IEICE TRANS. INF. & SYST., VOL.E99–D, NO.8 AUGUST 2016

PAPER

Grammar-Driven Workload Generation for Efficient Evaluation of Signature-Based Network Intrusion Detection Systems

Min SHAO[†], **Min S. KIM**[†], **Victor C. VALGENTI**[†], ***Nonmembers***, ***and*** **Jungkeun PARK**[††a)], ***Member***

SUMMARY Network Intrusion Detection Systems (NIDS) are deployed to protect computer networks from malicious attacks. Proper evaluation of NIDS requires more scrutiny than the evaluation for general network appliances. This evaluation is commonly performed by sending pre-generated traffic through the NIDS. Unfortunately, this technique is often limited in diversity resulting in evaluations incapable of examining the complex data structures employed by NIDS. More sophisticated methods that generate workload directly from NIDS rules consume excessive resources and are incapable of running in real-time. This work proposes a novel approach to real-time workload generation for NIDS evaluation to improve evaluation diversity while maintaining much higher throughput. This work proposes a generative grammar which represents an optimized version of a context-free grammar derived from the set of strings matching to the given NIDS rule database. The grammar is memory-efficient and computationally light when generating workload. Experiments demonstrate that grammar-generated workloads exert an order of magnitude more effort on the target NIDS. Even better, this improved diversity comes at much smaller cost in memory and speeds four times faster than current approaches.

key words: *workload generation, intrusion detection*

1. Introduction

Network security threats have continued to rise over the past decade with more than 70% of reported vulnerabilities remotely exploitable in 2014 [1]. Network Intrusion Detection Systems (NIDS) guard against such security threats by examining network packets entering a protected network. One of the major category of NIDS, signature-based NIDS, identifies attacks through comparison of the network packets to known attack patterns (also termed signatures). Given precise patterns describing attacks, such "signature-based intrusion detection" provides reliable results with few false positives. (For convenience all references to NIDS in this work imply signature-based NIDS.) However, large signature databases combined with high-speed traffic make comparing each signature against each network packet time and resource intensive. This implies the need to understand the limits of NIDS before deployment.

The performance of NIDS varies dependent on the nature of the traffic examined. Modern NIDS employ sophisticated algorithms which may be vulnerable to Denial of Service (DOS) attacks when traffic fits a specific pattern. For example, in Snort [2] specifically crafted network traffic can exploit the pattern matching algorithm resulting in inspection times up to 1.5 million times longer than that of benign packets [3]. Attackers can use weaknesses such as these to bypass or neutralize NIDS [4].

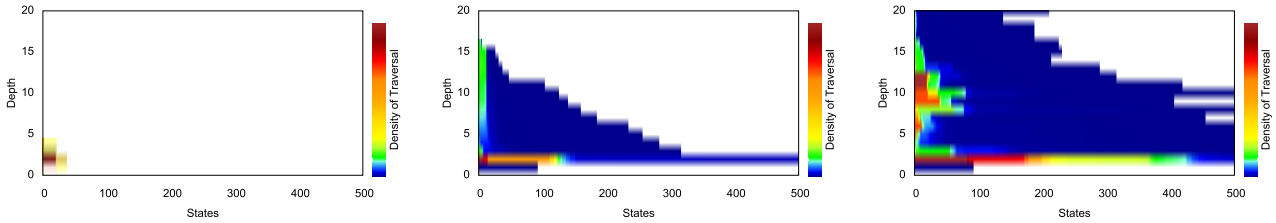
Evaluation of NIDS is commonly performed using pre-generated traffic that is transmitted directly into the NIDS. These pre-generated files are created either as a snapshot of traffic in a real network, or as carefully controlled malicious and benign traffic as first illustrated in the MIT Lincoln Laboratory DARPA data sets [5]. The NIDS performance then becomes a function of read-time and number of correctly identified attacks. Unfortunately, these techniques suffer severe disadvantages. First, for synthetically created traffic there often exist unintended phenomena that can mark the traffic as suspicious or benign [6]. This can cause NIDS evaluation to have misleading results if those phenomena are targeted. Worse, such pre-generated traffic often exerts little burden on NIDS as the vast majority of the traffic is both benign and disjoint to the set of rules used by the NIDS. Thus, the traffic can be processed at maximum efficiency providing an overly optimistic evaluation of the NIDS capabilities.

Generating traffic derived from the NIDS rules database has been proposed as a more effective evaluation of NIDS [7], [8]. These methods produce payloads as part of a random walk through a finite automata derived from the NIDS rules. The resultant data matches, or partially matches, many of the NIDS patterns which increases the NIDS effort in matching; requiring orders of magnitude more effort to process than traffic that is disjoint to the rules. However, this diversity in evaluation comes at the cost of large internal memory required for data generation. This, coupled with the random nature of the data generation, makes the generation process very slow; far below the speed of commonly-used multi-Gbps links. Use of specialized hardware for traffic generation can improve matters, but comes at significant monetary cost, and such systems are not always designed for reproducible test results.

This paper describes a novel approach to workload generation for NIDS evaluation using a generative grammar. The generative grammar represents an optimized context-free grammar equivalent to a subset of all the strings matching the given rule database. Payload generation is up to four times faster than current rule-derived payload generation techniques, utilizes a fraction of the memory, and exerts nearly 80 times more burden on the NIDS over random traffic. Further, we adopt a flexible pipelining architecture

Manuscript received December 1, 2015.
Manuscript revised March 31, 2016.
Manuscript publicized May 13, 2016.
[†]The authors are with Petabi, Inc., Irvine, CA 92612, USA.
[††]The author is with Department of Aerospace Information Engineering, Konkuk University, Seoul 05029, Korea.
a) E-mail: parkjk@konkuk.ac.kr (Corresponding author)
DOI: 10.1587/transinf.2015EDP7483

Copyright © 2016 The Institute of Electronics, Information and Communication Engineers



(a) Zero-filled payload (b) Random payload (c) Rule-derived payload

Fig. 1 NFA coverage comparison out to depth of 20.

for implementation to provide for multi-core generation of traffic while ensuring the repeatibility of experiments.

2. Workload Generation for Signature-Based NIDS

Signature-based NIDS describe rules in specific formats. These rule formats describe traffic features including protocol header specifications and payload patterns. Header features have efficient traffic generation under current methodologies [4], [8]. Payload generation, however, can prove complex especially if it is necessary to exert load on a target NIDS.

To visualize the operation of the matching engine of a NIDS, we use the simple diagram in Fig. 2 as a representation of an automaton. In the diagram we omit details, such as labels on transitions, for simplicity. The label of each node represents the depth of the node (the shortest distance from the root to that node). Any traversal must pass through shallower nodes prior to reaching deeper nodes. Further, any non-match traversal will return to the root. Thus, shallower nodes are much more likely, and more often, visited than deeper nodes.

Given Fig. 2 it is easy to derive that if traffic never moves deep into the automata then only those states near the root will ever be traversed. An example shallow traversal is marked with the grid pattern in Fig. 2. Since there exist only a small number of such states near the root then all of these states will likely fit in the cache. Thus, a shallow traversal of the automaton can run at maximum efficiency as there are very few cache misses. Uniformly distributed payloads, where each byte is in the interval $[0, 255]$, will demonstrate such a shallow traversal. The probability of random input matching more deeply with any rule diminishes exponentially as the traversal extends deeper into the automaton. This causes continuous shallow traversals with the rare deeper traversal. In fact, most benign traffic is largely disjoint to the patterns described in the NIDS rules and thus mimics the shallow traversal of random traffic. This stems from the fact that NIDS target outliers rather than average traffic and thus normal traffic is unlikely to match deeply with the rules.

Traversal for a matching packet, however, demands more time and resources. The states marked with slanted lines in Fig. 2 represent such a traversal ending at depth $i+1$. It goes deeper into the automaton, passing through the shallow area. If the path is long the later transitions likely cause

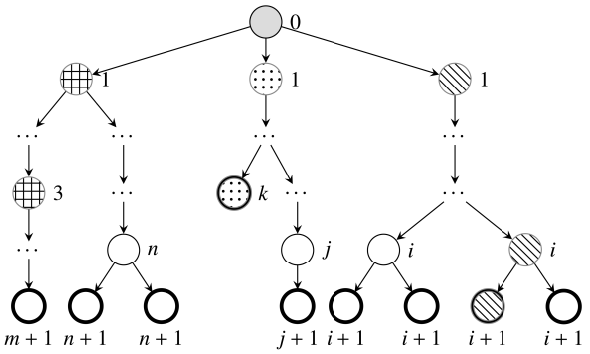


Fig. 2 Automata traversal (thick circles signify final states).

cache misses which increase the matching latency. It is simple for an attacker to send a packet that matches to a depth of i but not $i+1$ and thus not fully matching a rule [4]. Such a packet has a matching latency almost as long as the matching packet, but does not trigger an alarm. By sending a large number of such packets for various rules an attacker can overburden some NIDS [3], [4].

Since the NIDS is most burdened under matching traffic a rule-derived traffic generator is the logical choice for imposing load on the target system [7], [8]. The rule-derived traffic model fills the payload of a packet using a sequence of bytes built through a random traversal of the very automaton used by the NIDS to match traffic. This model may use a single pattern, or the automaton for all of the rules. In either case, there exists an automaton of one or more patterns from the NIDS rule database. The traversal follows a simple strategy. Given a variable p, $0 < p \leq 1$, a traversal goes deeper into the automaton with probability p, and with probability $(1-p)$ returns to the start (or optionally selects a new rule). At each step deeper into the automaton, a valid traversal (i.e a valid byte to move deeper) is randomly selected and appended to the generated payload string. This process continues until a payload string L in length is generated for some predetermined $L > 0$.

Figure 1 shows the coverage of a non-deterministic finite automaton (NFA) under zero-filled payloads (each byte is zero), random payloads, and rule-derived payloads. The automaton is an NFA built using GPP-Grep [9] with regular expressions randomly selected from the Snort rule set [10]. The NFA recorded the counts of each state visit as it traversed the payloads. After processing the payloads the

nodes were sorted in ascending order by depth, then in descending order by number of visits at each depth in order to plot a heat map as illustrated in Fig. 1. In the heat map, a coordinate (x, y) corresponds to the x^{th} state (according to the number of visits) at depth y. The temperature reflects the number of visits to the state.

Figure 1 (a) illustrates the heat map for matching against zero-filled payloads. Because the zero-filled payload causes the NFA to visit exactly the same set of states for every packet, and since the number of states is small, this is a best-case scenario for the matching engine. Only a small number of states within the depth of 5 are visited. The random payloads, as shown in Fig. 1 (b), cause more states to be visited though the bulk of all visits still reside in those few states clustered near the root of the NFA. Finally, Fig. 1 (c) demonstrates how payloads generated by random walks over the same automaton cause much wider and deeper coverage of the matching automata.

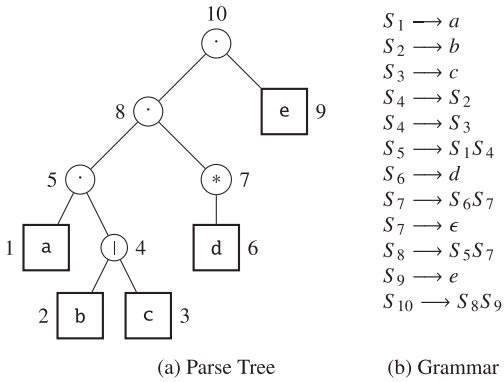
Random traffic, and even network snapshots, are easily generated but are insufficient to evaluating a NIDS under load. Rule-derived traffic, however, can generate traffic that will intersect with the NIDS rules out to some ratio defined by p enabling a systematic evaluation of the NIDS matching automata. Unfortunately, this method is both memory intensive and slow as it must perform multiple traversals of the automata as well as maintain that automata in memory. Any non-trivial rule set will create a large automata that cannot be efficiently traversed and which may make multi-Gigabit traversal unfeasible.

Workload for NIDS evaluation must be created such that it imposes the maximum load on the target NIDS which includes not only generating traffic that burdens the matching automata, but also generating traffic at line speeds. Thus, the workload generation problem that this paper addresses is to create network packets with payloads derived from the given rule set with the following properties.

- **Burden** Every payload must cause the NIDS to exert a maximum amount of effort to match.
- **Scalable** Since payload generation is a computationally intensive task it is necessary that any solution provide a natural path for parallelism.
- **Reproducible** True evaluation must be reproducible such that one test given a specific set of configurations can be reproduced exactly, every time, so long as the configuration does not change.

3. A Rule-Derived Generative Grammar

To meet the aforementioned criteria we propose a rule-derived generative grammar that is optimized for fast and reproducible payload generation. The grammar is a simplification of the normal rule-derived process and, as such, can create payloads that exert heavy burden on the target NIDS. The grammar is constructed according to following steps. First, build a parse tree for each rule. Second, each parse tree is converted into a context-free grammar (CFG). Third, and final, each context-free grammar is optimized for efficiency during payload generation.



(a) Parse Tree (b) Grammar

Fig. 3 Annotated parse tree

3.1 Building a Parse Tree from a Regular Expression Rule

The patterns used in Signature-based NIDS can be represented as regular expressions (even if they are not originally in such a syntax). Given a regular expression it is a simple process to parse the regular expression, using a library like Perl Compatible Regular Expressions (PCRE) [11], to create a sequence of opcodes that can be translated into a parse tree of the regular expression [12]. Figure 3 (a) illustrates an example parse tree built using this methodology.

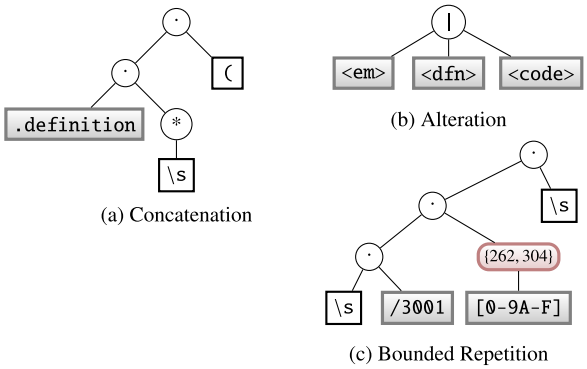
3.2 Building a Grammar from a Parse Tree

T is an abstract syntax tree of the given regular expression, and n is a node in T at which the subtree to convert to productions is rooted.

REGEXP-PRODUCTIONS(n)

```
1  if n.type = EPSILON
2     P = {S_n.id → ϵ}
3  elseif n.type = SYMBOL
4     P = {S_n.id → n.value}
5  elseif n.type = CONCATENATION
6     P = {S_n.id → S_n.left.id S_n.right.id} ∪
           REGEXP-PRODUCTIONS(n.left) ∪
           REGEXP-PRODUCTIONS(n.right)
7  elseif n.type = ALTERNATION
8     P = {S_n.id → S_n.left.id, S_n.id → S_n.right.id} ∪
           REGEXP-PRODUCTIONS(n.left) ∪
           REGEXP-PRODUCTIONS(n.right)
9  elseif n.type = KLEENE-STAR
10    P = {S_n.id → S_n.left.id S_n.id, S_n.id → ϵ} ∪
           REGEXP-PRODUCTIONS(n.left)
11 return P
```

A CFG is constructed from the parse tree through a depth-first search where each node in the tree is recursively translated into a corresponding grammar production as defined by the steps in REGEXP-PRODUCTIONS. REGEXP-PRODUCTIONS takes a node as an argument with the root node



(a) Concatenation (b) Alteration (c) Bounded Repetition

Fig. 4 Flattening examples

the initial argument. As an example, we take the regular expression `a(b|c)d*e` and show its parse tree in Fig. 3 (a). Each node is decorated with an ID, representing a depth-first search ordering of the nodes. Figure 3 (b) shows a grammar built by the REGEXP-PRODUCTIONS.

3.3 Grammar Optimization

A grammar built by the REGEXP-PRODUCTIONS generates payloads that match a given regular expression. However, generating a large payload may require an excessive amount of time as each production can produce at most one byte of payload with potentially multiple productions to arrive at a single byte of payload. More productions cause more overhead which can make for slower generation. Thus, it is desirable to combine productions, where appropriate, to limit the impact of productions on generation efficiency. However, care must be maintained to ensure that the merging of productions does not serve to reduce the quality (in terms of load caused on the target NIDS) of the payload generated.

3.3.1 Production for a Fixed String

A notable characteristic of regular expressions in NIDS rules is that they contain many long fixed strings. For example, a rule from the Snort rule set has a regular expression

```
\x2edefinition\s*\x28
```

where `\x2e` and `\x28` are the hexadecimal representation of the ASCII characters "`.`" and "`(`", respectively. Note that the regular expression starts with a fixed string "`.definition`". Converting this string into productions using REGEXP-PRODUCTIONS will result in a skewed binary tree. Traversing such a tree at run-time to generate the same string will consume a lot of time. Thus, we flatten the tree and merge the corresponding productions into a single production. Figure 4 (a) is the result of flattening. As is shown, using a leaf node that can generate a fixed string instead will reduce the number of productions that are involved during workload generation.

3.3.2 Production for Alternation

Many regular expressions in NIDS rules have alternations so as to capture variations of the same attack in a single rule. For example, the following regular expression is part of a Snort rule.

```
(<em>|<dfn>|<code>)
```

The result of REGEXP-PRODUCTIONS for this rule will be a tree with a root alteration node that has another alteration node as a child. One fixed string is attached to the first alternation node, and two to the second. A major drawback of this tree is that it leads to a biased selection among alternatives if the generation algorithm takes each branch of an alternation node with equal probability. The node closer to the root has a higher probability to be selected over those further from the root. However, we note that this tree can be flattened into a single alternation, with three choices, as illustrated in Fig. 4 (b). Flattening the tree not only removes productions but also provides an equal probability to each alternative.

3.3.3 Production for Repetition with a Range

Repetition is another common feature of regular expressions used for NIDS. Strictly speaking, a Kleene star is the only operator for a repetition. In practice, however, many alternations can be considered as repetitions with lower and upper bounds on the number of repetitions. For example, "?" means zero or one occurrence, and in general, "`{n,m}`" means `n` or more repetitions up to `m`. The following is a regular expression selected from the Snort rule set that contains a bounded repetition.

```
\s/3001[0-9A-F]{262,304}\s
```

The parse tree of this regular expression will have a subtree consisting of 261 concatenations followed by an alternation node with 43 child nodes. To reduce the size of memory used to store the productions, we define a special production that contains the repetition boundaries together with the symbol to be repeated as shown in Fig. 4 (c).

4. Workload Generator Implementation

In order for payload generation to meet the demands of high speed environments where it may be necessary to generate as many as 14.88 million packets per second to saturate a 10 Gb link, it is necessary that all workload generator components are as efficient as possible. In particular, the following items are of significant importance.

- Minimal generation time. The time spent generating each packet must be a small as possible while still maintaining the quality of content.
- Minimal data/input overhead. Where possible avoid copies of data or added layers of abstraction that may slow down writing packets.

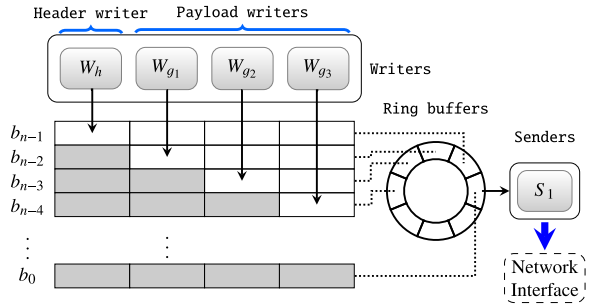


Fig. 5 Workload generator architecture

- A careful parallel design. Minimize blocking among threads to allow for seamless addition of cores as well as reproducibility of tests.

The overall architecture of the proposed workload generator is presented in Fig. 5. It has two kinds of entities: Writers and Senders. Packets, b_0 to b_{n-1}, are represented by large rectangles divided into four pieces, each of which is assigned to a different writer. For example, W_h, the Header Writer is filling the header of b_{n-1}, pointed to by the arrow. Payload Writers like W_{g_1} generate part of the payload according to the loaded generative grammar. As packets are completed, they are then flagged for the Sender, packets prior to b_{n-3} are such packets and are ready to be sent by the Sender. Because a Writer conducts a CPU-intensive task, each Writer should run on its own processor/core in a multiprocessor/multi-core system, and its implementation must be optimized for CPU time. On the other hand, the main task of the Sender is to move packets from the ring buffer to the network medium, and minimizing I/O bandwidth is its optimization goal. Finally, multiple Writers must coordinate their packet generation to avoid conflicts in accessing memories.

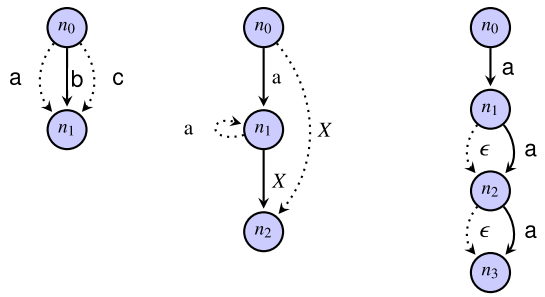
4.1 Pre-Selection in Alternation

As noted earlier, burden for the matching automata of NIDS is primarily a function of how many unique states are visited during traversal (i.e., moving deeper into the automata causes more burden). The productions outlined in Sect. 3 are simplifications of the rule-derived method of payload generation yet still produce payloads that will intersect deeply with the target NIDS matching automata. Some productions, however, do not contribute to increasing load on the target NIDS while requiring much computational work during generation. For instance, a character class, such as \d, creates ten different transitions between the same pair of states. The corresponding production will have ten alternatives, one for each terminal symbol, and the generator must choose one at run-time. Regardless of the choice of symbol the next state remains the same in the automaton; providing more choices does not always lead to larger state coverage in the automaton. Such a production offers limited value at a significant cost in generation time.

We analyzed 700 regular expressions from the Snort rule set and present in Table 1 the number of occurrences of alterations per rule. As is shown in the table, on average, 6.87 productions per rule can be simplified by pre-selecting one of the choices and omitting the other transitions as shown in Fig. 6 (a). Considering the large rule base generally adopted by NIDS, pre-selection has great potential to minimize redundancy caused by the alternation.

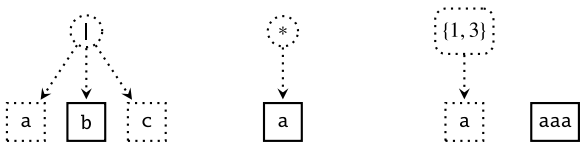
Table 1 Alternations and repetitions in the Snort rule set

		# of occurrences / rule	Total
Alterations	Character Class	6.18	6.87
	Case-insensitive	0.69	
Repetitions	*	4.12	6.03
	+	1.21	
	?	0.51	
	Open Interval	0.02	
	Closed Interval	0.18	



(a) Alternation (b) Unbounded Repetition (c) Bounded Repetition

Fig. 6 Pre-selection in NFA



(a) Alternation (b) Unbounded Repetition (c) Bounded Repetition

Fig. 7 Pre-selection in parse tree

To apply pre-selection for alternation, we identify productions having alternation with terminal symbols and ϵ only. Then we randomly select one of the symbols and discard the rest. We replace the parent node with the chosen node as shown in Fig. 7 (a).

4.2 Pre-Selection in Repetitions

We observe that repetitions also appear frequently in NIDS rule sets. For example, as shown in Table 1, the Snort rules examined have, on average, 6.03 instances of repetitions per rule. An unbounded repetition creates a state with a self-loop. Taking a transition over the loop keeps the active state in the same state, and does not increase the burden on the matching. As a matter of fact, it only prevents the traversal process from exploring a deeper and broader area of the NFA. It is helpful to remove such loops in the pre-generation stage as demonstrated in Fig. 6 (b).

NFA may be constructed differently. For example, Thompson automata [13] are constructed with ϵ-transitions, Glushkov automata [14] are constructed without ϵ-transitions. For different NFA construction, the impact of the same payload can be different as well. For NFA constructed without ϵ-transition, using an empty string to replace unbounded repetition will lead to visiting fewer states during matching. To try to explore all the states in the NFA, during pre-selection, one instance of the operand of unbounded repetition is generated instead as in Fig. 6 (b). The pre-selection process for the Kleene star is also demonstrated in Fig. 7 (b). Instead of having the recursive production created by the Kleene star node in the parse tree, we retain only the base case.

In the case of a bounded repetition, there are at least as many states as the upper bound of the repetition. For example, Fig. 6 (c) shows the NFA traversal of `a{1,3}`. It is clear that, by selecting the maximum number of repetitions, the string will cause the matching engine to visit the maximum number of states. Thus, we replace the bounded repetition production with one generating a string according to the upper bound as illustrated in Fig. 7 (c).

4.3 Randomness

There are multiple occasions during the traffic generation process where random numbers are necessary to ensure the unpredictability of the generated content. These occasions include protocol selection, rule selection, and alternation and repetition in the grammar. However, pseudo-random number generation is a CPU-intensive task, which becomes a bottleneck in high-speed traffic generation. This can be alleviated by pre-generating a random sequence and using the sequence at run-time instead of computing a new pseudo-random number each time as needed. To maintain a certain degree of unpredictability, a relatively long random sequence must be available in the traffic generator at run-time.

4.4 Transmission

Once packets are created, the next step in workload generation is to transmit those packets to the network. Handling the generated packets through the network stack in an operating system involves a large amount of data transfer between the user space and the kernel space. With the size of the data we need to pass along the system, the copying latency can be larger than 100 ns [15] which can become a bottleneck.

The predominant solution to avoid this data transfer cost is to use dedicated hardware devices optimized for data generation and transfer. This approach is used by many commercial vendors, like Ixia [16], who create hardware devices that can generate traffic at multi-Gigabit rates. However, hardware implementations are typically difficult to adapt and use and often cost-prohibitive. These issues can serve as serious drawbacks when evaluating NIDS. In order to minimize latency for data transmission and avoid protocol processing overhead a number of architectures have been proposed including Intel's Data Plane Development Kit (DPDK) [17], PF_RING [18], and netmap [15]. They adopt a common strategy that allows packet buffer sharing between the operating system kernel and a user application. Any of these architectures can be used to gain most of the advantage of hardware implementation while retaining the advantage of developing in the user space on a general purpose processor.

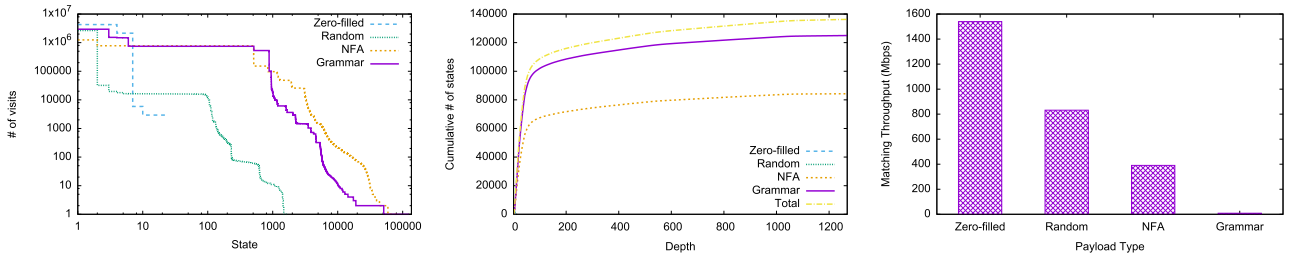
The workflow of the proposed workload generation system is as follows. The workload generator randomly selects a rule and fills the protocol header fields in a shared packet buffer. To minimize run-time computations most of the protocol headers are pre-generated in the form of a template for each protocol. At run-time, this template is copied into the buffer, and only the variable fields are modified. The payload of the packet is written into the packet buffer using the generative grammar corresponding to the selected rule. The cursor in the packet buffer is then incremented to point to the next slot in the buffer. When there are no more empty slots available, the workload generator notifies the Ethernet device driver of the availability of packets which then transmits the packets out the interface. As soon as the slots are emptied, the traffic generator resumes packet generation.

4.5 Parallel Generation

Generating traffic at 10 Gbps link speeds is challenging with as small a time budget as 70 nanoseconds within which to generate a 64 byte packet. As such, parallelism is an important, even necessary, aspect of any workload generation scheme that seeks to saturate high-speed links. However, there is an added constraint for workload generators in that the generated traffic must be reproducible given the same initial configuration. Naïve applications may introduce randomness to packet ordering. This is caused by contentions among threads. This poses great difficulties in reproducing test results. Therefore, multiple threads in the traffic generator must coordinate to maintain the order of packets in every run. This is an impossible goal if each thread works independently. Instead, we propose pipelining of packet generation so that the order of packets is determined solely by the first stage of the pipeline.

Figure 5 demonstrates a snapshot of the pipelining process. W_h fills in the header of packet b_{n-1}. Packet b_{n-2} has already had its first section, the header section, filled and is thus assigned to writer W_{g_1} where the second section i.e., the first section of the payload, is written to the packet. In a similar fashion, W_{g_2} fills the third section for packet b_{n-3} while W_{g_3} completes the last section for packet b_{n-4}. The shadowed regions in the figure represent the completed sections for each packet. Each writer only writes a specifically set section of the packet (i.e., the header section or one of the payload section—in Fig. 5 a payload is divided into 3 sections). At each time point the writers are all working on different packets.

The context of the content generation must follow the packet through the pipeline. This includes both the sequence



(a) NFA Coverage (b) Depth Reached (c) Burden

Fig. 8 Coverage, depth, and burden on NIDS.

of grammar symbols derived up to that point in the pipeline as well as the position of the next byte to be written in the slot. W_{g_1} starts payload generation with the start symbol, generating payload using left-most derivation, and when it fills the given portion of the packet, the derivation status is passed to the next writer W_{g_2}. As shown in Fig. 5, to complete the second data portion of packet b_{n-3}, represented by the rectangle in the third row and the third column, W_{g_2} has to start from where W_{g_1} ends. Therefore the entire context information should be passed to the writer in the next stage.

Each thread must maintain state for their tasks in order to distribute workload and avoid conflicts. Each writer needs its own pseudo-random number generator state and an independent cursor in the ring buffer. Such information is crucial for avoiding race conditions, isolating their memory accesses and, eventually, guaranteeing the maximal resource utilization.

5. Evaluation

To evaluate the proposed solution we first examine how well the generated traffic covers the target NIDS matching automata. Second, we look at the run-time aspects of the workload generation including generation speed and memory usage. Comparisons are made between zero-filled payload generation, random-payload generation, rule-derived payload generation, and the generative grammar proposed in this work. The evaluations were conducted on a server with an Intel Xeon CPU E5-26500 @ 2.00 GHz, and 8200 MB RAM running a target NIDS that is based on GPP-Grep [9], which utilizes an NFA as a matching automata. The matching automata provides counters for tracking the number of visits to each state in the NFA. The NFA is constructed from a set of regular expressions. The results shown here represent a proprietary rule set provided by a commercial NIDS vendor. We use these rules in this paper as they contain definitions for the most recent attacks. For completeness, we have conducted the same experiments with other rule sets and observed identical trends.

5.1 NFA Coverage and Depth

To compare NFA coverage between different workload types, we count the number of visits to each NFA state for the four workload types. In Fig. 8 (a), we order the states in the NFA according to the frequency of visits, and plot how many visits each state receives during matching. Note that both axes use log scale. Zero-filled payload covers a very small number of states. Random payload covers more, but still less than 1% of the NFA. On the other hand, the rule-derived traffic and the grammar-based traffic cover considerably more states in the NFA. In fact, the generative grammar payload nearly mirrors the rule-derived payload while covering 124,978 states out of the 136,275 states in the NFA, or more than 90%. This demonstrates that the generative grammar payload is as effective as the rule-derived payload generation in NFA coverage.

To further demonstrate the influence of payload types on NFA traversal we present Fig. 8 (b). It shows the cumulative number of states visited as a function of the depth of the states. In Fig. 8 (b) the "Total" curve shows the maximum number of states that could be visited dependent on the depth. As such, Fig. 8 (b) shows that the generative grammar payload follows the "Total" curve quite closely illustrating that it is very close to exploring all states in the NFA. The rule-derived payloads also perform well, but less so than the generative grammar payloads. This is due to the fact that the traversal in the NFA can get caught in loops that do not necessarily move deeper into the NFA. Further, we note that the zero-filled and random payload generation does not reach deeper than a depth of 9 and visit no more than a few thousand states and do not even register on Fig. 8 (b).

5.2 Burden

Next we examined the burden exerted on the NFA given the four payload generation techniques. The expectation is that the payload generation techniques that have a larger NFA coverage will exert greater burden on the target NIDS.

Figure 8 (c) shows the throughput of the target NIDS. Zero-filled payload cause little burden to the target NIDS while random payloads incur more burden, though they are still matched in excess of 800 Mbps. The target NIDS matches the rule-defined payload at half the speed of random payload, but that is still much faster than the generative grammar payload which causes the NIDS to match at barely 10 Mbps. This demonstrates that the generative grammar is actually more effective at generating load on the target NIDS by a factor of forty. Once again, this stems from the fact that the generative grammar will randomly select a rule

Table 2 Performance of workload generator

	Generation Speed (Kpps)	Memory Usage (KB)
Zero-filled	812.750	8312
Random	155.590	8312
NFA	6.440	290000
Grammar	26.400	13490

and follow that rule to its completion while the rule-derived approach can get caught in loops or end up following more common paths through the NFA (as a result of multiple transitions between particular states).

5.3 Workload Generation Speed

Given a minimum-sized Ethernet frame of 64 bytes (without payload), our workload generator is able to send 14.88 million pps on a 10 Gbps link (saturating the link) on a single core. As such, the overhead from header generation and writing to the network interface offers no bottleneck. Table 2 illustrates the speed of generation, in packets per second, for 1518 byte frames using the specified payload generation technique on a single core. The zero-filled payload generation saturated the 10 Gbps link with a rate higher than 800 Kpps (Kilopackets per second) while the random payload generation shows a speed fast enough to saturate a gigabit link. The rule-derived generation is the slowest with a throughput of 6.44 Kpps. The grammar-based workload generator achieves 26.4 Kpps, four times as fast as rule-derived payload generation, yet in comparison with random traffic, is still too slow. Despite the efforts to make the runtime generation faster for the generative grammar it is still too slow for multi-gigabit generation. Future work will seek methods to remedy this issue.

5.4 Memory Consumption

Table 2 presents the amount of memory consumed by each workload generation technique. The zero-filled and random payload generation consume little memory during generation. The rule-derived workload generator consumes an excessive amount of memory. Interestingly, the generative grammar consumes only about half-again as much memory as the random payload generator, which is orders of magnitude less than the rule-derived payload generation. This illustrates that memory considerations for the generative grammar have been successful.

5.5 Scalability

To demonstrate scalability through pipelining we present Fig. 9 which demonstrates workload generation throughput increases as the number of pipeline stages increases. We see a linear increase as the number of pipeline stages are added with four stages demonstrating a 2.5 times speedup implying that a gigabit link could be saturated with roughly 8 threads assuming the linear speedup holds.

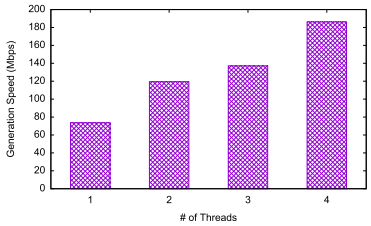


Fig. 9 Scalability

6. Background

NIDS use a variety of techniques to detect malicious traffic. These techniques are often divided into two specific types: anomaly detection and signature-based detection. Anomaly detection applies algorithms to specific network features to detect anomalous behaviors and misuse. Signature-based systems employs sets of rules that define suspicious traffic and alert whenever the criteria of these rules are met. Most modern NIDS use a combination of both methods.

As discussed in Sect. 2 evaluation techniques for NIDS are often inadequate. The makeup of the traffic typically lacks the diversity to sufficiently evaluate the NIDS [7], [8], [19]. The primary problem is that most traffic generators focus on creating realistic traffic, not on creating traffic to specifically stress the detection algorithms of NIDS. Brauckhoff et al. offer FLAME (a Flow-Level Anomaly Modeling Engine) [20] and Sommers et al offer Trident [21], [22]. These tools are great for creating traffic and attacks but are designed with the purpose of evaluating accuracy rather than other performance metrics and do not offer much support for burdening detection algorithms.

Replaying captured traffic, with a tool like Tcpreplay [23], is the simplest way to generate network traffic that resembles real traffic. Unfortunately, traffic captures suffer from a host of issues from privacy to a general lack of diversity in the traffic such that using them for anything other than determining fitness for a particular environment is largely inadequate [19]. In fact, nearly all packet-capture based evaluation methods suffer from a lack of diversity as well as other issues [6], even if they offer a good opportunity for testing detection accuracy.

There are numerous traffic generators, such as OSNT [25], Caliper [26], SWORD [27], rule2alert [24], and StreamGen [28] that are excellent for evaluating packet headers and verifying protocol implementations but lack adequate algorithms for generating payloads to burden NIDS pattern matching algorithms. A common tactic to circumvent this problem is to insert attack samples in random payloads, as done by Ixia [16] and Xena [29]. The majority of traffic, however, remains random, and the inserted attacks are still not systematic in their evaluation of the NIDS pattern matching and may retain levers that can be exploited for trivial detection of suspicious traffic [6].

The only way to sufficiently test a NIDS is to develop a systematic method to examine the ranges of possible in-

put [19]. Becchi et al demonstrated a workload generation model for evaluating regular expression matching engines [7] and Valgenti et al expanded this model for NIDS in general [8]. These techniques employ the rule sets of the target NIDS to generate traffic that can systematically examine the boundaries of the NIDS. However, these generation techniques are resource intensive and often slow and thus motivate the work outlined here.

7. Conclusion

We demonstrated the deficiency of using random traffic, or traffic disjoint from the NIDS rules, in evaluating a NIDS operation under load. To address this issue we have proposed a generative grammar to maintain a balance between efficiency and effectiveness of workload generation for the NIDS evaluation. The approach achieves higher speed while using less memory and exerts many times more load on the target NIDS. We have implemented the proposed model, using a pipeline architecture for multi-core generation and have demonstrated linear speed-up as cores are added to the pipeline. Generation speeds on a single core are still too slow to meet multi-gigabit link-speed requirements but are sufficient to saturate gibabit links without the need of high-cost hardware. We believe that future refinements to the workload generation will provide the necessary keys to effective NIDS workload generation, at multi-gigabit speeds, on General Purpose Processor platforms.

References

[1] "Secunia vulnerability review 2014: Key figures and facts from a global IT-security perspective," Feb. 2014.
[2] The Snort Project, SNORT User Manual 2.9.7, Oct. 2014. https://www.snort.org/documents/snort-users-manual
[3] R. Smith, C. Estan, and S. Jha, "Backtracking algorithmic complexity attacks against a NIDS," Proc. 22nd Computer Security Applications Conference, pp.89–98, Dec. 2006.
[4] V.C. Valgenti, H. Sun, and M.S. Kim, "Protecting run-time filters for network intrusion detection systems," Advanced Information Networking and Applications (AINA), IEEE 28th International Conference on, pp.116–122, May 2014.
[5] R. Lippmann, J.W. Haines, D.J. Fried, J. Korba, and K. Das, "The 1999 DARPA off-line intrusion detection evaluation," Computer Networks, vol.34, no.4, pp.579–595, Oct. 2000.
[6] J. McHugh, "Testing intrusion detection systems: A critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by Lincoln Laboratory," ACM Transactions on Information and System Security, vol.3, no.4, pp.262–294, Nov. 2000.
[7] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," Proc. IEEE International Symposium on Workload Characterization, pp.79–89, Sept. 2008.
[8] V.C. Valgenti and M.S. Kim, "Simulating content in traffic for benchmarking intrusion detection systems," Proc. 4th International ICST Conference on Simulation Tools and Techniques, March 2011.
[9] V.C. Valgenti, J. Chhugani, Y. Sun, N. Satish, M.S. Kim, C. Kim, and P. Dubey, "GPP-Grep: High-speed regular expression processing engine on general purpose processors," Proceedings of the 15th International Symposium on Research in Attacks, Intrusions and Defenses, vol.7462, pp.334–353, Sept. 2012.
[10] M. Roesch, Snort. Cisco, 2015. http://www.snort.org
[11] P. Hazel, PCRE—Perl-compatible regular expressions, University of Cambridge, Jan. 2014.
[12] M. Shao, Grammar-driven workload generation for efficient evaluation of intrusion detection systems, Master's thesis, Washington State University, May 2015.
[13] K. Thompson, "Programming techniques: Regular expression search algorithm," Commun. ACM, vol.11, no.6, pp.419–422, 1968.
[14] V.M. Glushkov, "The abstract theory of automata," Russian Mathematical Surveys, vol.16, no.5, pp.1–53, 1961.
[15] L. Rizzo, "netmap: A novel framework for fast packet I/O," Proc. 2012 USENIX Annual Technical Conference, June 2012.
[16] "Anue GEM and XGEM network emulator specifications." http://www.ixiacom.com/products/ixia-network-emulators
[17] Intel Data Plane Development Kit: Programmer's Guide, June 2014.
[18] PF_RING User Guide, April 2014.
[19] V.C. Valgenti and M.S. Kim, "Increasing diversity in network intrusion detection system evaluation," IEEE Global Communications Conference, Exhibition, and Industry Forum (GLOBECOM), pp.1–7, Dec. 2015.
[20] D. Brauckhoff, A. Wagner, and M. May, "FLAME: A flow-level anomaly modeling engine," Proc. Workshop on Cyber Security Experimentation and Test, July 2008.
[21] J. Sommers and P. Barford, "Self-configuring network traffic generation," Proc. ACM/USENIX Internet Measurement Conference, Oct. 2004.
[22] J. Sommers, "Toward comprehensive traffic generation for online ids evaluation." Technical Report available at: http://pages.cs.wisc.edu/˜pb/trident_final.pdf
[23] A. Turner, tcpreplay—Replay network traffic stored in pcap files, 2008. http://tcpreplay.synfin.net/tcpreplay.html
[24] "rule2alert." https://code.google.com/archive/p/rule2alert/
[25] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. McKeown, N. Feamster, B. Felderman, M. Blott, A.W. Moore, and P. Owezarski, "OSNT: Open source network tester," IEEE Network, vol.28, no.5, pp.6–12, Oct. 2014.
[26] M. Ghobadi, G. Salmon, Y. Ganjali, M. Labrecque, and J.G. Steffan, "Caliper: Precise and responsive traffic generator," Proc. 20th IEEE Annual Symposium on High-Performance Interconnects, pp.25–32, Aug. 2012.
[27] K.S. Anderson, J.P. Bigus, E. Bouillet, P. Dube, N. Halim, Z. Liu, and D. Pendarakis, "SWORD: Scalable and flexible workload generator for distributed data processing systems," Proc. 38th Conference on Winter Simulation, pp.2109–2116, Dec. 2006.
[28] M. Mansour, M. Wolf, and K. Schwan, "StreamGen: A workload generation tool for distributed information flow applications," Proc. 43rd International Conference on Parallel Processing, vol.1, pp.55–62, Sept. 2004.
[29] "Xena test modules." http://www.xenanetworks.com/test-modules

Min Shao received the B.E. degree in automation in Northwestern Polytechnical University, Xi'an, China, in 2010 and M.S. degree in computer science in Washington State University, in 2015. In 2013, she joined Petabi, Inc. and has been a software developer for high-performance traffic generating since then.

Min S. Kim earned his B.S. in Computer Engineering from Seoul National University in 1996, and Ph.D. in Computer Science from The University of Texas at Austin in 2005. Then he jointed Washington State University, where he remained a member of the faculty in School of Electrical Engineering and Computer Science until 2015. He founded Petabi, Inc. in 2013 and has been the Chief Executive Officer since then, developing high-performance network security appliances.

Victor C. Valgenti earned a BA in Linquistics from the University of Montana in the early 90's. After several years teaching English he migrated to computers earning a MS in Computer Systems in 2002 from City University in Seattle, WA and a PHD in Computer Science from Washington State University ten years later. His initial research concerned simulating traffic for Network Intrusion Detection System Evaluation. More recentaly he has adopted a language approach to the detection of unauthorized network access as well as the dissemination and management of detected events within a system. Over the last decade and a half Victor has held numerous jobs in the computer industry from help desk to software developer. In 2013 he became part of the initial force behind Petabi, Inc where he works to bring his reasearch to market.

Jungkeun Park is currently an Associate Professor of the Department of Aerospace Information Engineering at Konkuk University. He received his B.S. and M.S. degrees in Electrical Engineering from the Seoul National University, Korea, in 1997 and 1999, respectively. He received his Ph.D. in Electrical Engineering and Computer Science from the Seoul National University in 2004. His current research interests include embedded real-time systems design, real-time operating systems, network intrusion detection systems, and UAV systems.