

[32] Liu, R., Huang, N., Chen, C. and Kao, C. (2004) A fast string-matching algorithm for network processor-based intrusion detection system. *ACM Trans. Embedded Comput. Syst.*, **3**, 614–633.

[33] Moscola, J., Lockwood, J., Loui, R.P. and Pachos, M. (2003) Implementation of a Content-Scanning Module for an Internet Firewall. *Proc. 11th IEEE Symp. Field-Programmable Custom Computing Machines* (*FCCM'03*), Napa, CA, April, pp. 31–38. IEEE.

[34] Clark, C. and Schimmel, D. (2003) Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. *Proc. Eleventh ACM/SIGDA Int. Conf. Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, September. ACM.

[35] Clark, C. and Schimmel, D. (2004) Scalable Parallel Pattern Matching on High Speed Networks. *Proc. Twelfth Annual IEEE Symp. Field Programmable Custom Computing Machines* (*FCCM'04*), Napa, CA, April. IEEE.

[36] Cho, Y. and Mangione-Smith, W. (2004) Deep Packet Filter with Dedicated Logic and Read Only Memories. *Proc. IEEE Symp. Field Programmable Custom Computing Machines*, Napa, CA, April. IEEE.

[37] Baker, Z.K. and Prasanna, V.K. (2004) A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs. *Proc. IEEE Symp. Field Programmable Custom Computing Machines* (*FCCM*), Napa, CA, April, pp. 135–144. IEEE.

[38] Gokhale, M., Dubois, D., Dubois, A., Boorman, M., Poole, S. and Hogsett, V. (2003) Granidt: towards gigabit rate network intrusion detection. *Proc. Eleventh Annual ACM/SIGDA Int. Conf. Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, September. ACM.

[39] Dharmapurikar, S., Krishnamurthy, P., Sproull, T. and Lockwood, J. (2003) Implementation of a Deep Packet Inspection Circuit using Parallel Bloom Filters in Reconfigurable Hardware. *Proc. Eleventh Annual IEEE Symp. on High Performance Interconnects* (*Hoti'03*), Stanford, CA, August, pp. 44–51. IEEE.

[40] Cho, Y. and Mangione-Smith, W. (2005) Fast Reconfiguring Deep Packet Filter for 1+ Gigabit Network. *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (*FCCM*), Napa, CA, April.

[41] Baker, Z.K. and Prasanna, V.K. (2004) Time and Area Efficient Pattern Matching on FPGAs. *Proc. ACM/SIGDA 12th Int. Symp. Field Programmable Gate Arrays*, pp. 223–232. ACM.

[42] Baker, Z.K. and Prasanna, V.K. (2005) High-Throughput Linked-Pattern Matching for Intrusion Detection Systems. *Proc. ANCS'05*, Princeton, NJ, USA, October, pp. 193–202. ACM.

[43] Song, H., Dharmapurikar, S., Turner, J. and Lockwood, J. W. (2005) Fast hash table lookup using extended bloom filters: an aid to network processing. *ACM SIGCOMM Comput. Commun. Rev.*, **35**, 181–192.

[44] Bloom, B.H. (1970) Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, **13**, 422–426.

[45] Manber, U. (1994) Finding Similar Files in a Large File System. *Proc. 1994 USENIX Winter Technical Conf.*, San Fransisco, CA, USA, pp. 1–10.

[46] Heintze, N. (1996) Scalable Document Fingerprinting. *Proc. Second USENIX Workshop on Electronic Commerce*, Oakland, CA, November, pp. 191–200.

[47] Fetterly, D., Manasse, M. and Najork, M. (2003) On the Evolution of clusters of Near-Duplicate Web Pages. *Proc. 1st Latin American Web Cong.*, Santiago, Chile, November, pp. 37–45. IEEE Computer Society.

[48] Rabin, M. (1981) Fingerprinting by Random Polynomials. Technical Report TR-15-81. Department of Computer Science, Harvard University.

[49] Pankanti, S., Prabhakar, S. and Jain, A. (2002) On the Individuality of Fingerprints. *IEEE Trans. Pattern Anal. Mach. Intell.*, **24**, 1010–1025.

[50] Feistel, H. (1973) Cryptography and computer privacy. *Scientific American*, **228**, 15–23.

[51] Damgaard, I.B. (1989) A Design Principle for Hash Functions advances in Cryptology. *Proc. CRYPTO89, Lecture Notes in Computer Science,* 435, pp. 416–442.

[52] Caceres, R., Danzig, P., Jamin, S. and Mitzel, D. (1991) Characteristics of Wide-Area TCP/IP Conversations. *Proc. Conf. Communications Architecture and Protocols*, Zurich, Switzerland, pp. 101–112. ACM.

[53] Aiello, W., Kalmanek, C., McDaniel, P., Sen, S., Spatscheck, O. and van der Merwe, K. (2005) Analysis of Communities Of Interest in Data Networks. *Proc. Passive and Active Measurement Workshop*, Boston, MA, March, pp. 1–14.

[54] Tan, G., Poletto, M., Guttag, J. and Kaashoek, F. (2003) Role Classification of Hosts within Enterprise Networks Based on Connection Patterns. *Proc. USENIX Annual Technical Conf.*, San Antonio, TX, June, pp. 306–316.

[55] Pang, R., Allman, M., Bennett, M., Lee, J., Paxson, V. and Tierney, B. (2005) A First Look at Modern Enterprise Traffic. *Proc. ACM Internet Measurement Conf.* (*IMC*), October, pp. 15–28. ACM.

[56] Fowler, H.J. and Leland, W.E. (1991) Local area network traffic characteristics, with implications for broadband network congestion management. *IEEE J. Selected Areas Commun.*, **SAC**, 1139–1149.

[57] Paxson, V. (1994) Growth trends in wide-area TCP connections. *IEEE Netw.*, **8**, 8–17.

[58] Sen, S. and Wang, J. (2002) Analyzing Peer-to-Peer Traffic Across Large Networks. *Proc. Internet Measurement Workshop*, Nov., pp. 137–150.

[59] Lawrence Berkeley National Laboratory (2005). *LBNL Enterprise trace repository*. http://www.icir.org/enterprise-tracing. (Last accessed April 6, 2009).

[60] MIT Lincoln Laboratory (2002). *DARPA intrusion detection evaluation data sets*. http://www.ll.mit.edu/IST/ideval/data/data_index.html. (Last accessed April 6, 2009).

[61] Mchugh, J. (2000) Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln laboratory. *ACM Trans. Information and System Security*, **3**, 262–294.

[62] Sourceforge (2009). *Tcpreplay: a suite of tools to edit and replay captured network traffic*. http://sourceforge.net/projects/tcpreplay. (Last accessed April 6, 2009).

[63] Fortinet Inc. (2009). *FortiGate: an anti-virus and intrusion prevention system*. http://www.fortinet.com. (Last accessed April 6, 2009).