Advance Access publication on September 2, 2008
doi:10.1093/comjnl/bxn043

A Pragmatic Methodology for Testing Intrusion Prevention Systems

Zhongqiang Chen[1], Alex Delis[2,*] and Peter Wei[3]

[1]Yahoo! Inc., Santa Clara, CA 95054, USA
[2]University of Athens, Athens, 15784, Greece
[3]Fortinet Inc., Sunnyvale, CA 94086, USA
*Corresponding author: ad@di.uoa.gr

Intrusion prevention systems (IPSs) not only attempt to detect attacks but also block malicious traffic and pro-actively tear down pertinent network connections. To effectively thwart attacks, IPSs have to operate both in *real-time* and *inline* fashion. This dual mode renders the design/ implementation and more importantly the testing of IPSs a challenge. In this paper, we propose an IPS testing framework termed *IPS Evaluator* which consists of a trace-driven inline simulator-engine, mechanisms for generating and manipulating test cases, and a comprehensive series of test procedures. The engine features *attacker* and *victim* interfaces which bind to the *external* and *internal* ports of an *IPS-under-testing* (IUT). Our engine employs a *bi-directional* injection policy to ensure that replayed packets are subject to security inspection by the IUT before they are forwarded. Furthermore, the *send-and-receive* mechanism of our engine allows for the correlation of engine-replayed and IUT-forwarded packets as well as the verification of IUT actions on detected attacks. Using dynamic addressing and routing techniques, our framework rewrites both source and destination addresses for every replayed packet on-the-fly. In this way, replayed packets conform to the specific features of the IUT. We propose algorithms to partition attacker/ victim-emanated packets so that they are subjected to security inspections by the IUT and in addition, we offer packet manipulation operations to shape replayed traces. We discuss procedures that help verify the IUT's detection and prevention accuracy, attack coverage and behavior under diverse traffic patterns. Finally, we evaluate the strengths of our framework by mainly examining the open-source IPS *Snort-Inline*. IPS deficiencies revealed during testing help establish the effectiveness of our approach.

Keywords: testing of intrusion prevention systems; testing methodology; inline operation; detection and prevention accuracy of IPSs

Received 20 July 2007; revised 21 May 2008

1. INTRODUCTION

Firewalls, anti-virus systems (AVSs), and intrusion detection systems (IDSs) have become indispensable elements of the network infrastructure providing protection against attacks [1–3]. However, such security devices may not always be effective against exploits. Firewalls mainly differentiate traffic on fixed ports and protocol fields and fail when it comes to attacks on standard services including *HTTP, SMTP* and *DNS* [4, 5]. Both AVSs and firewalls do not inspect traffic initiated within intranets allowing compromised internal machines to become spring-boards for *distributed denial-of-service* incidents [6–9]. Although IDSs may perform layer-7 inspection on traffic originating from both internal and external networks, they are 'passive' in nature and do not prevent attacks from reaching their destinations [5]. In this context, *intrusion prevention systems* (IPSs) attempt to address the aforementioned weaknesses by working in *inline* fashion between internal and external networks. As they examine every passing packet to prevent malicious attacks in *real-time*, IPSs are considered *active* devices [3, 10] that function pro-actively, and so they can drop packets containing attack signatures, selectively disconnect network sessions and deny reception of streams from specific sources [3]. These actions ultimately change the traffic characteristics as additional packets such as *ICMP destination unreachable*, and *TCP RESET* messages are finally injected into the traffic by IPSs [10].

Since IPSs virtually operate as switches/routers, they often provide packet forwarding, network address translation (NAT) and proxy services all of which are unavailable in IDSs [3, 11].

The *store-and-forward* mechanism of IPSs and their tight integration with the networking infrastructure allow for both detection/prevention of evasive attacks and traffic normalization/scrubbing [12, 13]. As evasive attacks typically manipulate outgoing traffic so that packets are fragmented, overlapped or shuffled [13], IPSs resort to IP de-fragmentation and *TCP* re-assembly to offset such exploits. IPSs may offer differentiated services based on the traffic types encountered—such as those generated by instant messaging and peer-to-peer systems—to limit resource consumption and avoid network congestion [10]. IPSs are also considered superior to IDSs when it comes to identification of malicious traffic as they can judiciously 'interpret' the context in which an attack occurs [10]. For instance, a *TCP*-based exploit without the appropriate three-way-handshake procedure is ineffective even if its packets with malicious payloads reach their destinations. IPSs are not expected to forward such *TCP* traffic in symmetric routing environments, and therefore do not raise any false alerts. On the contrary, IDSs generate alarms for such unsuccessful attacks to avoid packet losses due to their low sniffing rates [14]. IPSs may also feature platform fingerprinting, vulnerability assessment, traffic correlation, dissection of application protocols and abnormal traffic analysis to widen their coverage on attacks [3, 10].

The dual requirement for IPS real-time and inline operation in conjunction with their complex services raises concerns regarding their detection accuracy, successful blocking rates and overall performance [10, 15]. For instance, false positives may induce IPSs to block legitimate traffic resulting in self-inflicted denial of services (*DoS*) attacks [14, 16]. Under extremely heavy traffic and *out-of-resource* conditions, the behavior of IPSs is critical to the viability of the protected systems. Contrary to the *fail-open* strategy followed by firewalls, AVSs and IDSs which all forward traffic without discrimination in such extreme operating conditions, the IPS *fail-close* policy insulates protected networks from both attackers and legitimate users. In light of the above IPS requirements and system complexity, it is evident that testing such devices for their compliance with design objectives is not only challenging but also of paramount importance [14, 15]. Methodologies proposed for testing firewalls, AV systems and IDSs cannot be directly applied as IPSs necessitate real-time and inline operation, delivery of pro-active actions against ongoing traffic, normalization of traffic flows, switching and routing capabilities, real-time *IP* de-fragmentation and *TCP* re-assembly [14, 17, 18]. For instance, the *uni-directional-feeding* method used in IDS testbeds such as `Tcpreplay` to inject packets into the test environment from a single network interface is ineffective here as IPSs refuse to forward any packet arriving at the wrong interface [15, 19]. Similarly, the *send-without-receive* mechanism used by the majority of IDS-testbeds is not applicable to IPSs as the latter do morph their traffic [14, 20]. Although the development of IPSs rapidly progresses to keep pace with the ever-increasing attack population, work on IPS testing lags behind and is far from mature [15]. In this context, we propose a comprehensive methodology to systematically analyze and establish measurements for an *IPS-under-testing* (IUT) with respect to its attack coverage, detection and prevention accuracy, reliability and robustness and performance under various types and intensities of traffic and attacks.

Our proposed trace-driven testbed termed *IPS Evaluator* establishes an inline working environment in which data streams from *internal* and *external* networks are injected into the IUT from different directions; this constitutes a major deviation from the *uni-directional-feeding* strategy used in IDS-testbeds [20]. In order to ensure that every replayed packet is forwarded and subsequently subjected to security inspection by the IUT, our testbed uses dynamic addressing and routing techniques to rewrite source and destination addresses of replayed packets, so that they conform to the test environment. To verify the behavior of an IUT and its actions imposed on the traffic, our testing framework also employs a *send-and-receive* mechanism to capture packets from the IUT and correlate them to replayed packets. Furthermore, our testbed integrates its own retransmission mechanism, traffic re-assembly capability and logging facility. We also discuss in detail test case generation, traffic manipulation and test procedures.

We demonstrate the effectiveness of our methodology by mainly applying it to the testing of the open-source IPS *Snort-Inline* and versions of the commercial product *ForGate*. Our findings show that although *Snort-Inline* displays satisfactory attack coverage and detection/prevention rates, it still generates false positives and negatives under some conditions and misses attacks when it is subjected to stress tests. The main contributions of the *IPS Evaluator* are that:

- It offers an inline working environment for IUTs and injects traffic into IUTs with a *bi-directional-feeding* mechanism to ensure that packet streams initiated by attackers and victims flow in different directions.
- It rewrites on-the-fly source and destination *MAC* and *IP* addresses of replayed packets, so that the latter conform with the test environment. Thus, packets are forwarded and subjected to appropriate security inspections by IUTs.
- Its *send-and-receive* mechanism detects IUT-imposed actions on underlying traffic including packet dropping and connection termination. In addition, the independent logging mechanism in our engine allows independent verification on the consistency between an IUT's actual behavior and its record of events.
- Its *IP* defragmentation and NAT process facilitate the evaluation of IUT's resistance to evasion attacks.

- Its integrated traffic partitioning and manipulation operations help shape the characteristics of the replayed traffic and automate testing procedures.

The rest of the paper is organized as follows: Section 2 outlines related work and Section 3 discusses our proposed trace-driven simulation-engine. Section 4 presents algorithms used to partition packets in traces so that the test procedures can be automated; we also describe our traffic manipulation operations that help produce test cases with desired features. Section 5 discusses our suggested procedure for IPS testing, while Section 6 outlines our experimental evaluation. Concluding remarks and future work are found in Section 7.

2. RELATED WORK

IPS testing has received limited attention thus far. On the contrary, a large number of issues pertinent to IDS testing have been investigated during the last few years [14, 21, 24]. IDS testing typically examines device detection accuracy, availability and reliability, latency and throughput, controllability, as well as alert processing and forensic analysis capabilities [23–25]. Additional issues in IDS testing entail automated test case generation [24, 26], test procedures and benchmarking [23, 27, 28], as well as metrics for IDS effectiveness, coverage, and performance [29]. The evaluation of such IDS features is conducted with the help of either simulation or live testbeds. In simulation-based testbeds, IDSs-under-testing are fed with either tool-generated test cases or captured traces; in contrast, live testbeds directly expose IDSs to real traffic and attacks [30, 20]. Unfortunately, the existing diversity in IDS test methodologies makes any attempt for comparing their effectiveness and test results extremely difficult [17, 18, 31].

The `nidsbench` is an open-source trace-driven IDS test platform that can simulate certain evasion attacks, protocol anomalies and subterfuge activities [20]. Its test cases are derived from captured traces and are replayed using the *uni-directional-feeding* and *send-without-receive* policies to the IDS-under-testing. In [22], scripts are used to generate traffic containing vulnerability exploits, so that attacks can be automatically launched against an IDS-under-testing. Similarly, in [32], an effort to automate the IDS testing process is proposed and in which *FTP*-based attack-free and malicious traffic streams are created and used to quantify attack detection and false positive rates. An off-line IDS testbed following a *training-then-testing* approach is presented in [17, 18] requiring a fixed network topology and not fully validated attacks [33]. A benchmark whose main objective is to capture the relationship between IDS performance and the intrinsic regularity in network traffic is discussed in [34].

A two-stage testing approach for establishing an IDS baseline behavior is discussed in [30]. In the first phase, the IDS is tested against simple attacks while during the second phase, the device is examined under complex and sustained attacks mixed with various types of synthetic background traffic. The methodology proposed in [35] follows a similar two-stage approach but it can use realistic background traffic derived from live networks. By using 27 common attacks and their variants created with evasive techniques, the testbed in [36] reveals that IDSs may detect less than 50% of malicious activities when the traffic intensity is more than 60% of the network bandwidth. This clearly demonstrates the necessity of testing IDSs under heavy traffic workloads. Similarly, evasion techniques are used to manipulate traffic before injecting into an IDS-under-testing in [13, 37], so that capabilities on the identification of stealthy attacks can be measured. The testbed in [14] is mainly designed to evaluate commercial IDSs for their architectures, ease of installation and attack coverage; tests on multiple commercial IDSs clearly show that detection rates deteriorate dramatically under heavy traffic workloads or evasive attacks.

All the above approaches and testbeds share in common the following characteristics: (a) *uni-directional-feeding* replay method is employed as IDSs are passive devices that maintain single access points in the network and are ‘blind’ to the direction of intercepted packets, (b) *send-without-receive* mechanism is used to handle traffic due to the fact that IDSs do not intervene and/or change the underlying traffic and (c) event logs from IDSs are mainly used to evaluate their behavior. Unfortunately, testing methodologies based on the above features are infeasible for IPS evaluation due to a number of reasons: first, packets that have source and destination network addresses within the same subnet are neither forwarded nor inspected by IPSs. Secondly, real-time IPS actions on identified malicious connections may change the characteristics of ongoing traffic [3]. This calls for testbeds to capture all IPS-emitted packets so that correlation with replayed packets is feasible and verification of the correctness of IPS counter-measures can be established. Lastly, IPS-testbeds should be able to independently verify the consistency between actions taken by IPSs and what is actually recorded on their logs. Discrepancies may reveal problems with IUT.

Recently, a few IDS testbeds have been reworked to help test IPSs in a meaningful way; for instance, `Tcpreplay` has been modified to replay traces bi-directionally by having both IP and MAC addresses of packets rewritten before injection into the IUT [19, 20]. Although such extensions make replayed packets IUT-forwardable, determining the direction of packet injection is not automated and does require manual intervention. In addition, extensions still fail to independently assess the correctness of IPS counter-actions. The trace-driven IPS testbed `Tomahawk` [15] statically modifies the content of routing and ARP tables of the test machine to conform to the environment in which the trace was captured. Although bi-directional-feeding and independent logging are in place, the derived test results entail only simple attack-blocking-rates [15]. We present the main features of

Tomahawk and Tcpreplay in Appendices 1 and 2, respectively. In [14], an IPS-testbed is introduced in which IUTs are subject to diverse traffic workloads and are assessed for reliability, availability, detection and blocking accuracy, as well as latency; stress-tests show that there is still a noticeable gap and delay between contemporary IPS attack coverage and real world attacks [14]. Moreover, evasion techniques remain effective against some IPSs and the performance of IPSs under heavy workloads suffers [14].

Penetration or pen tests use tool-generated attack traffic against targets in an 'active' way [38]. A penetration test typically involves an active analysis phase of the system under test for potential vulnerabilities that may result by its misconfiguration, hardware/software flaws and operational weaknesses followed by an attack phase [39]. Security vulnerabilities identified during penetration tests help assess the impact of successful attacks and develop defense strategies [38]. Tools including Nessus, NMap, and Metasploit are often used for penetration testing [40–42]. For instance, Metasploit can launch attacks with various shellcode payloads, and upon success, payloads are executed on the targeted systems [42]. Should systems under test be placed behind an IPS, penetration testing can be used to verify the effectiveness of the IPS in question. Although this appears to be a viable proposition, it does suffer from the drawback that applications under attack have to be also replicated in the testing environment, clearly, an expensive and occasionally an infeasible option. Even though attack tools can be directly used in IPS testbeds, it would be challenging to manage both intensity and period of the resulting attack traffic. Lastly, it is unrealistic to expect that an IPS testbed would feature a complete selection of attack tools in order to help conduct thorough and nearly complete tests. In [16], a *live* IPS-testbed in a production environment along with measurements for gauging the stability, false positives and forensic analysis capabilities of the IUT are discussed. However, such live testbeds lack in terms of test controllability and repeatability especially when it comes to traffic intensity and network latency. It is nearly impossible to manipulate attacks in live testbeds as far as their type, rate, period and intensity are concerned which is certainly a weakness. As simulation methods demonstrate excellent repeatability, controllability and comparability in test-case generation, evaluation procedure, and performance results, in this paper, we propose our IPS test framework based on a trace-driven simulation engine. We should point out however that trace-driven and live testing systems are complementary as IPSs should be first thoroughly tested in simulated testbeds before they move to production settings.

3. THE PROPOSED TESTBED PLATFORM

The IPS requirements for inline operation, switching/routing functionality and proactive real-time counter-measures on traffic necessitate a significant deviation from the design of conventional IDS-testbeds [15, 20] whose operation is based on *uni-directional-feeding* of packets from a single *NIC* and *send-without-receive* mechanisms [22, 23]. In this section, we introduce the salient features of our proposed IPS-testbed termed *IPS Evaluator*.

3.1. Design rationale and architecture for the *IPS Evaluator*

To facilitate the inline mode, an IPS has to maintain at least two network interfaces, so that it can splice into a network path and be able to intercept ongoing data flows as the IPS testbed model of Fig. 1 depicts. For simplicity, we assume that the IUT has exactly two network interfaces, *internal* and *external*; the former connects to the private network(s) being protected, while the latter connects to the outside world. Should test-machines 1 and 2 simulate a *victim* and an *attacker*, a valid network path can be established by attaching the victim and attacker to the internal and external interfaces of the IUT, respectively. In the resulting network path, bi-directional traffic may take place with one data stream traveling from victim to attacker and the second stream going the opposite direction.

IPS-testbed designs following the IDS-like model of Fig. 2—where dotted components and/or communication

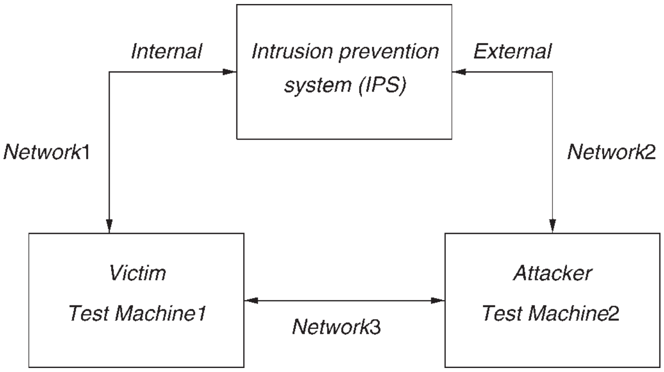

FIGURE 1. Trace-driven IPS testbed model.

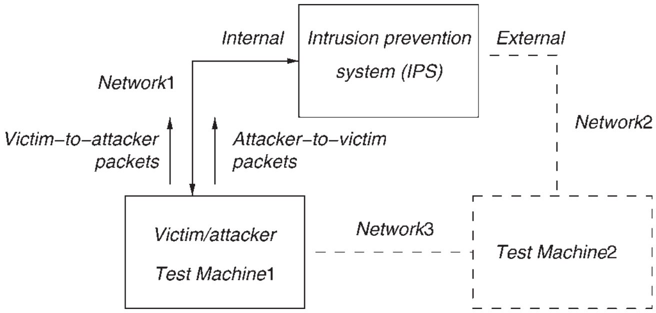
Internal
Intrusion prevention system (IPS)
External
Network1
Network2
Victim–to–attacker packets
Attacker–to–victim packets
Victim/attacker
Test Machine1
Network3
Test Machine2

FIGURE 2. An infeasible IPS testbed model.

channels do not really exist but they are provided for comparison with choices suggested in Fig. 1—are problematic and not a viable testbed option for the following reason: should the IUT operate as a switch, the IUT would forward packets according to its MAC-to-interface table. The latter is initially empty and over time gets populated by binding the source MAC address of each received packet with its arrival interface. If attacker-originated packets are fed into the IUT through its internal networks, the IUT associates the attackers' MAC addresses to its internal interface. Similarly, victims' MAC addresses are associated with the IUT's internal interface as all victim-to-attacker packets reach the IUT via its internal interface due to the *uni-directional-feeding* replay policy. The established MAC-to-interface mapping table leads the IUT of Fig. 2 to 'believe' that both attackers and victims reside in the same network segment. Hence, the IUT refuses to forward subsequent packets and foregoes any further security inspection. In case that an IPS predominantly functions as a router, its routing table has to be fully configured and consequently the IPS is aware of both internal and external networks. Any time, an attacker-to-victim packet originates from the internal network, the IUT is able to identify the incorrect origin with the help of its routing table and should not forward the packet.

IPS-testbeds should not be 'blind' to the direction of packets. To this effect, packets from traces should be grouped into two sets, attacker-and victim-initiated packets, and be injected into the IUTs from different directions based on their origin with a *bi-directional-feeding* policy. In this manner, replayed packets can be properly forwarded and be subjected to security inspection by the IUTs. In addition, IPS-testbeds are also expected to capture traffic due to:

- *pro-active behavior of IPSs*: streams containing traits of attacks may be dropped, malicious connections may be discontinued and possibly additional messages may be introduced such as *ICMP destination unreachable* or *TCP RESET*.
- *traffic normalization*: IUTs may remove protocol anomalies generated by evasion attacks or perform IP de-fragmentation before forwarding, rendering the outgoing traffic different from that injected.
- *network address translation (NAT)*: IPSs modify source IP addresses and ports of packets coming off the internal network before forwarding; similarly, IPSs re-map destination IP addresses and ports of packets arriving at its external port.
- *discrepancies between IPS actions and its logged events*: IPS testbeds have to record the IUTs' actions that are not actually delivered as claimed in their event logs to help resolve inconsistency analyses.

For these reasons, IPS-testbeds cannot possibly employ a *send-without-receive* packet replay method used by most IDS testbeds.

We could establish a viable IPS-testbed by using different test machines to simulate both attacker and victim following the blueprint of Fig. 1. With the help of partition techniques that we discuss in Section 4, packets in a trace can be grouped into P_{attacker} and P_{victim} sets based on their origin; those in P_{attacker} reach the IUT's external port via `Network2` and may be forwarded to `Network1` while those in P_{victim} travel in the opposite direction. The effectiveness of the IUT is evaluated by having the IPS-testbed check whether the replayed packets are equivalent to those reaching their destination. To honor the temporal features of the original traffic, the two test machines should coordinate their actions. This entails maintenance of transmission order and time gaps between packets as well as establishing that the IUT correctly forwards replayed packets, properly normalizes traffic and finally imposes the specified counter-measures on identified malicious connections. Additional communications between the test machines of Fig. 1 are required to carry out the above coordination. The separate communication link `Network3` of Fig. 1 helps diminish interference between replayed traffic and control messages. Nevertheless, such a dedicated link substantially increases both the testbed complexity and cost as additional *NIC*s are required for each test machine. Furthermore, required communications among test machines may adversely affect the testbed scalability especially when it comes to the stress tests as extra communications slow down traffic injection speeds and demand more test machines to saturate the IUT's bandwidth.

Figure 3 depicts our choice for the design of the *IPS Evaluator*-testbed. It avoids the extra link (i.e. `Network3`) and resorts to fast interprocess communications (IPCs) to simulate the communications between victim and attacker test machines. This proposed design is feasible provided that the single test machine features two different *NIC* interfaces controlled, respectively, by the now-co-located attacker(s) and victim(s).

3.2. A trace-driven simulation-engine for IPS testing

The high-level *IPS Evaluator* model of Figure 3 consists of a number of distinct modules including a *Traffic Partitioner*, a *Simulation Scheduler*, a *Sender*, a *Receiver*, and a *Behavior Arbitrator*. These modules are shown in Fig. 4. In order to feed a specified traffic trace into the IUT, the *Traffic Partitioner* first separates packets of the trace into two groups, P_{attacker} and P_{victim}. The former contains packets initiated by attackers while the latter holds packets from victims. Subsequently, the *Simulation Scheduler* constructs a replay plan based on the characteristics of the trace and specifications from tester. With the help of *Sender*, packets in groups P_{attacker} and P_{victim} are fed into the IUT's external and internal interfaces, respectively; IUT-forwarded packets are captured and stored by the module *Receiver*. The *Behavior Arbitrator* module observes and records the behavior of the IUTs and

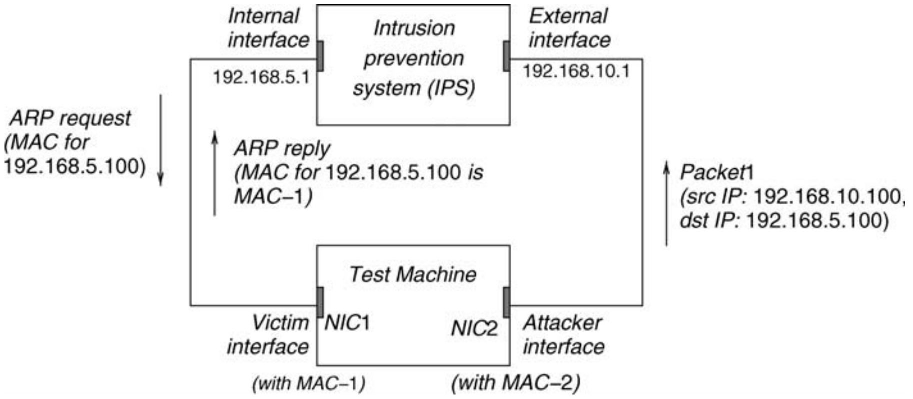
Internal interface
Intrusion prevention system (IPS)
External interface
192.168.5.1
192.168.10.1
ARP request (MAC for 192.168.5.100)
ARP reply (MAC for 192.168.5.100 is MAC–1)
Packet1 (src IP: 192.168.10.100, dst IP: 192.168.5.100)
Test Machine
Victim interface
NIC1
NIC2
Attacker interface
(with MAC–1)
(with MAC–2)

FIGURE 3. Trace-driven IPS testbed model with colocated attacker and victim.

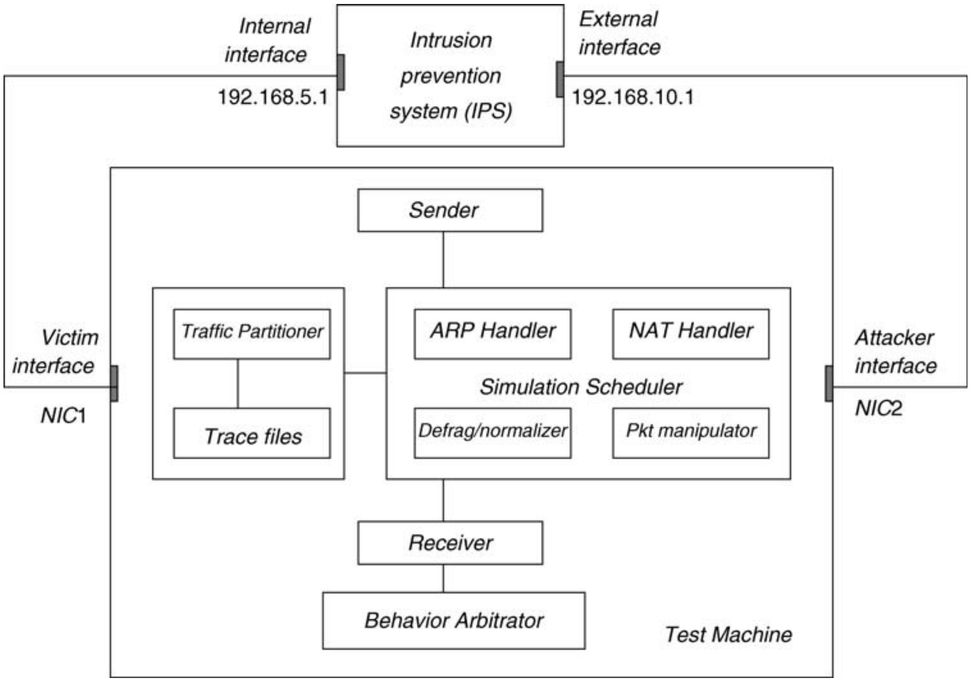

FIGURE 4. Components of the proposed *IPS Evaluator*.

finally delivers the evaluation report. In addition, the *Behavior Arbitrator* can also discover any discrepancies between the IUT event-log and the actions taken by the IPS on the underlying traffic. These differences often emanate from IPS design and/or implementation defects, occasional malfunctions as well as out-of-resource and/or heavy workload conditions. For instance, the IUT may state in its log that an attack has been blocked, but in fact the traffic containing the attack is still forwarded by the IUT and reaches its victim. Evidently, incorrect conclusions may be drawn if the IUT's own log records are exclusively used in the evaluation of its behavior.

Should the IUT detect a malicious incoming packet, it may drop it and log the event; a packet may be also dropped in light of network malfunctions and/or congestion. *IPS Evaluator* may retransmit lost packets a configurable number of times using a timer to trigger the retransmission mechanism. A packet is considered to be dropped by the IUT and not due to network congestion if it fails all retransmission attempts. The use of the retransmission mechanism in the testbed may cause the observation of the same attack by the IUT multiple times. The IUT may react differently to exploits delivered with various types of transportation mechanism. In *TCP*-based attacks, for example, a malicious packet and its likely retransmitted instances share the same *TCP* sequence numbers; thus, the IUTs should be able to recognize all such packets as part of a single attack instead of several independent exploits. For *UDP* and *ICMP*-based attacks, however, IUTs cannot distinguish an attack and its retransmissions. Consequently, the

IUT treats the attack and its retransmissions as isolated incidents. To enhance the flexibility of our testbed, we provide a user-configurable number of retransmissions max$_{\text{retrans}}$ for *TCP*, *UDP* and *ICMP* transmissions.

By default, *IPS Evaluator* respects the temporal characteristics of the trace including packet orders and their time gaps by adjusting its replay pace according to the timestamps of packets in the trace. However, our testbed can also be configured to replay a trace with an arbitrary rate (in packets or bits per second) instead of the original pace. Such a flexibility in replay speed is valuable when it comes to stress-testing. Clearly, measurements including throughput, average network latency and maximum number of concurrent connections reveal the IPSs capabilities under diverse and stress-related workloads. Although background traffic can be generated through the execution of attack-free applications in the testbed, it is very much desired in an IPS-testbed to have greater freedom when it comes to the traffic composition as far as the transport protocols used (TCP, UDP and/or *ICMP*) and the intensity of generated traffic streams are concerned. Our *IPS Evaluator* can create such workloads in a controlled manner by injecting both attack or *foreground* and attack-free or *background* traffic into the IUT through the mixing of multiple streams each replayed at different speed and varying ratio with the help of the testbed shown in Fig. 5; here, foreground and background traces, captured separately and stored in different files, are replayed by using multiple test machines.

Algorithm 1 depicts the main operations of our *IPS Evaluator* and helps derive test results by replaying a given traffic trace to an IUT in a bi-directional fashion. Based on the replay plan created by *Simulation Scheduler*, the *Sender* dispatches a set of packets using the *attacker* interface if the packet under processing is in P_{attacker} or *victim* port if the packet belongs to P_{victim}. Similarly, the component *Receiver* waits for packets forwarded by the IUT on either the *attacker* or *victim* interface. Our testbed conserves on communication costs by having *Sender* and *Receiver* exchange information for synchronization and coordination only via module *Simulation Scheduler*. During the replay process, the *IPS Evaluator* may rewrite certain protocol fields such as MAC and IP addresses on-the-fly in order to ensure conformance of the replayed packets with the IUT settings and the network configuration of the test environment; this rewriting is performed by function *AddressMap(P)* that we discuss in detail in Section 3.3. The *IPS evaluator* may also create event records when it detects the IUT's pro-active countermeasures that terminate ongoing sessions by dynamically generating *TCP RESET* or *ICMP destination unreachable* messages to either or both ends of the connection. Furthermore, to ensure that a received packet is indeed identical to what is replayed, the *IPS Evaluator* can be configured to compare not only packet header but also packet payload of IUT-forwarded messages against transmitted packets. To reduce computational overhead, it is typical to verify packet integrity by checking packet headers only

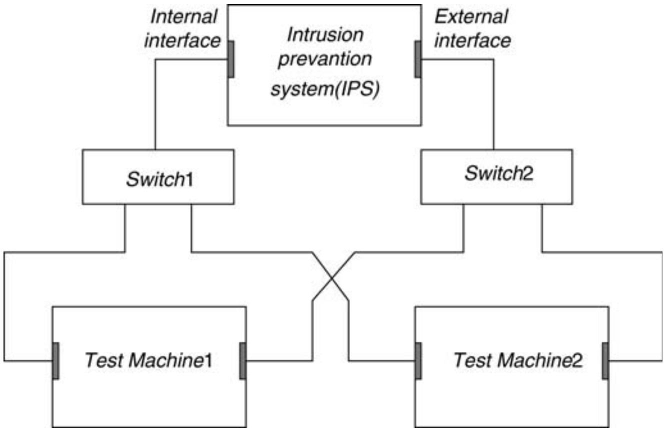

FIGURE 5. *IPS Evaluator* with multiple foreground/background traffic generators.

Algorithm 1. Operation of IPS Evaluator.

```
traffic trace is partitioned into $P_{\text{attacker}}$ and $P_{\text{victim}}$ by module Traffic Partitioner with Algorithms 5 and 6, which will be described in Section 4;
a replay plan is generated by the module Simulation Scheduler according to test specifications;
while (more unprocessed packet $P$ in the replay plan) do
    port ← attacker if $P$ is in $P_{attacker}(P)$; port ← victim otherwise;
    $P$ is processed with function AddressMap(P) (see discussion in Section 3.3); $P$ is sent out through port by component Sender at most max$_{\text{retrans}}$ times;
    while (there is packet $P'$ received by module Receiver) do
        invoke function PacketIntegrity($P$, $P'$), which will be described in Section 3.4, a test record is created if $P'$ is not identical to any transmitted message so far by comparing  packet header and/or payload;
        generate test record if $P'$ is TCP RESET OR ICMP unreachable packet that is not in the original trace;
    end while
end while
test results are generated by the module Behavior Arbitrator based on the records generated by the IPS-testbed
```

TABLE 1. Packets in the `Nimda` attack trace file.

#	Direction	Timestamp	TCP hdr/pld	Payload	Description
protocol: TCP; IP/port for attacker (A): 10.80.8.183/32872; IP/port for victim (V): 10.80.8.221/80					
1	$A \rightarrow V$	0.000000	40/0	(SYN)	Request
2	$V \rightarrow A$	0.000223	40/0	(SYN|ACK)	Reply
3	$A \rightarrow V$	0.000631	32/0	(ACK)	Confirm
4	$A \rightarrow V$	5.514226	32/64	GET /scripts/..%255c../winnt/system32/cmd.exe? /c+dir HTTP/1.1	Attack
5	$V \rightarrow A$	5.514313	32/0	(ACK)	Acknowledge
6	$A \rightarrow V$	6.137619	32/2	|0D 0A|	Attack
7	$V \rightarrow A$	6.137692	32/0	(ACK)	Ack
8	$V \rightarrow A$	6.138571	32/191	HTTP/1.1 200 OK|0D 0A|Server: Microsoft-IIS /5.0|0D 0A|Date: Fri, 11. . .	reply
9	$A \rightarrow V$	6.138814	32/0	(ACK)	Acknowledge
10	$V \rightarrow A$	6.156986	32/36	Directory of c:/inetpub/scripts|OD 0A 0D 0A|	Directory
11	$A \rightarrow V$	6.174736	32/0	(ACK)	Acknowledge
12	$V \rightarrow A$	6.199095	32/40	10/10/2002 02:24p <DIR>.	Content of dir

when it comes to background traffic. We discuss the procedure for determining packet integrity in Section 3.4.

Throughout this paper, we use a trace of the `Nimda` attack whose packets appear in Table 1 as a running example. This attack exploits security holes in products such as *internet information service (MS-IIS)*. Once a machine is infected, `Nimda` attempts to replicate itself by probing other *IIS* servers through multiple mechanisms including the *extended unicode directory traversal vulnerability* discussed in Section 6.1. Here, the attacker is located at host with IP address 10.80.8.183 and *TCP* port 32872, while the victim is a Web server with *IP* address 10.80.8.221 and *TCP* port 80. The first three packets carry out the initial *TCP* three-way-handshake procedure between the attacker and victim. Packet 4, originating from the attacker and with TCP payload of 64 bytes (see Column *'TCP hdr/pld'*), is an *HTTP* request attempting to activate program *'cmd.exe'* on victim's system. When this packet reaches the victim Web server, the file name in the request—substring preceding '?'—is first decoded by *IIS* based on UTF-8 format for security inspection. However, a flaw in *IIS* mistakenly decodes the file-name part again when the parameter part is handled [43], forcing the execution of */winnt/system32/cmd.exe* with parameter */c dir* offering a backdoor to attackers with full control of the victim machine.

IPSs typically detect `Nimda` by searching for the telltale pattern *'cmd.exe'* in traffic. By configuring an IUT to block the `Nimda` attack and with the help of Algorithm 1, our *IPS Evaluator* can capture the IUT's behavior. When processing the trace of Table 1, Algorithm 1 forms two packet groups: P_{attacker}—(1, 3, 4,. . .) and $P_{\text{victim}} = (2, 5, 7,\ldots)$. The *simulation scheduler*'s replay scheme preserves both order and interarrival times of the trace packets. For example, the *IPS Evaluator* respects the long time-gap between packets 3 and 4. The IUT is deemed effective if packet 4—that contains the pattern in question—is *Sender*-transmitted $\max_{\text{retrans}}$ times and still fails to reach the *Receiver* module.

3.3. Addressing and routing issues in the proposed IPS-testbed

An IPS may function in either *transparent* (i.e. as a switch) or *routing* mode (i.e. as a router). When in *transparent* mode, the IPS establishes a map between the source MAC-address of every incoming packet and its arrival interface, and forwards the packet based on its destination MAC-address with the help of the established map. If no pertinent entry is found in the map, the IPS floods the packet to all its interfaces except the one at which the arrival occurred. If the source and destination MAC addresses of an incoming packet associate with the same interface, the IPS declines to forward the message and carries out no security inspection. An IPS in routing mode maintains a routing table based on protocols such as RIP and ARP that helps map IP to MAC addresses and refuses packet forwarding if no route entry is found.

For a packet to be forwarded correctly by the IUT in an IPS-testbed, its source/destination IP and MAC addresses should conform those of the test environment. For instance, when in routing mode, the IUT internal and external interfaces should belong to different subnets. Without the help of other routers, the IPS can only handle one-hop routing, requiring that the source and destination subnets of any incoming packet be the same as its arrival and departure interfaces on the IPS, respectively. Apparently, the traffic of Table 1 cannot be forwarded by the IUT in routing mode if the test environment is configured according to Figure 3; here, the IUT internal and external interfaces belong to subnets 192.168.5.0 and 192.168.10.0, respectively, but both victim and attacker of the trace reside on subnet 10.80.8.0 if netmask 255.255.255.0 is used. Hence, it is necessary to rewrite MAC and IP addresses of every packet before the packet is injected into the IUT:

(i) Should a static method be used, MAC and IP addresses are changed directly in traces, rendering the resulting traces useless in other testbeds with different network topologies and configurations. Such a static method is also time-consuming as separate traces should be generated for different test modes (switching or routing) and different network topologies. Consequently in addition to support static methods, our *IPS Evaluator* can also be configured to employ dynamic addressing and routing methods.

(ii) In a dynamic addressing scheme, the *IPS Evaluator* maintains two non-overlapped IP address pools, A_{attacker} and A_{victim}, to store IP addresses exclusively used by the packet groups P_{attacker} and P_{victim}; addresses in these two pools feature the same subnets to the IUT external and internal interfaces, respectively. Two mapping tables, M_{attacker} and M_{victim}, store the associations between source IP addresses in P_{attacker} and A_{attacker}, and source IP addresses in P_{victim} and A_{victim} respectively. Any time, the module *Sender* replays a packet P, it first examines P's source IP address P_{sip}. If P belongs to P_{attacker}, the *Sender* queries table M_{attacker} regarding P_{sip}. If no such entry exists, the *Sender* acquires an IP address, denoted as A, from A_{attacker} and inserts the tuple $<P_{\text{sip}}, A>$ into M_{attacker}. Otherwise, our testbed locates an entry for P_{sip} in M_{attacker}, denoted as $<P_{\text{sip}}, A>$. Subsequently, the *Sender* replaces P's source IP with A. The *Sender* applies the same operation to the destination IP address of P by using M_{victim} and A_{victim} instead.

In *switching* mode, the IUT forwards packets based on their destination MAC addresses. To ensure that a replayed packet P is correctly forwarded, our testbed replaces the source and destination MAC addresses of P with those of its attacker and victim interfaces, respectively, if P is in the P_{attacker} group; similar replacement is imposed on packets in P_{victim} as well. In *routing* mode, the IUT forwards packets according to their destination IP addresses with the help of its routing and ARP tables. As IP addresses of replayed packets are from A_{attacker} or A_{victim} and no physical device assumes such IP addresses in the test environment, it is obvious that no corresponding entries exist in the IUT's routing and ARP tables. Therefore, the IUT sends out an ARP request for each IP address that has no entry in its ARP table. In a clear deviation from static handling, the component *ARP Handler* of our *IPS Evaluator* creates replies to IUT-issued ARP requests for IP addresses in A_{attacker} or A_{victim}. To this effect, the IUT establishes its ARP table dynamically.

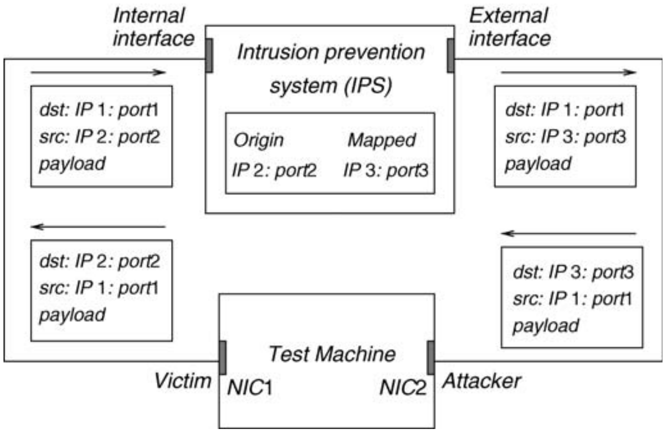

FIGURE 6. An IUT with the functionality of network addressing translation (NAT).

NAT may prove to be critical in the operation of IPSs as it allows for the mapping between unregistered/private and registered/routable IP addresses either statically or

Algorithm 2. Operation of *AddressMap(P)* within our IPS Evaluator.

```
$P_{sip}$ and $P_{dip}$ are the source/destination IP addresses of $P$; $M_{attacher}$, initially empty, maintains associations between addresses in $A_{attacker}$ and
    source addresses of $P_{attacker}$; $M_{victim}$, initially empty, maintains associations between addresses in $A_{victim}$ and source addresses of $P_{victim}$;
$IP_{attacker} \leftarrow P_{sip}$ and $IP_{victim} \leftarrow P_{dip}$ if $P$ belongs to $P_{attacker}$; otherwise, $IP_{attacker} \leftarrow P_{dip}$ and $IP_{victim} \leftarrow P_{sip}$;
 $A \leftarrow$ (search result in $M_{attacher}$ with key $IP_{attacker}$); $V \leftarrow$ (search result in $M_{victim}$ with key $IP_{victim}$);
 if ($A$ is empty) then
   $A \leftarrow$ (next available IP address in pool $A_{attacker}$); entry ($IP_{attacker}$, $A$) is inserted into $M_{attacker}$;
 end if
if ($V$ is empty) then
    $V \leftarrow$ (next available IP address in pool $A_{victim}$); entry ($IP_{victim}$, $V$) is inserted into $M_{victim}$;
end if
$IP_{attacker}$ and $IP_{victim}$ of $P$ are replaced with $A$ and, $V$ respectively; $P_{dp}$ is the destination port of $P$;
if($P$ belongs to group $P_{attacker}$) AND (pair $V$, $P_{dp}$ is, in $M_{nat}$) then
let pair $V'$, $P'_{dp}$ be the pair associated with $V$, $P_{dp}$ in $M_{nat}$; replace $V$ and $P_{dp}$ with $V'$ and $P'_{dp}$, respectively;
end if
```

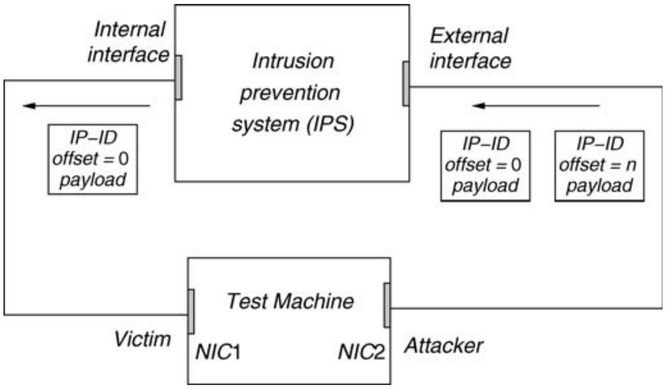

FIGURE 7. An IUT with the functionality of ID de-fragmentation.

dynamically [11, 44]. Multiple unregistered IP addresses can be mapped to different routable addresses or a single registered IP address but with different ports as shown in Fig. 6. Here, the source IP and port <*IP2:port2*> of a victim-originating packet is rewritten by the IPS as <*IP3:port3*> before forwarding to the outside world. Similarly, the destination IP and port of its reply packet arriving externally is mapped back from <*IP3:port3*> to <*IP2:port2*> with the help of mapping table built inside the IPS. To handle NAT, the *IPS Evaluator* uses a table M_{nat} to establish the mappings between the <*IP,port*> pair assigned to a packet by the IUT and its corresponding pair assigned by our engine.

Function *AddressMap(P)* of Algorithm 2 outlines the key points in rewriting the source/destination addresses of a packet *P* before this packet is transmitted. *AddressMap(P)* is invoked by the *Sender* in Algorithm 1.

3.4. Handling IP Fragmentation in *IPS Evaluator*

To normalize traffic and provide stateful inspection service, IPSs may defragment received IP packets before such fragments are forwarded. This occurs when the size of an IP packet is larger than the *maximum segment size* (*MSS*) supported by the underlying link, for instance, *MSS* is 1518 bytes on an Ethernet network. Nowadays, IP fragmentation is routinely used by evasion attacks and exploits crafted by tools such as `fragroute` [13, 20]. In addition, the generated IP fragments can be shuffled, overlapped, and/or duplicated before transmission. To overcome such evasion exploits, many IPSs temporarily stage all IP fragments with the same IP identifier (i.e. `IP-ID`) before forwarding; once no attack or protocol anomaly is detected in staged fragments, the stored IP frames can be forwarded. In actually carrying out the forwarding, an IPS may just assemble the IP fragments together and refragment them following its own scheme should aggregate frames be larger in size than *MSS*. For instance, in Fig. 7, two IP fragments with the same `IP-ID` arrive at the external port of the IUT, but only their aggregation (i.e. a single complete IP frame) is forwarded to the IUT internal port. In this context, the functionality of defragmentation in IPSs may change the characteristics of injected traffic in terms of packet numbers, sizes and arrival times. Furthermore, IP fragmentation also makes it difficult to verify the integrity of injected packet in order to determine whether a received fragment is what has been actually sent out by the testbed as the IUT may refragment the IP packet anew on its own.

A key concern for packet integrity checking is to correctly demark the first and last fragments of every IP frame in a traffic trace. For this, our *IPS Evaluator* clusters replayed IP fragments according to their protocol field `IP-ID` in IP headers. Once a packet is replayed by the *Sender*, it is stored and re-assembled with replayed fragments having the same `IP-ID` with the assistance of the *Defrag/Normalizer* component. Our *IPS Evaluator* uses a hash table and an interval-tree [45] to organize all IP fragments as depicted in Fig. 8. Similarly, when a packet *P* is received by the module *Receiver*, its `IP fragment`

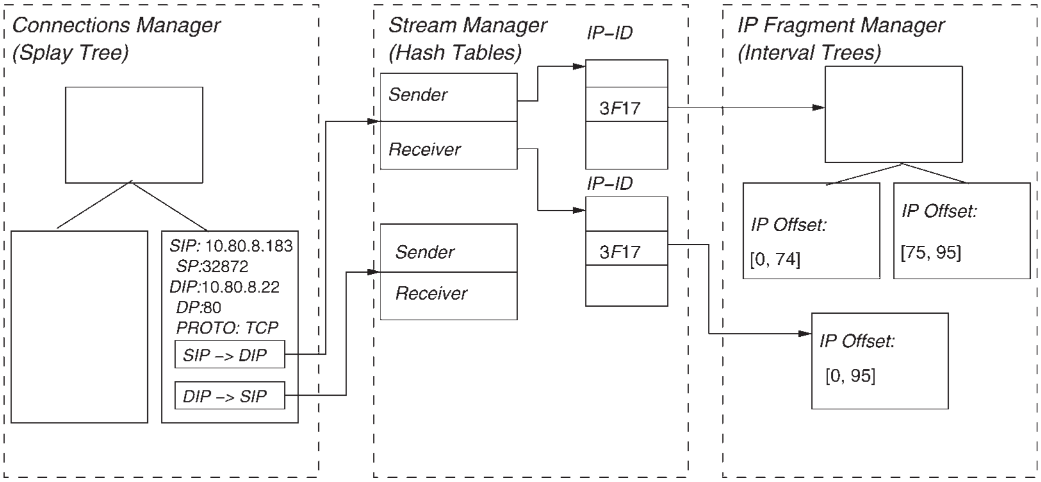

FIGURE 8. IP fragments organized with hash, binary tree and interval tree.

Algorithm 3. Procedure *PacketIntegrity* (P, P') invoked by *Defrag/Normalizer*.

```
$P$ is a packet replayed by Sender and $P'$ is a packet received by Receiver; $M_{\text{nat}}$, initially empty, maintains associations between pairs IP/port assigned by IPS/NAT and our testbed; $P_{\text{sip}}/P_{\text{sp}}$ and $P_{\text{dip}}/P_{\text{dp}}$ are source and destination IP/port of $P$; $P'_{\text{sip}}/P'_{\text{sp}}$ and $P'_{\text{dip}}/P'_{\text{dp}}$ are source and destination IP/port of $P'$;
if($P$ belongs to group $P_{\text{victim}}$) AND (tuple formed by pairs '$P_{\text{sip}}$, $P_{\text{sp}}$' and '$P'_{\text{sip}}$, $P'_{\text{sp}}$' is in $M_{\text{nat}}$) then
    replace $P'_{\text{sip}}$ and $P'_{\text{sp}}$ of $P'$ with $P_{\text{sip}}$ and $P_{\text{sp}}$;
else if ($P$ belongs to group $P_{\text{attacker}}$) AND (tuple formed by pairs '$P_{\text{dip}}$, $P_{\text{dp}}$' and '$P'_{\text{dip}}$, $P'_{\text{dp}}$' is in $M_{\text{nat}}$) then
    replace $P'_{\text{dip}}$ and $P'_{\text{dp}}$ of $P'$ with $P_{\text{dip}}$ and $P_{\text{dp}}$;
end if
$S$ is $P$'s session returned by Connection Manager of Figure 8 with tuple $<P_{\text{sip}}, P_{\text{sp}}, P_{\text{dip}}, P_{\text{dp}}, P_{\text{protocol}}>$;
if ($P$ or $P'$ is fragmented packet) then
    obtain the stream corresponding to '$P_{\text{sip}} \rightarrow P_{\text{dip}}$' with the help of Stream Manager of Figure 8; insert $P$ and $P'$ into interval trees associated with their IP-ID by IP Fragment Manager of Figure 8;
    return UNDECIDED if $P$ is not the last fragment in IP fragments with the same IP-ID of the given trace;
    $Q \leftarrow$ (de-fragmented IP frame formed by all fragments with the same IP-ID as $P$); $Q' \leftarrow$ (de-fragmented IP frame formed by all fragments with the same IP-ID as $P'$);
else
    $Q \leftarrow P$; $Q' \leftarrow P'$;
end if
return DIFFERENT if any specified protocol fields or contents assume different values in $Q$ and $Q'$; otherwise, return IDENTICAL;
```

bit in IP header is checked to determine whether it is fragmented; if not, it is safe for the *IPS Evaluator* to perform integrity inspection at packet level. If the received P is indeed a fragment, it is stored and re-assembled with other received fragments having the same `IP-ID`. The packet integrity inspection is conducted on the aggregated IP frame instead of individual IP fragments.

Overlapping fragments make the IP defragmentation process complicated. Two or more IP fragments are considered overlapping if some of their IP payloads share the same IP fragment offsets. Ambiguity occurs if overlapped fragments bear different contents in their overlapping parts. A number of IPSs use the most recently received fragments or *favor-new* in the final aggregation while others use the earliest arrival packets or *favor-old* policy. Our framework can be configured to perform either *favor-new* or *favor-old* IP defragmentation. Algorithm 3 outlines the key functionalities of *PacketIntegrity*(P, P') which verifies the identity of packets P and P' with respect to their protocol headers and/or contents.

By applying IP fragmentation to the traffic of Table 1 with the command *ip_frag 75* discussed in Section 4.3, we obtain an entirely different packet stream shown in Table 2; here, every original packet with IP payload larger than 75 bytes has been fragmented into IP frames with smaller payloads. For instance, packet 4 of Table 1 is split into two pieces: the first with IP payload 75 bytes and total frame size 109 bytes (including 14-byte Ethernet header and 20-byte IP header), while the second with IP payload 21 bytes. Packets 4 and 5 of Table 2 reflect the outcome of the IP fragmentation. When this traffic is replayed by our *IPS Evaluator*, the *Simulation Scheduler* instructs the *Defrag/Normalizer* to conduct the integrity check only after both packets 4 and 5 of Table 2 have been replayed by *Sender* and received by *Receiver*. Right after packet 5 is received, the data structure maintained by *IPS Evaluator* with the help of Algorithm 3 has the status shown in Fig. 8. We should clarify that the TCP payload of packet 4 of Table 1 after its IP fragmentation is divided into two packets; the first carries the substring upto *cmd.e* and the second the remaining command. This attack is expected to be missed by IPSs that cannot conduct IP defragmentation but simply scan for the pattern *cmd.exe* in every IP packet.

4. TEST-CASE GENERATION AND MANIPULATION OF TRACES FOR IPS-TESTING

Traces that can be used in IPS-testing such as those available from the *MIT's Lincoln Laboratory*, are heavily influenced by network topologies, host-addresses, subnet masks and aggregation of streams from different time periods [17]. The volume of the traces is also significant requiring in excess of a few hundred MBytes for just one hour traffic. Our own analysis of these data sets pointed out that networks in many traces essentially form a mesh topology. Hence, there is not a single location to deploy an IPS-under-testing that could observe all communications. Such mesh topologies that emerge from traces complicate issues pertinent to the *bi-directional replay* nature of IPS-testbeds. In addition, the ever increasing number of reported vulnerabilities—15107 upto 2005 according to `common vulnerabilities and exposures (CVE)` [46]— in conjunction with specific combinations of OSs, services and applications needed for exploits to occur make it impractical to generate all attack traces in a single network environment or testbed. It is simply too time-consuming to

TABLE 2. IP fragmented traffic for Nimda trace (Table 1).

#	Direction	IP-ID	TCP hdr/ply	Payload	Description
Protocol: TCP; IP/port for attacker (A): 10.80.8.183/32872; IP/port for victim (V): 10.80.8.221/80					
1	A → V	3F15	40/0	(SYN)	Attacker request
2	V → A	0000	40/0	(SYN|ACK)	Victim ack
3	A → V	3F16	32/0	(ACK)	Attacker confirm
4	A → V	3F17	32/43	GET /scripts/..%252f../winnt/system32/cmd.e	First part of attack in URL
5	A → V	3F17	0/21	Xe?/c+dir HTTP/1.1	Second half of attack in URL
6	V → A	0D65	32/0	(ACK)	Victim
7	A → V	3F18	32/2	|0D 0A|	Attacker
8	V → A	0D66	32/0	(ACK)	Victim
9	V → A	0D67	32/43	HTTP/1.1 200 OK|0D 0A|Server: . . .	First IP fragment
10	V → A	0D67	0/75	Date: Fri, 11 Oct 2002 19:37:45 GMT . . .	Second IP fragment
11	V → A	0D67	0/41	. . .Volume Serial Number is E802-9963 . . .	Third IP fragment
12	A → V	3F19	32/0	(ACK)	Ack
13	V → A	0D68	32/36	Directory of c:/inetpub/scripts|0D 0A 0D 0A|	Returned directory
14	A → V	3F1A	32/0	(ACK)	Ack
15	V → A	0D69	32/40	10/10/2002 02:24p <DIR>.	Content of directory

reconstruct every attack scenario in order to capture the resulting traffic. In addition, expecting that all attack tools are available for IPS-testing is not feasible. Lastly, attack tools hardly provide the flexibility for manipulating the intensity and mixture of needed traffic streams required for effective IPS testing. Thus, it is typical for a testbed to obtain traffic traces captured and/or generated in diverse network topologies and configurations. Regardless of the origin and type of a trace, it is imperative that the *IPS Evaluator* can effectively distinguish traffic coming off attackers and victims. To achieve this objective, our simulation-engine automatically partitions packets in a traffic trace into two parts based on their origin—P_{attacker} and P_{victim}—and dynamically rewrites MAC and IP addresses as needed when a trace is replayed. It is also critical that our simulation-engine provides traffic manipulation operations to shape replayed traffic so that the resulting data stream possesses desired characteristics. The above two issues are handled by the *Traffic Partitioner* and *Packet Manipulator* components of Fig. 4 and are described in detail in the following sections.

4.1. Partitioning traffic traces without constraints for IPS-testing

As every packet P in a trace maintains a source and a destination IP address denoted as P_{sip} and P_{dip}, the trace can be treated as a

Algorithm 4. Brute-force method to partition vertices of a graph/trace $G(V,E)$ into two groups.

```
$W_{\max}$ is the maximum weight among the partitions for $G(V,E)$ so far and is initially zero; $S_{\max}$ holds all partitions with weight of $W_{\max}$;
$M_{\text{red}}$ is the number of vertices with color red and is initialized to be 1;
while ($M_{\text{red}}$ is less than $|V|$) do
  compute all possible combinations of $M_{\text{red}}$ from $|V|$ (i.e. $Cn\ k$ where $n = |V|$and $k = |M_{\text{red}}|$); results are stored in Set $C$;
  while ($C$ is not empty) do
    remove head element of $C$ and put it into $G_{\text{red}}$; $G_{\text{black}} \leftarrow (V - G_{\text{red}})$;
    compute $w = \sum(v_r \in G_{\text{red}}, v_b \in G_{\text{black}})\ w(v_r, v_b)$;
    if ($w > W_{\max}$) then
      $W_{\max} \leftarrow w$; $S_{\max} \leftarrow (G_{\text{red}}, G_{\text{black}})$;
    else if ($w = W_{\max}$) then
      insert ($G_{\text{red}}$, $G_{\text{black}}$) into $S_{\max}$;
    end if
  end while
  $M_{\text{red}} \leftarrow (M_{\text{red}} + 1)$;
end while
all partitions in $S_{\max}$ have maximum bichromatic edges of $W_{\max}$;
```

graph $G(V, E)$, should P_{sip} and P_{dip} represent vertices in $G(V, E)$. The edge from P_{sip} to P_{dip} reflects the flow of packets in this direction; the edge's weight w can be the number of packets traveling along this route. In bi–directional traffic, the graph maintains the path from P_{dip} to P_{sip} as well. If a trace exclusively consists of attacker/victim traffic such as that of `Nimda` in Table 1, its corresponding graph $G(V, E)$ should be bipartite and its vertices could be covered with two colors. If we use a depth-first-search (DFS) method to color the bipartite [45], the algorithmic complexity is O(|V| + |E|), where |V| and |E| are the numbers of vertices and edges in G. If G turns out to be non-bipartite, then some packets are exchanged among attackers (or victims) only and clearly are not IUT-forwardable. To reduce the number of such un-forwardable packets so that the IUT is forced to perform security inspections on as many packets as possible, we try to bipartite G by removing a minimum number of its edges. In particular, for an undirected graph $G(V, E)$ with weight function w: $E \rightarrow N$, where N is a set of natural numbers, a two-color assignment c of G is defined as c: $V \rightarrow$ (*red,black*). Given that an edge is 'monochromatic' if its two end points have the same color, we seek a color assignment c with the minimum weight of monochromatic edges $\sum_{(v_1,v_2)\in E:c(v_1)=c(v_2)} w(v_1, v_2)$. The problem at hand is a special case of the *minimum edge deletion K-partition* problem with $K = 2$ and is known to be not only NP-complete, but also very difficult to find a polynomialtime approximation scheme with approximation accuracy guarantee [47, 48].

The straightforward method to tackle the problem at hand is to enumerate all possible bi-partitions of vertices in the given graph, compute the number of monochromatic edges for each partition, and find the partitions with minimum number of monochromatic edges. Algorithm 4 depicts such a brute-force method. Suppose that the number of vertices in the specified graph $G(V, E)$ is $|V| = n$ and the vertices are grouped into G_{red} and G_{black} with sizes of $|G_{\mathrm{red}}| = n_{\mathrm{r}}$ and $|G_{\mathrm{black}}| = n_{\mathrm{b}}$, respectively. Clearly, the number of vertices n_{r} in group G_{red} can be 1, 2, ..., $(n - 1)$, and for a particular n_{r} $(1 \leq n_{\mathrm{r}} < n)$, the number of all possible combinations of n_{r} from n is ${C_{n_r}}^{n}$. It can be derived that the total number of partitions is $\sum_{n_r=1}^{(n-1)}(C^n_{n_r}) = \sum_{n_r} = \sum_{n_r=0}^{n}(C^n_{n_r}) - 2 = 2^n - 2$. Therefore, the computational complexity of Algorithm 4 is O (2^n).

As the $G(V, E)$ corresponding to the `Nimda` attack of Table 1 has only two vertices, it is trivial to obtain its two partitions with Algorithm 4, (red: 10.80.8.183, black: 10.80.8.221) or(red: 10.80.8.221, black: 10.80.8.183). Table 3 shows a partial trace of traffic with more complicated network topology being generated by the `Cyberkit` attack tool, which integrates network services including *ping*, *traceroute, finger*,and *whois* and helps in conducting network reconnaissance [49]. For instance, by probing a network with `Cyberkit`-created *ICMP ECHO REQUEST* messages,an attacker can 'fingerprint' whether the targeted system or network are mis-configured and/or expose vulnerabilities [46]. IPSs may identify the *ICMP ECHO REQUEST* messages in question by detecting a long string of characters |AA| in the payload of such messages. The latter is feasible for example through the use of *Snort-Inline* signature *sid-483* which exploits such a telltale pattern. For each pair of P_{sip} and P_{dip}, Table 3 shows the number of packets traveling and indicates whether an attack is contained. Figure 9 depicts the undirected graph constructed from the table in question; every node corresponds to an IP address and the weight over the edge is the total number of packets exchanged between the two end-nodes regardless of their directions. In this regard, the weight of the edge between 67.115.180.150 and 67.117.243.205 is 18 (i.e. 10+8). The existence of cycles with odd number of edges renders the graph of Fig. 9 non-bipartite.One such cycle is formed by nodes 67.117.243.205, 67.115.180.150 and 67.117.243.204.

By applying Algorithm 4 to the graph of Fig. 9, we can obtain that, among all possible bi-partitioning schemes of the graph, four partitions achieve the minimum weighted sum of monochromatic edges (i.e. 6 packets); two of them are (red: 67.117.243.201, 67.117.243.205, 67.117.44.225; black: 67.117.243.207, 67.116.219.220, 67.115.180.150, 67.119.190.203, 67.117.243.204, 67.117.14.146) shown in Fig. 10 and (red: 67.117.243.201, 67.117.243.205; black: 67.117.243.207, 67.116.219.220, 67.115.180.150, 67.119.190.203, 67.117.44.225, 67.117.243.204, 67.117.14.146); two more

TABLE 3. Cyberkit-generated traffic exploiting network vulnerabilities

P_{sip}	P_{dip}	Number of pockets	Attacks	P_{sip}	P_{dip}	Number pockets	Attacks
67.117.243.201	67.117.243.207	3		67.117.243.205	67.117.243.207	4	
67.116.219.220	67.117.243.205	1		67.117.243.205	67.116.219.220	1	
67.115.180.150	67.117.243.205	10	yes	67.117.243.205	67.115.180.150	8	
67.119.190.203	67.117.243.205	2		67.117.243.205	67.119.190.203	2	
67.117.44.225	67.117.243.205	1		67.117.243.205	67.117.44.225	1	
67.119.190.203	67.117.243.204	1		67.117.243.204	67.119.190.203	1	Yes
67.117.44.225	67.117.243.204	1		67.117.243.204	67.117.44.225	1	
67.115.180.150	67.117.243.204	1		67.117.243.204	67.115.180.150	1	
67.117.14.146	67.117.243.205	7		67.117.243.205	67.117.14.146	8	
67.117.243.204	67.117.243.205	4		67.117.243.205	67.117.243.204	3	

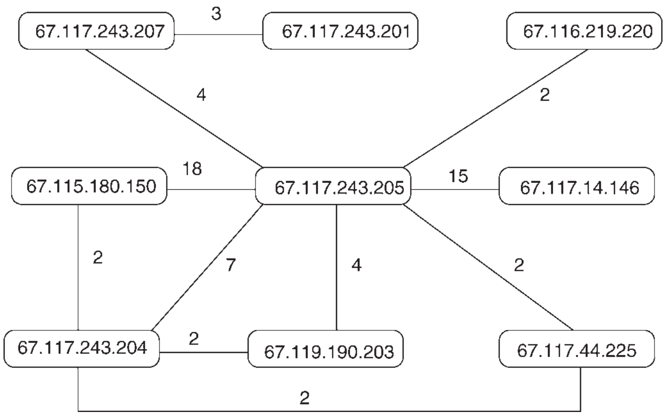

FIGURE 9. `Cyberkit` trace graph-representation.

partitions can be materialized by exchanging the roles of colors. Due to the fact that minimizing the weighted sum of monochromatic edges is equivalent to maximizing the weighted sum of bichromatic edges, we derive the maximum weight of bichromatic edges to be 55 (in packets) in the above optimal conditions.

Although Algorithm 4 is only viable for small graphs, we use it as a baseline for comparison with the approximate algorithm that we introduce later to limit the number of monochromatic edges. Our heuristic algorithm works as follows: once we ensure that a graph is non-bipartite, we sort vertices of G by decreasing order of their degrees (i.e. number of edges incident to a vertex) and place them into a queue Q. Initially, all vertices of G are set to *UNCOLORED*; ultimately, they are to be marked as *RED* or *BLACK*. For each vertex u in Q, we assign it an unused color if such a color is available. Otherwise, its neighbors—vertices with edges to u—are examined and two weighted sums s_r and s_b are computed as: $s_r = \sum_{((u,v)\in E:\, c(v)=red)} w(u,v)$, $s_b = \sum_{((u,v)\in E:\, c(v)=black)} w(u,v)$. Vertex u gets a *RED* color if $s_r < s_b$, and *BLACK* otherwise. Such a color assignment strategy attempts to generate minimum number of monochromatic edges in the neighborhood of vertex u. Once all vertices have been processed, we split them up in *RED* and *BLACK* groups; one group is designated as the *attackers* and the other as the *victims*. Algorithm 5 outlines our approach for partitioning a trace using a two coloring scheme. The overall complexity of Algorithm 5 is

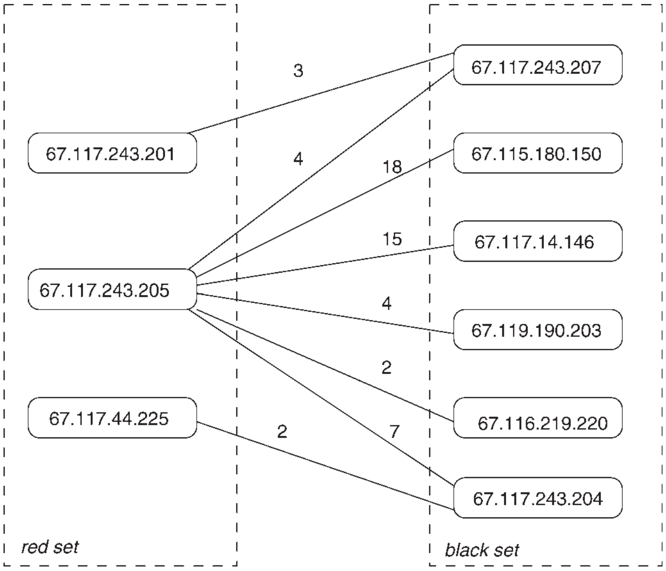

FIGURE 10. Bipartite of Fig. 9 with minimum monochromatic edges.

Algorithm 5. Partitioning vertices of a graph/trace $G(V,E)$ into two groups

```
vertices of $G$ are sorted according to their non-increasing orders of degrees and the results are put into set $A$;
for (each element u in $A$) do
  visit[$u$] $\leftarrow$ UNVISITED; partition[$u$] $\leftarrow$ UNCOLORED;
end for
put the first element $u$ of $A$ into a queue $Q$; visit[$u$] $\leftarrow$ VISITING; partition[$u$] $\leftarrow$ RED;
while ($Q$ is not empty) do
  remove first element $u$ from $Q$; two weighted sums $s_r$ and $s_b$ are initialized to be zero;
  for (each neighbor $v$ of vertex $u$) do
    $s_r$ +=(weight of edge $(u, v)$) if (partition[$v$] = RED); $s_b$ +=(weight of edge $(u, v)$) if (partition[$v$] = BLACK);
    if (visit[$v$] is UNVISITED) then
      visit[$v$] $\leftarrow$ VISITING and push $v$ into queue $Q$;
    end if
  end for
  if (partition[$u$] = UNCOLORED) then
    partition[$u$] $\leftarrow$ RED if ($s_r < s_b$) and partition[$u$] $\leftarrow$ BLACK otherwise; visit[$u$] $\leftarrow$ VISITED;
  end if
end while
$V$ of $G(V,E)$ are split into groups red and black with color RED and BLACK respectively;
```

$O(|V|\ \log\ (|V|) + |E|)$ with the sorting operation having complexity $O(|V|\ \log\ (|V|)$ and the rest of the operations $O(|V| + |E|)$.

By applying Algorithm 5 to the graph of Figure 9, we obtain that nodes 67.117.243.201, 67.117.243.205 and 67.117.243.204 form one group while the rest are in the second group as depicted in Fig. 12. The sum of the monochromatic edges in the resulting partition is 7 (in packets). The bichromatic edges can be computed to be 54 (in packets), which is within 2% of the optimal value (i.e. 55 packets found by brute-force method in Algorithm 4). With this partitioning in place, packets between 67.117.243.205 and 67.117.243.204 are not replayed by our engine as both vertices are within the same partition. In contrast, communications between 67.115.180.150 and 67.117.243.205 as well as 67.119.190.203 and 67.117.243.204, which contain malicious attacks, are injected into the IUT. It is worth pointing out that although node 67.117.243.207 shares the same subnet with nodes 67.117.243.201, 67.117.243.204 and 67.117.243.205 if subnet mask 255.255.255.0 is in use, they fall into different groups and so their traffic is injected into the IUT from different directions. If the IUT operates in routing mode including NAT, P_{sip} and P_{dip} are rewritten with addresses that conform with the test environment. In this regard, if the test environment is configured according to Fig. 3, *IPS Evaluator* maintains the address mapping table shown in Fig. 11.

4.2. Partitioning traffic traces with constraints for IPS-testing

It is often required that packets containing attacks are fed into IUT so that false negatives are not generated. Failing to do so may cause the IUT to generate false negatives. To avoid such negatives, testers should be able to impose constraints on packet partitioning in order to warrant that specific packets are ultimately replayed. The *IPS Evaluator* does not honor subnet masking in traffic traces by default as demonstrated in Fig. 12, where node 67.117.243.207 falls into different group from nodes 67.117.243.201, 67.117.243.204, and 67.117.243.205 even though they share the same prefix 67.117.243. Obviously, the resulting traffic partitioning may not be desirable. For instance, testers may know that a number of subnet or host addresses belong to the internal systems and should be assigned in the same group (e.g. *red*). Testers may also refrain from replaying packets due to ARP requests/replies, some types of *ICMP* packets or *DNS* messages, simply because such packets are not only irrelevant to the ongoing testing but also may require services outside the test bed and interfere with the test process. Therefore, constraints often emanated from human expertise or manual analyses of traces can be integrated into our *IPS Evaluator*.

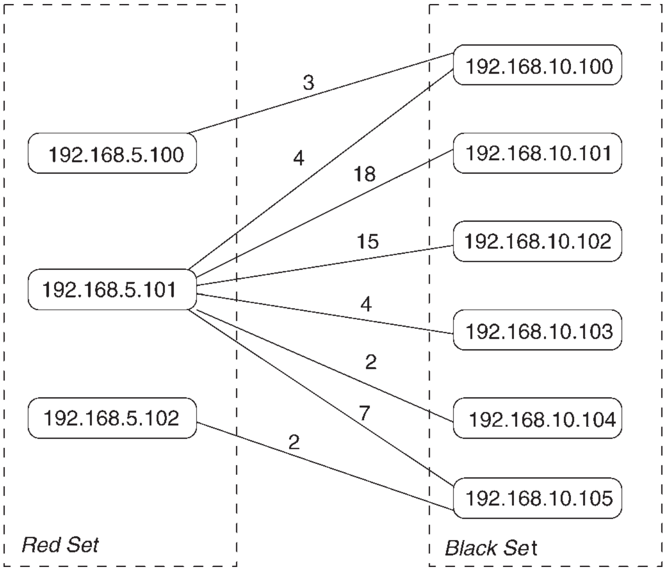

FIGURE 11. The address table generated by our testbed if we use the test environment configuration of Fig. 3.

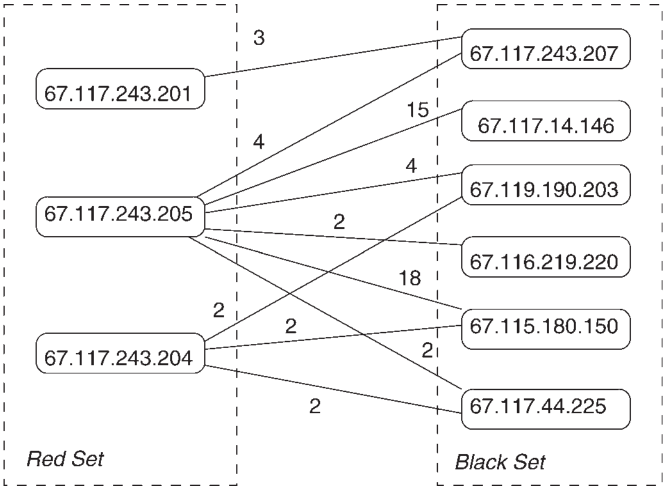

FIGURE 12. Bipartite graph for trace in Fig. 9 generated by Algorithm 5.

Algorithm 6 fulfills stated constraints as follows: if specific subnets, IP addresses or packets are not to be replayed, their corresponding vertices and/or edges are simply omitted when graph G is constructed. For any other constraints in forms of subnets, IP addresses or packets, we commence by constructing a new graph G' (V', E') in which V' and E' include all vertices and edges appearing in the constraints; clearly, $V' \subset V$ and $E' \subset E$. If G' is non-bipartite, then conflicts exist in the stated constraints which are thus impossible to satisfy simultaneously. Testers should refine their constraints to eliminate such inconsistencies. If G' is bipartite, then a two-coloring scheme c' for G' can be generated with the help of DFS. Vertices in G then inherit their color assignments from c' if such vertices also appear in G'. We sort all vertices of G according to their non-increasing degree order and store the results in a queue Q. When processing a vertex u in Q, we assign it an unused color if available; otherwise, we determine the color of u with the help of two weighted sums s_r and s_b formed by u's neighbors in a fashion similar to Algorithm 5. Algorithm 6 shows our heuristic method for partitioning a trace graph G with constraints.

While partitioning with Algorithm 5, nodes 67.117.243.204 and 67.117.243.205 are assigned the same color (i.e. *red*), and communications between the two nodes are not visible to IUT. By specifying the constraint that packets exchanged between 67.117.243.204 and 67.117.243.205 of Table 3 must be included in the resulting partition, we can obtain with the help of Algorithm 6 one such partition: the *red* group contains nodes 67.117.243.201, 67.117.243.205 and 67.117.44.225, while the *black* group consists of nodes 67.117.243.207, 67.116.219.220, 67.115.180.150, 67.119.190.203, 67.117.243.204, and 67.117.14.146; which happens to be one of the optimal partitions generated by Algorithm 4. Clearly, vertices 67.117.243.204 and 67.117.243.205 are assigned to different partitions, forcing their communications to be replayed by the simulation engine to IUT in different directions and therefore inspected by the IUT.

4.3. Manipulation operations for shaping traffic

In testing IPSs, it is imperative that we can shape the replayed traffic to possess desired properties such as background/foreground traffic mixture and attack intensity. To this effect, traffic might selectively include specific attack types and/or simulate the behavior of particular applications. Moreover, we should be able to test the IUT for its ability of carrying out protocol *normalization* or *scrubbing* [12, 13]. Traffic scrubbing is a required and important feature of IPSs as protocol inconsistencies and ambiguities resulting from different protocol implementations are often exploited by intruders [13]. Testing for normalization is feasible only if traffic contains overlapping IP or TCP fragments, out-of-order packets as well as packets with invalid sequence numbers and unexpected protocol headers. Although the `fragroute` tool can be used to carry out some of the above manipulations, its scope is limited as its operations are applied to individual packets only [13, 20].

To provide flexible and comprehensive traffic manipulation operations, we design multiple traffic operators by extending `fragroute`'s repertoire. Typically, these operations are specified in a script processed by the *Simulation Scheduler* component of our *IPS Evaluator* in Fig. 4 and are applied to traces before replay. Along with the baseline set of `fragroute`-like instructions that includes `ip_frag`, `tcp_seg`, `order`, `drop`, `dup`, `ip_chaff`, `ip_opt`, `ip_ttl`, `ip_tos`, `tcp_chaff` and `tcp_opt` to manipulate packets [20], we offer a range of additional operations some of which are presented in Table 4. We use the notation $(\cdot)+$ to indicate that items within the parenthesis may be repeated multiple times. This enhanced set of traffic shaping commands allows us to easily replace content of packets, segment/merge, duplicate, insert, delete, reshuffle, set specific

Algorithm 6 Partitioning vertices of a graph/trace $G(V,E)$ into two groups with constraints

```
construct $G'$ using vertices and edges specified in constraints; exit if $G'$ is not a bipartite;
$G'$ is colored with RED and BLACK: $C_{red}$ groups vertices with color RED while $C_{black}$ for vertices with color BLACK;
vertices of $G$ are sorted according to their non-increasing orders of degrees and the results are put into set $A$;
for(each vertex $u$ of $A$)
    visit[$u$] $\leftarrow$ UNVISITED;
    vertex partition[$u$] $\leftarrow$ RED if $u$ is in $C_{red}$; partition[$u$] $\leftarrow$ BLACK if $u$ is in $C_{black}$; otherwise, partition[$u$] $\leftarrow$ UNCOLORED;
end for
$u \leftarrow$ (first element of $A$); visit[$u$] $\leftarrow$ VISITING; partition[$u$] $\leftarrow$ RED; put $u$ into query $Q$;
while ($Q$ is not empty) do
    remove first element $u$ from $Q$; two weighted sums $s_r$ and $s_b$ are initialized to be zero;
    (each neighbor $v$ of vertex $u$) do
        $s_r$+=(weight of edge $(u, v)$) if (partition[$v$] = RED); $s_b$+=(weight of edge $(u, v)$) if (partition[$v$] = BLACK);
    if(visit[$v$] is UNVISITED)
        visit[$v$] $\leftarrow$ VISITING and push $v$ into queue $Q$;
    end if
  end for
  if (partition[$u$] = UNCOLORED) then
        partition[$u$] $\leftarrow$ RED if $(s_r < s_b)$ and partition[$u$] $\leftarrow$ BLACK otherwise; visit[$u$] $\leftarrow$ VISITED;
  end for
end while
$V$ of $G(V, E)$ are split into groups red and black containing vertices with color RED and BLACK respectively;
```

TABLE 4. Format and description of traffic manipulation commands applied by *IPS Evaluator* the traces.

Command	Format	Description
Payload manipulation		
`tcp_load`	*(port-num file-name)+*	Payload of packets from *port-num* are replaced with respective content from *file-name*
`tcp_replace`	*index filename [size]*	Content of packet *index* is replaced with that in *filename*
`tcp_scatter`	*index (size)+*	Packet *index* is segmented into several packets with size of *size*
`tcp_sign`	*index length string*	Packet *index* is changed to have size *length* and content *string*
Order manipulation		
`tcp_split`	*index (size)+*	TCP packet *index* is split into segments with sizes in *size*
`chop_insert`	*from to size [checksum]*	A new packet derived from packet *from* is generated with *size* and *checksum* and inserted after packet *to*
`dup_insert`	*from to*	A new packet, clone of *from* (identical payload and header), is created and inserted after packet *to*
Protocol field manipulation		
`ip_field`	*index (name value)+*	Value in *name* field of packet *index* is changed to *value*
`icmp_field`	*index (name value)+*	Value in *name* field of packet *index* is changed to *value*
`udp_ field`	*index (name value)+*	Value in *name* field of packet *index* is changed to *value*
`tcp_flag`	*(index flags)+*	Flag field in TCP header of packet *index* is changed to *flags*
`tcp_port`	*(from-port to-port)+*	from-port appeared in all packets is changed to *to-port*
`tcp_field`	*index (name value)+*	Value in *name* field of packet *index* is changed to *value*

order for packets, modify protocol fields in *TCP, UDP, ICMP* and *IP* headers and generate temporal properties of traffic including delay and retransmission.

We only outline the function of `tcp_load` and `tcp_scatter` for brevity as most of the operators in Table 4 are self-explanatory. With `tcp_load`, we replace the payload of all TCP packets originating from source port *port-num* with the content of a file designated by *file-name*. Multiple pairs of *port-num* and *file-name* can be used to change the payloads of multiple traffic streams. Clearly, `tcp_load` is a stream-based instead of packet-oriented operator. The command `tcp_scatter` partitions a TCP packet *index* into smaller segments whose size are specified with the *sizes* option. If multiple values are specified in *sizes*, the resulting segments assume the corresponding sizes in the provided order.

5. IPS TEST PROCEDURES

The inline and real-time operation of IPSs calls for new test procedures that can efficiently verify their effectiveness, attack coverage and overall performance. The multiple options which IPSs offer for handling attacks raise new issues for testing as it is no longer valid to examine the IUT's behavior exclusively based on its event log; the latter may differ from what actually occurs. Moreover, the continual appearance of new attack variants and vulnerabilities further exacerbate matters when it comes to IPS testing. With intrusion techniques becoming both versatile and divergent, it is increasingly challenging, if not unrealistic, for IPS testbeds to generate all possible exploits in an exhaustive and enumeration-based testing scheme. Therefore, we predominantly resort to group-based testing strategy. We focus on a number of widely-recognized attack families with each family represented by a set of test cases [43, 46, 50, 51]. It is generally accepted that such attack classifications offer an equally-effective alternative to enumeration-based testing for IPSs [14, 15]. In our group-based testing approach, attacks are first classified into groups systematically so that species within the same group possess similar characteristics. Representatives are then selected from each attack group and their corresponding traffic traces are used to evaluate IPSs. An IPS is considered to be able to identify all attacks in a group if it successfully detects the selected attacks; otherwise, the IPS is further tested by using every attack in the group. Clearly, the effectiveness of group-based testing heavily depends on the attack classification scheme employed. In Appendix 3, we discuss conditions under which group-based testing methods are more efficient in terms of the test cases used than traditional enumeration-based counterparts.

5.1. Classifying attack traffic and generating testing workloads

Classification of computer attacks is a multi-faceted task that entails considerations and evaluations on attack objectives, intrusion techniques involved, system vulnerabilities exploited and damages caused. The objective of an attack may range from reconnaissance to penetration and *DoS*. The intrusion genre includes Viruses, Worms, Trojans or Backdoors that exploit system vulnerabilities existing on operating systems, network protocols and applications. Organizations such as the `CVE` and `Bugtraq` uniquely name every known vulnerability or attack without attempting to offer a classification scheme [43, 46]. *Snort-Inline* [51] and *X-Force* [50] group attacks mainly based on exploited services; the former uses a flat classification structure while the latter forms a mesh in its categorization scheme. The above classification schemes may be inflexible in practice as they are inherently non-hierarchical and ambiguous. To overcome this limitation, we developed our own classification scheme by clustering attacks and their corresponding traces hierarchically with multiple features such as intrusion types, exploited services and severity levels of vulnerability. To facilitate the classification of attack traces, we first establish the associations between attack traces and the *CVE* database, then develop classifier to categorize automatically the *CVE* database with different taxonomic features to produce diverse classifications. For instance, Table 5 shows a service-based vulnerability classification resulting from our scheme. Here, web-related services are the most frequently exploited applications as they harbor about 61% of known loopholes; *SQL, Mail*, and *FTP* services constitute the next most favorite targets. Column *num* of Table 5 indicates the number of distinct attacks in the *CVE* database involved in each service class. The classification scheme of Table 5 is non-mutually exclusive as a single *CVE*-entry may describe multiple service vulnerabilities and exposures. Evidently, a single attack may simultaneously target multiple loopholes on more than one services, consequently, an attack trace may be assigned to multiple groups.

To provide additional flexibility in our group-based testing, the *IPS Evaluator* can also classify attacks according to their *malware type* (i.e. *intrusion type*) such as Virus, Worm, Trojan and Backdoor. Along the above lines, the *IPS Evaluator* can also organize attacks hierarchically based on malware type, service types as well as severity levels, offering a multi-level classification scheme. For instance, by classifying attacks first on malware type and then on services exploited, we have `Nimda` first fall into the *Worm* category and then within the *Web* service sub-group. Moreover, `Nimda` can be labeled as highly severe due to its potent nature. Clearly, by adjusting the granularity of the hierarchy for the attack classification scheme, we can readily create the required number of attack groups and therefore the number of test cases.

The workload traffic characteristics of the test cases generated significantly impact the performance of the IPS-under-testing [14]. These features include the ratio with which the *TCP/UDP/ICMP* protocols partake in the testing workload, the average packet size, the ratio of packet overhead to payload and the packet generation rates [52]. In addition, the mixture of foreground and background traffic, types of exploits, attack intensity and various evasion methods all play an important role in the evaluation of the IUT behavior. These features help produce diverse types of traffic that may force IPSs to analyze the protocol headers of all packets and/or inspect packet-payloads using *layer-7* analysis. Traffic with desired characteristics could be generated and captured in real-world network environments, stored in testbeds and then directly used in IPS test procedures. However, the often voluminous storage requirements for such traces impose significant constraints on testbed resources. For instance, to capture the traffic in a network with bandwidth of 100 Mbit/s for 1 h requires upto 45 GBytes. Such volumes can readily force the testbeds to reach out-of-resources state while in stress testing.

TABLE 5. The service-based classification of vulnerabilities.

No.	Service	Description	num.	pct	Examples
1	*WEB*	vulnerabilities in Web related services including *HTTP, HTML, CGI and PHP*	9,171	60.71	CVE-2000-0010
2	*SQL*	vulnerabilities in products based on SQL such as ORACLE and INFORMIX	1 736	11.49	CVE-2001-0326
3	*MAIL*	vulnerabilities on mail services such as SMTP, IMAP, POP and MIME	1 728	11.44	CVE-2001-0143
4	*FTP*	security loopholes in File Transfer Protocols	727	4.81	CVE-1999-0017
5	*CVS*	concurrent version systems such as CVS, SUBVERSION	250	1.65	CVE-2000-0338
6	*DNS*	flaws in Domain Name Services such as BIND	247	1.64	CVE-2004-0150
7	*SunRPC*	exploits targeting services based on SUN RPC, NIS, NFS	224	1.48	CVE-2001-0662
8	*SSL*	Secure Socket Layer	160	1.06	CVE-1999-0428
9	*SSH*	Secure Shell related attacks	152	1.01	CVE-2002-1024
10	*TELNET*	Telnet related exploits	147	0.97	CVE-1999-0073
11	*DECRPC*	exploits based on SMB, MS RPC, NETBIOS, SAMBA	140	0.93	CVE-2002-1104
12	*SNMP*	Simple Network Management Protocols	110	0.73	CVE-1999-0472
13	*LDAP*	Light-weight Directory Access Protocols	94	0.62	CVE-1999-0895
14	Total		15 107	100.00	

Therefore, in the test procedures of our *IPS Evaluator*, we predominantly use a subset of real-world attacks as templates and derive various traffic streams on-the-fly with designated features. The latter is accomplished with the help of traffic manipulation operators of Section 4.3.

5.2. Tests on prevention effectiveness and attack coverage

Similar to IDSs, it is important that an IPS provides a good coverage for exploits under a wide range of foreground/background traffic intensities. However, the most critical aspect in assessing an IPS on its prevention effectiveness is the consistency between what is log-recorded in the unit and what actually occurs during testing. The inconsistency between an IPS's event log and the actions it takes on the underlying traffic may reveal defects on system design, flaws in system implementation and system mis-configurations. For instance, many security devices including the open-source peer-to-peer detection/prevention *IPP2P* system and early *Snort-Inline* versions assume that TCP packets with flags *SYN, FIN* or *RESET* should not contain any payload and hence, they simply forward such packets without any security inspection. This is obviously a poor choice that a testing framework should expose. Moreover, in a number of open-source IPSs such as *Snort-Inline* and *Bro*, the functionalities of attack detection and delivery of countermeasure actions are performed by different subsystems or even external-to-IPS programs. Evidently, such choices may lead to inconsistencies between what is recorded in the IPS event-log and the actions taken on the ongoing traffic.

Upon detecting an attack, an IPS may forward, shape, block or carry protective actions on the traffic. While forwarding, an IPS lets an attack pass through but creates an alert record on its event log for subsequent forensic analysis. Through shaping, an IPS attempts to limit the bandwidth consumption by streams typically generated by applications such as instant messaging or peer-to-peer systems. When operating in blocking, an IPS drops malicious packets; discards all subsequent packets from the same session, refuses attempted connections from the same source host and/or subnet or even blocks all traffic for a period of time. In pro-active protection mode, an IPS actually tears down a bad connection by dispatching *TCP RESET* packets or *ICMP destination unreachable* messages to either or both ends of the session.

We build our procedure for testing IPS prevention effectiveness and attack coverage by first considering a group-based classification scheme (e.g. a sample scheme is depicted in Table 5). We form a representative set of attacks *A*—typically several dozens—based on their popularity, scope of distribution, complexity, severity and propagation mechanism. For instance, *A* may include `DeepThroat` and `Back Orifice` from the *Trojans* group, `Tribe Flood Network 2000 (TFN2K)` and `Stacheldraht` from the *DoS Attacks* class, as well as `Nimda`, `Slammer` and `Sasser` of the *Worm* group. To help automate the generation of the attack set *A*, our framework can be instructed to sample different attack types with specified mixture ratio. The attack sampling process is controlled by a seed so that repeatability is guaranteed. At first, we partition *A* into sets P_{attacker} and P_{victim} with the help of Algorithm 5. We could also use Algorithm 6 instead, should we have to accommodate additional tester-imposed constraints. The packets of P_{attacker} and P_{victim} are then injected into the IUT via its external and internal interfaces, respectively, and using the IUT's log as well as the recording mechanism of our *IPS Evaluator*, we seek to establish the IUT baseline behavior. By swapping the roles of attacker and victim so that sets P_{attacker} and P_{victim} are fed to the IUT through its corresponding internal and external interfaces, we can verify whether the IUT detects attacks originated from both internal and external networks.

We subsequently create a set of variant attacks *V* similar to those found in *A*. We accomplish this by employing different versions of attacks, attacks that use different exploits for the same vulnerability, attacks that target different operating systems and finally attacks in *A* that feature different yet valid protocols fields. We should point out that most variant attacks in *V* can be generated with the help of our shaping operations in Table 4. For instance, various versions of the `DeepThroat` Trojan uses slightly different banners, which are typically used as telltale patterns by IPSs to detect `DeepThroat` traffic. The banner used in `DeepThroat` version 1.0 is *–Ahhhhhhhhhh My Mouth Is Open SHEEP*, where *SHEEP* is the host name of the victim's machine. Based on the traffic of `DeepThroat` version 1.0, we can easily simulate `DeepThroat` communications for versions 2.0, 3.0 and 3.1 by changing the banner to *SHEEP -Ahhhhh My Mouth Is Open (v2)*, *SHEEP -Ahhhhh My Mouth Is Open (v3.0)*, and *SHEEP -Ahhhh My Mouth Is Open (v3.1)* with the help of traffic operator `udp_replace`, which is similar to `tcp_replace` of Table 4. Then, all attacks in *V* are fed into the IUT whose reaction is recorded with the help of the *IPS Evaluator* component *Behavior Arbitrator*. Based on the behavior that the IUT exhibits under both *A* and *V*, we can evaluate its attack coverage by computing the ratio of detected attacks by the IUT over the total attacks in both *A* and *V*. Suppose that *Snort-Inline* is fed with the traffic of `DeepThroat` version 1.0 in *A* and traffic simulating versions 2.0, 3.0 and 3.1 in *V*, it only raises alerts for `DeepThroat` versions 1.0 and 3.1 with the help of its signature database, but fails to recognize connections created by `DeepThroat` version 2.0 and 3.0, therefore achieving attack coverage of 50%.

To assess the prevention effectiveness of an IUT, we use background or benevolent application traffic *B* in conjunction with the above set *A* (or *V*). *B* consists of *TCP* and *UDP* traffic and is mixed with foreground traffic in *A* in a ratio α (in attacks per packet and by default, 80% is background traffic and 20% foreground traffic). Similar to attack-set *A*, background traffic set *B* can be created automatically with specified mixture of application types such as *HTTP, FTP* and *SMTP*. In this way, a number of distinct experiments are formulated in which the average packet size in *B* may ranges from 64 to 1518 bytes, the connection creation rate is between 1000 to 250 000 connections per second and the average life-time of connections is set between 10 to 60 000 ms. Initially, the IUT is configured to forward all detected attacks fed from *A* and during the experiments, the *IPS Evaluator* records all actions to help us ensure that the IUT is capable of identifying all malicious events. We repeat the above process for each IPS reaction option including *blocking*, *shaping* and *pro-active* protection and collect appropriate statistics. To finally determine the consistency of the IUT's external behavior, we correlate the IUT event-logs with what has been recorded by our testbed. We may repeat the above process while mixing attack traffic in *A* (or *V*) and background traffic in *B* with various ratios α and random orders to more accurately map the prevention effectiveness of the IUT in light of diverse traffic streams. Similar to attack coverage, the attack prevention consistency can be computed as the ratio of attacks successfully blocked by the IUT over the total reported attacks in IUT's event-log.

5.3. False positives and negatives in IPS-testing

The IUT attack detection accuracy is a key aspect that has to be evaluated and in this regard, we use the notions of attack detection and prevention rates. The former is defined as the ratio of the number of attacks detected over the number of attacks contained in IPS-injected traffic and the latter is the ratio of blocked attacks over the number of attacks launched. A false positive is an IPS-generated alert for attack-free traffic deemed malicious, while a false negative occurs when an IPS fails to detect/prevent a real attack and treats it as legitimate traffic. It is worth pointing out that false positives/negatives can be defined with respect to attacker, victim and security device [53]. In the victim-centric definition, false positives not only refer to events during which attacks were detected yet they did not actually took place, but also include attacks reported by the security device that did not have any effect on victim systems. In this context, the *Snort-Inline*-reported alert *WEB-FRONTPAGE/_vti_bin/access* is considered to be a false positive in an *Apache* web server environment as such an attack is only effective to *IIS* [54]. In the view point of the attacker, this action may be deemed successful if the intent were to fingerprint a web server. Clearly, the intention of the attackers are not measurable and/or testable by IPSs. Moreover, as the features of end-systems may vary greatly, their 'views' on false positives/negatives may also be very diverse. Consequently, we adopt more IPS-centric definitions in this paper: a false positive is an event raised incorrectly by an IPS with respect to the IPS's configuration. A false negative refers to an event that is expected to generate an alert according to the IPS's configuration but the IPS fails to do so [53].

There exist close relationships and therefore trade-offs among attack coverage, attack detection rate and false positives/negatives. For example, to achieve better attack coverage, an IPS may use a large signature base and relax checking conditions for some attack types; this may result in false positives since some normal traffic may be mistakenly identified as malicious. In addition, IPSs may generate false positives if they do not perform stateful inspection. For instance, a successful *TCP*-based attack should perform the normal *three-way-handshake* process before the real attack can proceed; otherwise, it is ineffective even if its malicious traffic reaches the target system, and IPSs without stateful inspections may still raise alarms for such ineffective attacks. However, to conduct stateful inspection, IPSs have to track every session. With a finite session table, IPSs may begin to drop new sessions or evict old connections under heavy traffic loads causing a self-inflicted denial of service. Therefore, the following aspects are critical as far as IPS-testing is concerned:

- Attack detection and prevention accuracy: we focus on whether the IUT blocks legitimate traffic.
- Stateful inspection capability: we aim at verifying whether the IUT maintains state information for sessions even under heavy traffic loads.
- Signature quality: we examine the quality of signatures used by the IUT. Fixed-port signatures can miss attacks that successfully target other ports. Real-world attacks typically target services instead of fixed ports and services can be provided on dynamic ports in addition to default ports [55]. For instance, about 2% of web servers provide services through non-standard ports (i.e. other than *TCP*-80). Thus, fixed-port signatures are expected to generate significant false negative rates [55].

To accomplish the above objectives in quantifying false positives and negatives as well as IUT detection and prevention rates, we use sets *A* and *B* of Section 5.2 and proceed in a four-phase procedure:

(i) By randomizing or reshuffling the order of attacks in *A* and then replaying the traffic, we can observe and record in our testbed the baseline behavior of the IUT. This step is repeated several times (20 times by default) with *A* being reshuffled before replay; the goal here is to ensure that detection/prevention accuracy and stateful inspection of the IUT are not affected by the order of attacks.

(ii) We then generate artificial/ineffective attacks as follows: for each *TCP*-based attack in *A*, we remove the *three-way-handshake* process making it a fake attack. Similarly, for each buffer-overflow attack, we modify its payload with operations `tcp_load`, `tcp_sign` and `tcp_replace` so that the payload is less than the size of the target buffer, making the attack invalid. We further change the fixed-port attacks to target alternative ports with operations `tcp_port` and `udp_field`. The resulting traffic is injected into the IUT and the step is repeated multiple times using reshuffling. Using ineffective attacks, we seek to quantify the IUT's false positive rate.

(iii) Subsequently, we create variant, yet effective attacks from *A* by using shaping operations including `dup_insert` to reorder packets, `chop_insert` to fragment and retransmit packets and `tcp_port` to change targeting ports. The resulting set is fed into the IUT multiple times using reshuffling and in this way, we compute the IUT's false negative rate.

(iv) In the final phase, we repeat the above three phases but we add background traffic from *B* in various intensities. Our objective here is to determine whether the IUT mis-classifies attacks or blocks legitimate traffic by mistake.

5.4. Testing IPS's resistance to evasion techniques

To avoid being detected, some attacks resort to evasion techniques including exploitation of *TCP/IP* protocol ambiguities, uniform resource locator URL obfuscation and service-oriented evasion mechanisms [13, 14]. *TCP/IP* protocol anomalies occur when an outgoing packet is split into multiple small fragments, some of which may present overlapping sequence number and different payloads. This is the case with traffic of Table 2 which is obtained by fragmenting the IP packets of Table 1 using the `ip_frag 75` command of Section 4.3. URL obfuscation techniques manipulate URL strings so that embedded malicious messages change their appearance to evade detection. For example, `Nimda` uses various character encoding schemes such as UTF-8 and hex codes to transform its malicious commands by rewriting a number of characters in URLs. As URLs can be encoded in a multitude of ways, an IPS should have the capability of recognizing all possible encoding schemes to defeat URL evasion attacks.

A number of URL obfuscation attacks also exploit ambiguities in Web protocols and inconsistencies in their implementations as manifested by tools such as `Whisker` and `SideStep` [56, 57]; Table 6 depicts a number of such obfuscation techniques; here, we denote the original URI with `org_URI` and the randomly generated string with *rand_str*. While some URL obfuscation techniques such as self-reference and *TAB* for delimiter are straightforward and require little effort by IPSs to identify, others including *fake parameter* and *HTTP* request pipelining demand complex decoding and/or deep inspection. Service-oriented evasion mechanisms exploit loopholes in applications protocols and/or their implementations such as *FTP, RPC, SNMP* and *DNS*. For example, by inserting data flows of telnet option negotiations into *FTP* control traffic, attackers may evade IPS detection if the latter does not perform *FTP* and *Telnet* protocol analysis [57]. Similarly, by using fragmented *SunRPC* records, attackers can split an *RPC*-based attack into multiple *RPC* fragments which can be still effective if reaching the victim but may not be detected by IPSs without *RPC* defragmentation functionality [57]. Furthermore, chunked encoding in *HTTP* services and rarely used message types in *DNS* are also exploited by evasion techniques [37].

TABLE 6. Some URL obfuscation techniques.

URL evasion technique	Content manipulation	Descriptions
Prepend long random string	Concatenation of *rand str* and *org URI*	Random string prepended to original URI
Random case sensitivity	Randomly choose characters in *org URI* and flip their cases	Change case for some bytes in URI
Directory self-reference	All character '/' in *org URI* is replaced by string "/./"	Change all / to"./"
Windows directory separator	All character '/' of *org URI* is replaced with double-backslash	Use backslash instead of slash in URI
Non-UTF8 URI encoding	Randomly choose characters in *org URI* and replace with hex	Change randomly select byte by its hex
Fake parameter	*/rand str1* . html%3F*rand str2=/../org URI*	Fake parameter 'html?*rand str2=*', which is removed by '/../'.
Premature URL ending	/%20*HTTP /1.1%0D%0AAccept%3A%20rand/../..org URI*	Fake URL end '%20HTTP /1.1%0D%0A', which is removed by'../..'
TAB as requested spacer	Replace all whitespace in *org URI* with tablet key	Change empty space to tab
Request pipelining	GET / HTTP/1.1|0D 0A|Host: fortinet.com . . .|0D 0A 0D 0A| GET /cgi bin%2Fph%66 HTTP/1.1|0D 0A 0D 0A|	More than one HTTP requests within a TCP packet, some chars are encoded in hex
POST instead of GET	POST / HTTP/1.1|0D 0A| . . .	Parameters in data section of the request

To investigate the IPS resistance to the aforementioned evasion attacks in the context of our testbed, we use both attack set A and background traffic set B constructed in Section 5.2, but we mainly focus on attacks targeting *HTTP, FTP, DNS, SMB* and *SunRPC*. Attacks in A are replayed to the IUT to establish the IPS baseline behavior. Then, for each attack in A, we create multiple variants by using the above evasive techniques. More specifically, the *IPS Evaluator* can generate most of the *TCP/IP* protocol anomalies through traffic manipulation operations such as `ip_frag`, `tcp_seg`, `ip_field`, `tcp_field`, `udp_field` and `icmp_field` of Table 4. URL obfuscation and service-oriented attacks can be created by commands `tcp_load`, `tcp_replace` and `tcp_sign` to rewrite the payload of *TCP* packets. All variants of attacks are injected into the IUT along with background traffic from B in various intensities. Based on the observed log-based IUT behavior in the above tests, our *IPS Evaluator* can evaluate the IUT's capability on evasion attack detection/prevention by computing the ratio of detected attacks over total evasive attacks generated. It is clear that without *TCP/IP* reassembly functionality, an IPS fails to identify any TCP/IP protocol anomaly; similarly, the IPS misses URL obfuscation and service-oriented evasion attacks without deep security inspection and application protocol dissection (i.e. layer-7 analysis).

5.5. Testing IPSs for performance

The deployment of IPSs either in switching or routing mode should not affect network performance noticeably. To this end, a number of IPSs opt for generating only a single alert for all identical attacks occurring in a row to save both CPU cycles and disk space. Similarly, for a session containing multiple attacks, IPSs may selectively record the first occurrence of exploits and skip the rest. The objective of our performance-specific testing procedure is to reveal the above peculiarity of IPSs, establish repeatability of experiments and quantify IPS throughput, latency and detection rates under typical, load-intensive and even out-of-resource settings. We resort to interleaved *UDP/TCP* and signature-based attacks so that IUTs are forced to scan every packet payload to detect all incidents requiring significant resource commitment. By increasing the number of concurrent attacks, we examine the behavior of IUTs as far as their session management is concerned. Every session contains a single type of malicious activity so that traffic workloads yield comparable results among different IPSs. In addition by increasing the life-span of injected connections, the *IPS Evaluator* forces the IUT to track more concurrent sessions and work under out-of-resource condition. Hence, the IUT may drop new sessions or evict old connections causing deterioration in performance.

In our IPS performance testing, we mainly manipulate two parameters: attack density and traffic intensity, the former defines the mixture of foreground and background traffic while the latter control the total traffic bandwidth. To facilitate the creation of attack with specified intensities, we first select all single-attack traces of A constructed in Section 5.2. We then further decompose A into a *UDP*-based attack portion A_{udp} and a *TCP*-based subset A_{tcp}. For instance, the A_{udp} may contain the `Slammer` attack with a single packet of 376 bytes, while A_{tcp} could contain the 16 `Nimda` packets of which only four have payload as Table 1 shows. Given the attack density α (in attacks per packet and $0 < \alpha < 1$), the *IPS Evaluator* computes the number of packets N in A and extracts $(1/\alpha - 1)N$ packets from the background traffic set B to form the traffic mixture, which has N/α packets in total. For each specified traffic intensity β (in packets per second), which is typically proportional to the IUT rated or nominal speed, the *IPS Evaluator* determines the replay speed and timestamps for each packet with the help of the *Simulation Scheduler* component. Clearly, the replay procedure lasts $N/(\alpha\beta)$ seconds and by adjusting the number of attacks selected from A and therefore the number of packets N in selected attack traces, we can control the feeding period.

By configuring the *IPS Evaluator* to take *blocking* action on all identified attacks, our testbed with settings shown in Fig. 5 randomly interleaves packets from attack sets A_{udp} and A_{tcp} as well as background traffic set B before feeding them into the IUT according to Algorithm 1 with the specified rate β packets per second. In the above tests, we monitor the IUT throughput and measure network latency in addition to detection/prevention rates. IUT throughput is the ratio of total traffic in terms of packets encountered over the duration of observation, and latency is the average time gap between a packet leaving its source and reaching its destination for all packets in the replayed traffic. By specifying different traffic intensities in the range of the IUT-rated speed and even larger than the IUT nominal rate if needed, we can obtain the maximum IUT throughput; this is the highest replayed traffic intensity that does not cause either blocking of legitimate traffic or forwarding of attack streams.

To test IPSs in the presence of massive concurrent connections, under heavy workloads and/or out-of-resources settings, we configure the IUT to take *blocking* action so we can readily identify instances of no-detection. Then with the help of our shaping operations `tcp_scatter`, `ip_frag` or `tcp_seg`, we split every attack trace in A_{tcp} into two or more IP-fragments. In this manner, we create two new traffic sets: A'_{tcp} and A''_{tcp}, the former consists of the prime IP-fragments of all attacks in A_{tcp} and the latter contains all the remaining fragments. Subsequently, we replay A'_{tcp} followed by near the IPS-stated-maximum number of concurrent connections from background set B for a specified period of time typically ranging from 0.30 to 60 s. Finally, the second part of attacks A''_{tcp} is replayed. The rationale of the above test procedure is to force the IUT to operate in the out-of-resource state so that the IUT exhausts its session table and may start dropping sessions. Our *IPS Evaluator* evaluates the IUT performance based on its action against attacks in A'_{tcp} and traffic in B. Obviously, the performance of IUT may actually degrade if the IPS forwards attacks or blocks legitimate traffic due to session management problems and/or out-of-resource situations.

6. EXPERIMENTAL EVALUATION OF IPSs USING THE TESTBED

We implemented the *IPS Evaluator* in *C* and *Perl* and used our testing methodology to examine a number of IPSs in order to investigate their features and performance aspects. For brevity, we outline key aspects of our experimentation with *Snort-Inline* and *FortiGate 2.80* [51, 58]. *Snort-Inline* is a lightweight IPS based on the `IPtables/Netfilter`, a packet-filtering utility to intercept and manipulate packets and `libnet`, a library that helps send out *TCP RESET* and *ICMP destination unreachable* messages. By performing pattern matching and analyzing traffic flow characteristics, *Snort-Inline* can detect and prevent various incidents such as buffer overflows, portscans, and protocol anomalies. *Snort-Inline* may take configurable actions on the malicious packets including alerting, dropping, or tearing down connections; limited stateful inspection capability and service-specific inspections are also provided. Figure 13 shows the various components of a test machine on which *Snort-Inline* is deployed as an IPS. *Snort-Inline* functions atop the `IPtables/Netfilter` and `libnet` modules and generates two types of verdicts: *NF_DROP* for malicious traffic or *NF_ACCEPT* for normal data streams based on rules specified in the configuration file *snort.conf*. Through the command ‘iptables -A INPUT -p ALL -j QUEUE’, one may have the `IPtables/Netfilter` module receive all the streams from all network interfaces. The command *‘snort-inline -c snort.conf -Q’* helps configure *Snort-Inline* to fetch packets from the `IPtables/Netfilter` module while generates *NF_DROP/NF_ACCEPT* verdicts.

In our experimentation we used *Snort-Inline v.2.3.2* along with its 4637 rules that are enabled by default. To pro-actively terminate attack connections, *Snort-Inline* may generate extra messages such as *TCP RESET* or *ICMP destination unreachable* transmitted via `libnet`. We should point out that non-routable packets are dropped by the `Router/Bridge` module of Fig. 13. As a result, they are not delivered to *Snort-Inline* and consequently they are not subject to security inspection. Hence, we use Algorithms 5 and 6 in our testbed to partition traffic traces so that replayed packets are IPS-routable.

FortiGate is an IPS/anti-virus product that detects and prevents attacks using multiple techniques including pattern matching, anomaly analysis, traffic correlation and layer-7 protocol dissection. Machines in our testbed are equipped with Intel 1.80 GHz, 512 MBytes of main memory, and 80 GBytes disk storage running either *Red Hat Linux Kernel 2.4.7* or *Windows 2000*. All machines maintain two network cards and are connected via 100/1000 Mbit/s switches. Here, we also use the out-of-box *FortiGate* configuration with respective signatures enabled. Results reported here pertain the default behavior of the IUTs. In case that a false positive/negative is generated by an IUT, we analyze whether it can be corrected by manipulating the IUT’s configurable parameters without any update on attack signatures or executables and present corresponding correcting measures if available. The traces used in our experiments are mainly captured or synthesized by the *Threat Analysis Center (TAC)* of *Fortinet* [59]. The set of traffic traces used covers most vulnerabilities presented in Table 5.

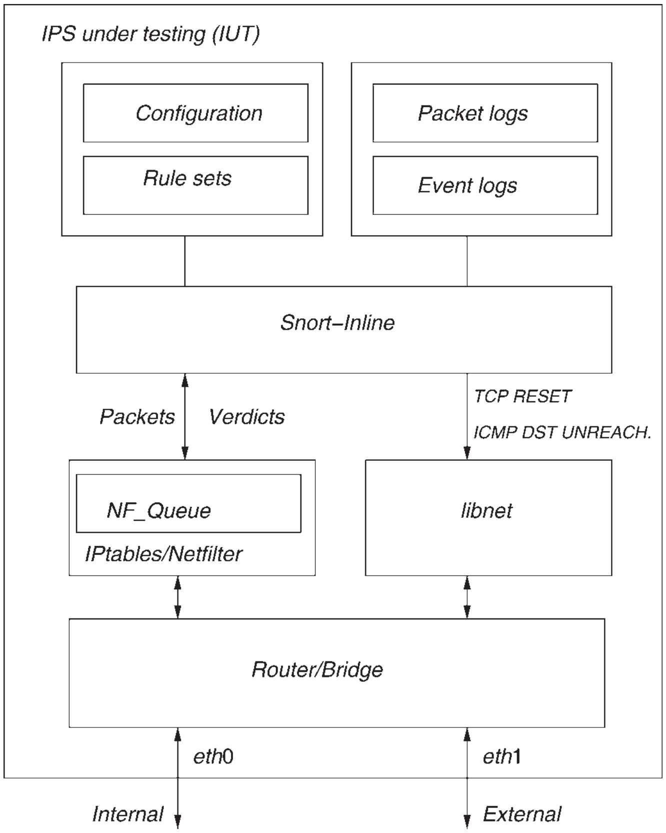

FIGURE 13. Components of *Snort-Inline* IPS system.

6.1. Attack coverage and prevention effectiveness

Initially in this set of experiments, we form a set of attacks A that includes `Nimda` and `Slammer` according to the guidelines of Section 5.2. We serially inject this set into the IUTs to determine the baseline behavior of *Snort-Inline* and *FortiGate* for their attack coverage and prevention effectiveness. We group packets of A into two sets P_{attacker} and P_{victim} and feed them into IUT’s external and internal interfaces in both possible ways. For most traces, the automatic traffic partitioning schemes generated by Algorithm 5 are valid and therefore can be used directly; only a few test cases with mesh network topologies or special IP addresses require tester intervention to specify constraints for Algorithm 6. For instance, this is the case when the *DoS* attack tool `Stacheldraht` is used. It generates messages with source *IP* address 3.3.3.3—a pattern often used by IPSs to detect attacks. Here, we craft constraints to avoid rewriting the *IP* address 3.3.3.3. Our testing shows that both *Snort-Inline* and *FortiGate* successfully detect and prevent all attacks in *A* initiated either internally or externally.

IUTs may generate multiple alerts for a single attack due to overlapping coverage of non-orthogonal signatures. For instance, *Snort-Inline* identifies `Nimda` by searching for pattern 'cmd.exe' in traffic — this telltale appears in the payload of packet 4 of Table 1. The signature for detecting `Nimda` in *Snort-Inline* is defined as *alert tcp $EXTERNAL_NET any → $HTTP_SERVERS $HTTP_PORTS (msg:'WEB-IIS cmd.exe access'; flow:to_server,established; uricontent:'cmd.exe' nocase; classtype:web-application-attack; sid:1002; rev:8;).* In the out-of-box configuration of *Snort-Inline*, both $*EXTERNAL_NET* and $*HTTP_SERVERS* are set to 'any' indicating that every incoming packet should be matched against the signature. In addition to pattern matching, *Snort-Inline* also subjects `Nimda` traffic to *HTTP* protocol dissection. Thus, *Snort-Inline* generates two alerts: the first alert *WEB-IIS cmd.exe access* is due to the afore-mentioned signature *sid-1002* and the second alert is due to the detection of *Double Decoding Attack* in payload '..%255c..' by *Snort-Inline*'s *HTTP* inspector. In comparison, *FortiGate* assigns a severity level (e.g., high, medium, low and informational) to each signature and can be configured to report the alarm with the highest severity when multiple rules are satisfied by a session. In what follows, we consider that an IPS successfully detects an attack as long as one of the invoked alarms is relevant.

In the next phase of this set of experiments, we generate variants of each attack in *A* with the help of the shaping operators of Table 4. For instance in the context of `Nimda`, we generate variants by changing the payload of packet 4 with command `tcp_replace`. Table 7 shows the payloads for some such rewritten packets using various URL encoding mechanisms including hex-encoding (payload 2), unicode scheme (payload 5), and invalid string (payload 6). We also create variants with the help of the evasion techniques of Table 6. For instance, payload 7 employs hex-encoding to transform the telltale *cmd.exe* to *cmd%2Eexe*, effectively hiding the malicious content. *Snort-Inline* raises the alarm *Double Decoding Attack*, but occasionally does not produce the alert *WEB-IIS cmd.exe access* indicating that some URLs with evasive techniques are not decoded appropriately. In contrast, the out-of-box configuration of *FortiGate* generates *WEB-IIS cmd.exe access* alerts for all `Nimda` variants indicating the correct behavior of its *HTTP* protocol analyzer.

In the last phase of this set of experiments, we investigate the coverage of IUTs for non-content-based incidents such as portscans. We use our testbed to simulate activities of network scanners such as `Nmap` and `Hping` [41, 60]; both are considered *attack tools* in our categorization scheme and are typically launched by attackers to fingerprint OSs and services of victims. *Snort-Inline* features a dedicated module `portscan` to detect both horizontal and vertical portscans [51, 61]. *Snort-Inline* identifies a vertical portscan mainly by checking the condition that a few hosts contact a small set of destination hosts but the numbers of unique ports and invalid responses (e.g. *TCP RESET*s) from destination hosts

TABLE 7. Payloads of variant `Nimda` attacks.

No	Payload	Description	Snort-Inline	FortiGate
1	GET /scripts/..%252f../winnt/system32/cmd.exe?/c+dir HTTP/1.1	Original payload	As expected	As expected
2	GET /scripts/..%%35%63../winnt/system32/cmd.exe?/c+dir HTTP/1.1	'%252f' to '%%35%63'	As expected	As expected
3	GET /scripts/..%%35%63../..%%35%63../..%%35%63.. /winnt/system32/ cmd.exe?/c+dir HTTP/1.1	'%252f' to '%%35%63'repeat 3 times	As expected	As expected
4	GET /scripts/..%255C../winnt/system32/cmd.exe?/c+dir HTTP/1.1	'%252f' to '%255C'	As expected	As expected
5	GET /scripts/..%C0%AF../winnt/system32/cmd.exe?/c+dir HTTP/1.1	'%252f' to '%C0%AF'	As expected	As expected
6	GET /scripts/..%C0%9V../winnt/system32/cmd.exe?/c+dir HTTP/1.1	'%252f' to '%C0%9V'	As expected	As expected
7	GET /scripts/..%252f../winnt/system32/cmd%2Eexe?/c+dir HTTP/1.1	cmd.exe: cmd%2Eexe	As expected	As expected
8	GET /scripts/..%252f../winnt/system32/cmd%252eexe?/c+dir	cmd.exe: cmd%252eexe	As expected	As expected
9	GET /scripts/..%252f../winnt/system32/cmd%32%65exe?/c+dir	cmd.exe: cmd%32%65exe	As expected	As expected
10	GET /scripts/..%252f../winnt/system32/cmd%U002Eexe?/c+dir	cmd.exe: cmd%U002Eexe	As expected	As expected
11	GET /%20HTTP /1.1%0D%0AAccept%3A%20*rand*/../../scripts/..%252f../winnt/ system32/cmd%2Eexe?/c+dir HTTP/1.1	premature URL and md.exe to cmd%2Eexe	As expected	As expected

TABLE 8. Horizontal and vertical portscans simulated in our testbed.

#	Timestamp	src IP	src por	dst IP	dst port	pkt len	TCP hdr/pld	TCP flag	Description
Horizontal portscan									
1	0.000000	10.80.8.183	32872	10.80.8.221	80	74	40/0	SYN	Request
2	0.000100	10.80.8.221	80	10.80.8.183	32872	74	40/0	SYN|ACK|RST	Reply RESET pkt
3	0.000200	10.80.8.183	32872	10.80.8.222	80	74	40/0	SYN	Request
4	0.000300	10.80.8.222	80	10.80.8.183	32872	74	40/0	SYN|ACK|RST	Reply RESET pkt
5	0.000400	10.80.8.183	32872	10.80.8.223	80	74	40/0	SYN	Request
6	0.000500	10.80.8.223	80	10.80.8.183	32872	74	40/0	SYN|ACK|RST	Reply RESET pkt
7	0.000600	10.80.8.183	32872	10.80.8.224	80	74	40/0	SYN	Request
8	0.000700	10.80.8.224	80	10.80.8.183	32872	74	40/0	SYN|ACK|RST	Reply RESET pkt
9	0.000800	10.80.8.183	32872	10.80.8.225	80	74	40/0	SYN	Request
10	0.000900	10.80.8.225	80	10.80.8.183	32872	74	40/0	SYN|ACK|RST	Reply RESET pkt
Vertical portscan									
1	0.000000	10.80.8.183	32872	10.80.8.221	80	74	40/0	SYN	Request
2	0.000100	10.80.8.221	80	10.80.8.183	32872	74	40/0	SYN|ACK|RST	Reply RESET pkt
3	0.000200	10.80.8.183	32872	10.80.8.221	81	74	40/0	SYN	Request
4	0.000300	10.80.8.221	81	10.80.8.183	32872	74	40/0	SYN|ACK|RST	Reply RESET pkt
5	0.000400	10.80.8.183	32872	10.80.8.221	82	74	40/0	SYN	Request
6	0.000500	10.80.8.221	82	10.80.8.183	32872	74	40/0	SYN|ACK|RST	Reply RESET pkt
7	0.000600	10.80.8.183	32872	10.80.8.221	83	74	40/0	SYN	Request
8	0.000700	10.80.8.221	83	10.80.8.183	32872	74	40/0	SYN|ACK|RST	Reply RESET pkt
9	0.000800	10.80.8.183	32872	10.80.8.221	84	74	40/0	SYN	Request
10	0.000900	10.80.8.221	84	10.80.8.183	32872	74	40/0	SYN|ACK|RST	Reply RESET pkt

are significant. Similarly, a horizontal portscan is also reported if an attacker attempts to connect simultaneously to many hosts on a small number of unique destination ports but the number of invalid responses from destinations is relatively high (i.e. more than 5).

Our testbed can simulate various types of portscans. For example to generate a horizontal portscan, we can extract the first two packets from the `Nimda` attack of Table 1 and manipulate them as follows: the first packet containing a *TCP SYN* message is duplicated *n* times and the destination IPs of the resulting packets are randomized[1]. The second extracted packet containing a *TCP SYN|ACK* message is also cloned *m* times with a *TCP RST* bit added to each replicated packet. The first half of Table 8 presents the resulting traffic stream with $n = m =$ 5. Vertical portscan can be simulated similarly by manipulating ports instead of IPs and an sample is depicted at the second half of Table 8 with $n = m = 5$. We should point out that *n* and *m* are not necessarily the same and by varying these two parameters, we can test the sensitivity of IPSs to portscans, which is defined as the lowest traffic intensity triggering IPS alerts. We vary *n* and *m* in range [2, 15] to simulate horizontal portscans and feed the resulting traffic into *Snort-Inline*, which generates the alert '*(portscan) TCP Portsweep*', if the replayed traffic is considered to be a horizontal portscan. Figure 14 depicts the outcomes of all the experiments that results from the different values of *n* and *m*. We differentiate between test cases yielding *Snort-Inline* alerts from those that do not. Evidently, *Snort-Inline* raises alarms for horizontal portscans only when the number of *TCP RESET* packets from scanned hosts is larger than 5. Port-scan activities triggering low responses slip *Snort-Inline*'s detection. In this regard, the first half of

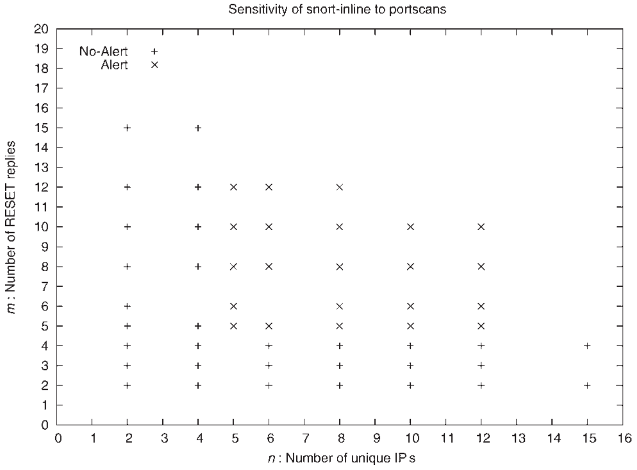

FIGURE 14. *Snort-Inline* sensitivity to portscans.

[1] Most IPSs including *Snort-Inline* detect portscans based on statistical characteristics of traffic, therefore, randomization and sequentialization have the same effect.

the trace in Table 8 is the horizontal portscan with the lightest traffic intensity detected by *Snort-Inline*. Similarly, the second half of Table 8 provides the minimum set of packets that causes *Snort-Inline* to identify as vertical portscan and raise the '*(portscan) TCP Portscan*' alert. We should point out that *Snort-Inline* computes statistical characteristics of portscan traffic with a sliding time-window, therefore, the replay speed of traffic traces also determines whether a portscan alert is raised.

6.2. Testing for false positives and negatives

Exposing possible weaknesses of IUTs regarding their false negatives/positives is of vital importance to the overall IPS testing procedure. To this end, we follow the four-phase procedure described in Section 5.2 by constructing an attack set A and background traffic set B. The set A typically contains 80 attack traces and consists of TCP-based A_{tcp} and UDP-oriented A_{udp} attacks; the ratio between A_{tcp} and A_{udp} is configurable with a default value of 80:20. To better facilitate and automate the testing procedure, we define about 40 template scripts with the help of the Table 4 traffic operators and apply these scripts to traces in A to generate upto 3000 attack variants. Table 9 shows a portion of such scripts applied to the `Nimda` trace of Table 1 and the respective outcomes for both *Snort-Inline* and *FortiGate*.

In particular, script *no-handshake* creates an ineffective `Nimda` attack without the normal TCP three-way-handshake procedure by removing the first three packets in the trace of Table 1. Both *Snort-Inline* and *FortiGate* raise no alarm for the traffic as the connection status *ESTABLISHED* is one of the conditions triggering an alert for the `Nimda` attack. Although the '*Double Decoding Attack*' alert is still generated by *Snort-Inline*, we consider it acceptable as such an alert is typically used as auxiliary information only by system administrators. Similarly, script *normal-retrans* simulates a normal retransmission by duplicating `Nimda`'s packet which forces *Snort-Inline* to produce a false alarm as it regards the traffic to be evasive retransmission by its *TCP* protocol dissector. Although such an alarm can be turned off, *Snort-Inline*'s ability to detect evasion attacks is also disabled as a result. In contrast, *FortiGate* recognizes the retransmission. The script *diff-checksums* yields two *TCP* packets with the same sequence number and packet size but different payloads and checksums; the first packet is attack-free while the second contains malicious content. In using this script, we sought to establish whether the IUT considers the second packet as a simple retransmission. Both systems successfully detect the attack and mark it as an evasive-retransmission. The test case *same-checksum* features two packets with the same sequence number, payload size and checksums with the first packet being attack-free and the second malicious. *Snort-Inline* generates a false negative as it only compares checksums to determine the identity of the original packet and its retransmitted clone. The logic to determine the identity of packets is hard-coded in the *TCP* protocol dissector of *Snort-Inline* and thus, it is not configurable by testers.

Snort-Inline behaves as expected in scripts *acked-retrans* and *forward-overlap*; the former represents the retransmission of an acknowledged packet, while in the latter, packet 4 is first duplicated and then the original packet is split into two segments of sizes 20 and 44 bytes with the copy being rewritten with random content. Although *Snort-Inline* correctly identifies overlapping packets and proceeds to normalize traffic using the *favor new* policy, it does forward the overlapping packets intact, providing an opportunity for evasion attacks. In both *acked-retrans* and *forward-overlap* cases, the *IPS Evaluator* helps establish that *FortiGate* produces no false positives/negatives. In the *acked-part-retrans* case, we initially split packet 4 into two segments of sizes 20 and 44, then replicate the first segment and place it after packet 6. Hence, a portion of the original packet 4 is retransmitted once the

TABLE 9. Scripts that help expose False positives/negatives for *Snort-Inline* and FortiGate.

No	Name	Command sequence	Description	Snort-Inline	FortiGate
0	*No-handshake*	drop 0, 1, 2	conn. without 3way handshake	As expected	As expected
1	*Normal-retrans*	dup insert 4 4	normal pkt retransmission	False positive	As expected
2	*Diff-checksums*	chop insert 4 3 0	pkt retrans with different checksums	As expected	As expected
3	*Same-checksum*	chop insert 4 4 0 0	pkt retrans. with same checksum	False positive	As expected
4	*Acked-retrans*	dup insert 4 5	retransmit acknowledged packet	As expected	As expected
5	*Acked-part-retrans*	tcp split 4 20 44; dup insert 4 6	split pkt 4 and insert first behind pkt 6	False positive	As expected
6	*Forward-overlap*	dup insert 4 4; tcp split 4 20 44;	pkt 4 is duplicated then split tcp replace 6 file 64 into two, pkt 6 is replaced	As expected	As expected
7	*Sport-alter*	tcp port 32872 32771	change port 32872 to 32771 (RPC)	False positive	As expected
8	*Dport-alter*	tcp port 80 8080	port 80 changes to 8080	As expected	As expected
9	*Ip-fragment*	ip frag 75	IP payload splits into 75-byte pieces	As expected	As expected
10	*Method-type*	tcp replace *GET*, *PUT*	HTTP method from GET to PUT	False positive	As expected
11	*Alt-exploit*	tcp replace *scripts msadc*	exploit other vulnerability	As expected	As expected

entire packet has been acknowledged, forcing *Snort-Inline* to produce an alert of *window violation* which is a false positive; instead, *FortiGate* treats the traffic as normal and produces no alert.

When the IUT mis-classifies incoming traffic and analyzes it with incorrect protocols, false positives can be produced. The script *sport-alter* is such an example in which the client of the *HTTP* connection happens to use 32771 as its source port. This port is registered by the *RPC* services. *Snort-Inline* treats this test as an *RPC incomplete record* attack, obviously a false positive. In script *dport-alter*, we change the Web-server port from 80 to other popular *HTTP* ports such as 8080 and both *Snort-Inline* and *FortiGate* fail to identify `Nimda` under their default configurations. This may not be considered to be a false negative in the viewpoint of IPSs as *Snort-Inline* and *FortiGate* only detect Web-specific attacks against servers listening on *TCP* port 80 by default. However, IPSs may be expected to raise alarm for script *dport-alter* if *IIS* web servers indeed provide services on port 8080. By adding *TCP* port 8080 as one of Web service ports recognized by their *HTTP* protocol analyzers, both IUTs can detect the attack. The evasive attack generated by the script *ip-fragment* is successfully detected by both *Snort-Inline* and *FortiGate* even though the telltale pattern is spread over multiple IP fragments. Similarly, script *alt-exploit* exploits the fact that directory traversal vulnerability is independent of the root directories, and changes the directory name from *scripts* to *msadc*; the resulting attack variant, which is still effective, is recognized by both *Snort-Inline* and *FortiGate*. The two IPSs while operating in *IPS Evaluator* behave differently in the traffic created by script *method-type*, which creates an ineffective attack by changing *HTTP* method from *GET* to *PUT*. *Snort-Inline* raises an alert, indicating that *HTTP* methods such as *GET* and *PUT* are not a triggering condition for its `Nimda` signature.

All the above results along with many other experiments we carried out point to the fact that *Snort-Inline* generates multiple false positives/negatives indicating loopholes in its attack detection mechanisms, protocol analysis and signature crafting; by comparison, *FortiGate* delivers improved attack detection accuracy.

6.3. Testing IPSs for performance

The *IPS Evaluator* helps us carry out stateful inspection and examine resistance to evasion attacks of IPSs under intense concurrent foreground and background traffic. By using the approach outlined in Section 5.5, we generate an attack set A as well as a background set B using predominantly two parameters: attack density α and traffic intensity β. While attempting to best ascertain the IUT's counter-evasion capability, we manipulate the traffic traces in A with the help of script *ip-fragment* defined in Table 9. For instance, by applying the command `ip_frag` 75 of script *ip-fragment* to the `Nimda` traffic in A, we obtain an evasive variant of the `Nimda` attack shown in Table 2; here, the telltale *cmd.exe* is fragmented between packets 4 and 5 containing strings *cmd.e* and *xe*, respectively. In general, we apply IP fragmentation for all packets in A with the rationale that signatures exploited by IUTs for attack detection are split in multiple IP fragments. Consequently, IPSs should feature IP-defragmentation to actually identify evasive attacks. With each attack density α in the range of [0, 0.80] and traffic intensity β proportional to the IUT's pro-rated speed (e.g. [10%, 75%]), the *IPS Evaluator* computes the number of packets N in A, randomly selects $(1/\alpha - 1)N$ packets from background set B, and replays the resulting traffic mixture to both *Snort-Inline* and *FortiGate* with the specified traffic intensity (i.e. β packets per second). The two IPSs successfully detect such attacks in various combinations of α and β, and in this way we verify their IP de-fragmentation functionality.

To test the IUT's management of session information for long-lasting streams, we proceed as follows: we split attacks into two parts A_{tcp}' and A_{tcp}'' as we outline in Section 5.5; for `Nimda` in particular, packets 1–4 become part of A_{tcp}' and packets 5–15 as well as TCP termination procedure (not shown in the table) are grouped in A_{tcp}''. After feeding the IUT with A_{tcp}', we generate 10 000 concurrent background sessions for *Snort-Inline* and 250 000 for *FortiGate* each lasting upto 60 s by using as many as 10 test machines; finally, we inject the A_{tcp}'' part of the traffic. Both *Snort-Inline* and *FortiGate* can identify the involved attacks demonstrating reliable session tracking capabilities; for *Snort-Inline* however, this is only attained for much fewer concurrent connections—10 000—and only if the background sessions last upto 30 s. When the background traffic features longer connections, *Snort-Inline* starts dropping sessions with the longest inactive time; this *session pruning* yields false negatives.

Following the test procedure of Section 5.5 and generating IPS traffic with intensity ranging between 10 and 600 Mbit/s with up to 10 test machines, we establish that the maximum throughput achieved by *Snort-Inline* before any false positives/negatives appear is only 17 Mbits; for *FortiGate*, this rate is at approximately 600 Mbit/s with the equipment vendor-rated at 400 Mbit/s. Actually, the same conclusion has been independently reached by the *NSS-Lab* [14]. Under the maximum throughput, the latency achieved by *Snort-Inline* is 300 μs and by *FortiGate* is 200 μs, while the average response time for background *HTTP* sessions is 220 ms for *Snort-Inline* and 200 ms for *FortiGate*. It is also worth pointing out that due to inexpensive IPC used in our framework, each test machine can readily generate upto 90 Mbit/s traffic out when the network interface cards are at 100 Mbit/s. If the IUTs are equipped with 1000 Mbit/s NICs, test machines can comfortably flood a network segment with 600 Mbit/s traffic. Consequently in order to test IUTs with rated networks of 1–4 Gbit/s bandwidth, our framework requires only a handful of machines to form the necessary testbed (of Fig. 5). Our experiments also indicate that the bottleneck when high-volume traffic is involved—easily generated by repeating a small trace such as that of Table 1—appears to be the network driver within the OS. This occurs due to excessive memory-to-memory copying between kernel and user space taking place during stress-tests that involve voluminous traffic. Such a bottleneck in stress-test, which is also observed by other testbeds such as *tcpreplay* [19], could be mitigated by allocating much more memory to the network driver.

To further investigate the relationship between traffic intensity and the capability of an IPS on detecting attacks, we use the traffic trace labeled as '1999 train set, week one, Wednesday' from *MIT's Lincoln Laboratory* [62]. By configuring *Snort-Inline* to work in bridge mode so that it forwards all traffic and by feeding it with the 351.5 MB *MIT* trace with various replay speeds in the range of [1, 25] Mbit/s, we record the number of alerts generated by *Snort-Inline* and its processing (wall) time. Figures 15 and 16 show the respective results. When the trace is replayed with its original speed, *Snort-Inline* generates 73 989 alerts. The change in replay speed may distort the temporal characteristics of the original traffic and therefore may affect the number of alerts generated by IPSs. However, the noticeable drop on the number of alerts after 17 Mbit/s is attributed to the fact that *Snort-Inline* cannot effectively deal with the intensive traffic streams. As the replay speed increases, the observed wall time gets diminished as Figure 16 depicts, indicating the accuracy with which our testbed controls the trace-feeding rate.

Overall, *FortiGate* demonstrates good attack detection accuracy and offers a broader attack coverage if compared to *Snort-Inline*; also, *FortiGate* allows upto a half million concurrent connections, provides lower network latency and handles better long-lived sessions. Through our testing, we also established that *Snort-Inline* occasionally generated multiple different alerts for a single attack exposing a problem with overlapping coverage by different signatures. In addition, service-oriented evasion attacks targeting *SSH* and *SSL* were missed revealing problems in the deep inspection capabilities of *Snort-Inline*. We have also used the *IPS Evaluator* to benchmark a handful of available IPSs including products from *Juniper*, *SonicWall*, and *TippingPoint*. There are a number of issues shared by most IPSs that we have used in our evaluation which briefly are:

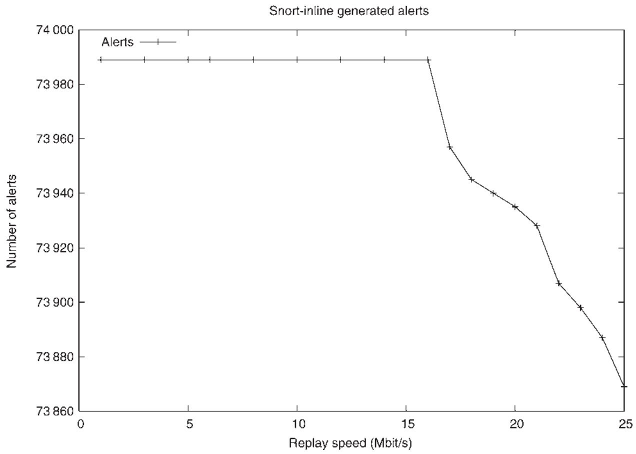

FIGURE 15. Alerts generated by *Snort-Inline* for the Lincoln trace '*1999 Train Week1 Wednesday*'

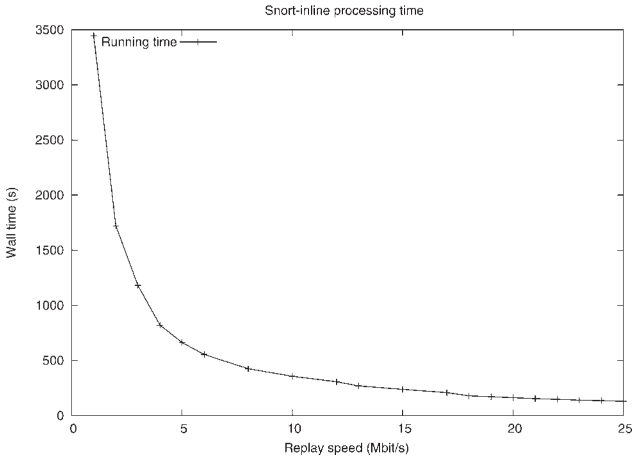

FIGURE 16. *Snort-Inline* processing time for the trace '*1999 Train Week1 Wednesday*'.

- Multiple alerts may be raised for a single attack due to the complexity of vulnerabilities and/or exploits as well as the overlapping coverage of different signatures used by IPSs.
- Trade-offs exist among false positives/negatives, attack coverage and performance. A better attack coverage requires a larger signature base which often adversely affects the IPS performance.
- Incomplete coverage on possible attack vectors does influence the prevention capabilities of IPSs. As it is desired to apply preventive actions on specific groups of attacks, we found that it is extremely difficult for many IPSs to entirely prevent all types of say instant messaging and/or peer-to-peer communications from occurring.
- Inconsistencies between event-logs and actions taken by the IUT on underlying traffic are predominantly due to defects in IPS design, implementation and configuration. Delegation of preventive actions to different subsystems or even physical devices is often the source for out-of-synchronization conditions among different IPS components.

7. CONCLUSIONS AND FUTURE WORK

Diverse attacks and exploits attempt to gain unauthorized access, reduce the availability of system resources and/or entirely compromise targeted computing systems. IPSs are deployed to detect and block such malicious activities in real-time; their inline mode of operation and delivery of real-time countermeasures make IPS development and more importantly IPS testing a challenge. In this paper, we propose a methodology for IPS testing built around a trace-driven testbed termed the *IPS Evaluator*. The proposed testbed offers an inline working environment for IPSs-under-testing (IUTs) in which IUTs constitute the splicing points between attacker and victim interfaces of test machines. The key objectives of our testing are to help thoroughly investigate attack coverage, verify attack detection/prevention rates and finally determine the behavior of IUTs under various traffic loads.

Our *IPS Evaluator* framework features a number of novel characteristics that include a *bi-directional-feeding* mechanism to inject traffic into the IUTs, dynamic rewriting of source and destination MAC and IP addresses for replayed traffic, use of a *send-and-receive* mechanism to allow for the effective correlation of replayed and forwarded packets, incorporation of IP de-fragmentation and NAT and finally integration of an independent logging mechanism to distinguish packet losses due to network malfunctions from IUT's blocking actions. To maximize the number of replayed packets that are forwarded and subjected to security inspection by IUTs, our testbed partitions packets in traces into two groups: packets originated from the attacker(s) and those from the victim(s). Our testbed is capable of taking into account user-specified conditions to yield more constrained partitioning. We also offer a number of traffic manipulation operations that help shape replayed flows.

We used our proposed methodology to evaluate contemporary IPSs including the *Snort-Inline*, an open source IPS and *FortiGate*, an anti-virus/IPS device. Our testing demonstrated both strengths and weaknesses for *Snort-Inline*; although it offers satisfactory attack coverage and detection rates, *Snort-Inline* generates false positives and negatives under a number of conditions and misses attacks when subject to volumes of heavy traffic. Our approach also helped us locate weaknesses of IPSs related to deep inspection and occasional inconsistency between event logs and actions taking place. We intend to extend our work by providing an automatic attack classification mechanism so that newly discovered attacks can be easily included in our testing; establishing benchmarks and measurements to help compare test results from different IPS testbeds; and integrating our methodology with others to facilitate testing of multi-functional security systems.

ACKNOWLEDGMENTS

We are very grateful to the anonymous reviewers for their valuable comments that helped us significantly improve the presentation of our work. We are also indebted to Minya Chen and Shiyan Hu for discussions on algorithmic aspects and to Joe Zhu, Hong Huang, Ping Wu and Chi Zhang of `Fortinet` for providing traffic traces and many comments on our testing methodology.

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] Cheswick, W., Bellovin, S. and Rubin, A. (2003) *Firewalls and Internet Security* (2nd ed), Professional Computing Series, Addison-Wesley, Boston, MA.

[2] Shieh, S.-P. and Gligor, V. (1997) On a pattern-oriented model for intrusion detection. *IEEE Trans. Knowl. Data Eng.*, **9**, 661–667.

[3] Xinidis, K., Charitakis, I., Antonatos, S., Anagnostakis, K.G. and Markatos, E.P. (2006) An active splitter architecture for intrusion detection and prevention. *IEEE Trans. Dependable Secur. Comput.*, **3**, 31–44.

[4] Valeur, F., Vigna, G., Kruegel, C. and Kemmerer, R.A. (2004) A comprehensive approach to intrusion detection alert correlation. *IEEE Trans. Dependable Secur. Comput.*, **1**, 146–169.

[5] Bass, T. (2000) Intrusion detection systems and multisensor data fusion: creating cyberspace situational awareness. *Commun. ACM*, **43**, 99–105.

[6] Kiam, Y., Lau, W.C., Chuah, M.C. and Chao, H.J. (2006) PacketScore: a statistics-based packet filtering scheme against distributed denial-of-service attacks. *IEEE Trans. Dependable Secur. Comput.*, **3**, 141–155.

[7] Mirkovic, J. and Reiher, P. (2005) D-WARD: A source-end defense against flooding denial-of-service attacks. *IEEE Trans. Dependable Secur. Comput.*, **2**, 216–232.

[8] Yuan, J. and Mills, K. (2005) Monitoring the macroscopic effect of DDoS flooding attacks. *IEEE Trans. Dependable Secur. Comput.*, **2**, 324–335.

[9] Wang, H., Zhang, D. and Shin, K.G. (2004) Change-point monitoring for the detection of DoS attacks. *IEEE Trans. Dependable Secur. Comput.*, **1**, 193–208.

[10] Yee, A. (2003) Network intrusions: from detection to prevention. *Inform. Secur. Bull.*, **8**, 11–16.

[11] RFC1631 (1994) *The IP Network Address Translator (NAT)*. Internet Engineering Task Force. Tokyo, Japan.

[12] Malan, G.R., Watson, D., Jahanian, F. and Howell, P. (2000) Transport and Application Protocol Scrubbing. *Proc. INFOCOM Conf.*, Tel Aviv, Israel, March, pp. 1381–1390. IEEE.

[13] Ptacek, T. and Newsham, T. (1998) Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical Report. Secure Networks, Inc., Alberta, Calgary, Canada.

[14] Group, T.N. (2008) *Intrusion prevention system (IPS) group test*. http://www.nss.co.uk.

[15] http://tomahawk.sourceforge.net (2007) *A methodology and toolset for evaluating network based intrusion prevention systems*. TippingPoint Technologies. http://www.tomahawktesttool.org/resources.html.

[16] Snyder, J., Newman, D. and Thayer, R. (2004) In the wild: IPS tested on a live production network. *Network World Fusion*. Network World, Inc. http://www.networkworld.com. http://www.nwfusion.com/reviews/2004/0216ipsintro.html.

[17] Lippmann, R.P., Fried, D.J., Graf, I., Haines, J.W., Cunningham, K. and Zissman, M.A. (2000) Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. *Proc. DARPA Information Survivability Conf. Exposition: DISCEX-2000*, Los Alamitos, CA, January, pp. 12–26. IEEE Computer Society.

[18] Haines, J.A., Rossey, L.M., Lippmann, R.P. and Cunningham, R.K. (2001) Extending the DARPA Off-Line Intrusion Detection Evaluations. *Proc. DARPA Information Survivability Conf. Exposition (DISCEX-01)*, Anaheim, CA, January, pp. 35–45. IEEE Computer Society.

[19] Turner, A. (2007) *Tcpreplay: Pcap editing and replay tools for UNIX*. http://tcpreplay.synfin.net.

[20] Song, D., Shaffer, G. and Undy, M. (1999) Nidsbench — A Network Intrusion Detection Test Suite. *2nd Int. Workshop on Recent Advances in Intrusion Detection (RAID 1999)*, West Lafayette, IN, September, pp. 1–21. Anzen Computing.

[21] Puketza, N.J., Zhang, K., Chung, M., Mukherjee, B. and Olsson, R.A. (1996) A methodology for testing intrusion detection systems. *IEEE Trans. Softw. Eng.*, **22**, 719–729.

[22] Puketza, N., Chung, M., Olsson, R.A. and Mukherjee, B. (1997) A software platform for testing intrusion detection systems. *IEEE Softw.*, **14**, 43–51.

[23] Antonatos, S., Anagnostakis, K.G. and Markatos, E.P. (2004) Generating Realistic Workloads for Network Intrusion Detection Systems. *Proc. 4rth Int. Workshop on Software and Performance (WOSP'04)*, Redwood Shores, CA, January, pp. 207–215. ACM.

[24] Dawson, S. and Jahanian, F. (1995) Probing and Fault Injection of Protocol Implementations. *Proc. 15th IEEE Int. Conf. Distributed Computing Systems*, Vancouver, BC, Canada, May/June, pp. 351–359. IEEE.

[25] Hall, M. and Wiley, K. (2002) Capacity Verification for High Speed Network Intrusion Detection Systems. *Fifth Int. Symp. Recent Advances in Intrusion Detection (RAID 2002)*, Zurich, Switzerland, October, pp. 239–251. Springer, Berlin/Heidelberg.

[26] Chang, S., Shieh, S.-P. and Jong, C. (2000) A security testing system for vulnerability detection. *J. Comput.*, **12**, 7–21.

[27] Athanasiades, N., Abler, R., Levine, J., Owen, H. and Riley, G. (2003) Intrusion Detection Testing and Benchmarking Methodologies. *Proc. First IEEE Int. Workshop on Information Assurance*, Darmstadt, Germany, March, pp. 63–72. IEEE.

[28] Robert, D., Terrence, C., Brian, W., Eric, M. and Luigi, S. (1999) Testing and evaluating computer intrusion detection systems. *Commun. ACM*, **42**, 53–61.

[29] Schaelicke, L., Slabach, T., Moore, B. and Freeland, C. (2003) Characterizing the Performance of Network Intrusion Detection Sensors. *Proc. 6th Int. Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, Berlin-Heidelberg, New York, September, pp. 155–172. Springer.

[30] MTR-97W096 (1997) *Intrusion Detection Fly-Off: Implications for the United States Navy*. McLean, VA.

[31] Maxion, R. (1998) Measuring Intrusion Detection Systems. *The First Int. Workshop on Recent Advances in Intrusion Detection (RAID-98)*, Louvain-la-Neuve, Belgium, September, pp. 1–41. ACM.

[32] Debar, H. and Wespi, A. (1998) Reference Audit Information Generation for Intrusion Detection Systems. *Proc. 14th Int. Information Security Conf. IFIP SEC'98*, Vienna, Austria, September, pp. 405–417. IFIP.

[33] 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. Inf. Syst. Secur.*, **3**, 262–294.

[34] Maxion, R.A. and Tan, K.M.C. (2000) Benchmarking Anomaly-Based Detection Systems. *1st Int. Conf. Dependable Systems and Networks*, New York, NY, June, pp. 623–630. IEEE.

[35] Mueller, P. and Shipley, G. (2001) Dragon claws its way to the top. *Network Computing*, August, pp. 45–67. United Business Media LLC. www.networkcomputing.com.

[36] Yocom, B. and Brown, K. (2001) Intrusion Battleground Evolves. *Network World Fusion*, October, pp. 53–62. Network World, Inc. http://www.networkworld.com, http://www.nwfusion.com/reviews/2004/0216ipsintro.html.

[37] Vigna, G., Kemmerer, R.A. and Blix, P. (2001) Designing a Web of Highly-Configurable Intrusion Detection Sensors. *Proc. 4th Int. Symp. Recent Advances in Intrusion Detection (RAID 2001)*, Davis, CA, October, pp. 69–84. Springer.

[38] Geer, D. and Harthorne, J. (2002) Penetration Testing: A Duet. *Proc. 18th Annual Conf. Computer Security Applications*, Las Vegas, NV, December, pp. 185–195. IEEE Computer Society.

[39] Arkin, B., Stender, S. and Mcgraw, G. (2005) Software penetration testing. *IEEE Secur. Priv.*, **3**, 84–87.

[40] Security, T.N. (2008) *Nessus: the network vulnerability scanner*. http://www.nessus.org.

[41] Fyodor (2008) Nmap: A Security Scanner. http://www.insecure.org.

[42] LLC, M. (2008) *The metasploit project*. http://www.netasploit.org.

[43] Focus, S. (2004) *BugTraq vulnerability database*. http://www.securityfocus.com.

[44] Shieh, S.-P., Ho, F., Huang, Y. and Luo, J. (2000) Network address translators: effects on security protocols and applications in the TCP/IP Stack. *IEEE Internet Comput.*, **4**, 42–49.

[45] Cormen, T.H., Leiserson, C.E. and Rivest, R.L. (1997) *Introduction to Algorithms*. The MIT Press, Cambridge, MA.

[46] MITRE Organization (2005) *Common vulnerabilities and exposures*. http://cve.mitre.org/.

[47] Sahni, S.K. and Gonzales, T.F. (1976) P-complete approximation problems. *J. ACM*, **23**, 555–565.

[48] Kann, V., Khanna, S., Lagergren, J. and Panconesi, A. (1997) Hardness of approximating MAX K-CUT and its dual. *Chicago J. Theor. Comput. Sci.*, **1997**, 1–18.

[49] Neijens, L. (2008) *The cyberkit network utility*. http://www.gknw.net/cyberkit.

[50] Systems, I.S. (2004) *X-Force Security Center*. http://xforce.iss.net/security_center.

[51] Roesch, M. (1999) Snort – Lightweight Intrusion Detection for Networks. *USENIX 13th Systems Administration Conf. – LISA'99*, Seattle, Washington, USA, November, pp. 229–238. The USENIX Assoication.

[52] Willigner, W., Taqqu, M.S. and Erramilli, A. (1996) A Bibliographical Guide to Self-Similar Traffic and Performance Modeling for Modern High-Speed Networks. In Kelly, F.P., Zachary, S. and Ziedins, I.(eds) *Stochastic Networks: Theory and Applications*, **Vol. 4**, pp. 339–366.

[53] InSecure (2008) *On the definition of false positive*. http://seclists.org/focus-ids/2005/Oct/0102.html.

[54] Afonso, J., Monteiro, E. and Costa, V. (2006) Development of an Integrated Solution for Intrusion Detection: A Model Based on Data Correlation. *Proc. Int. Conf. Networking and Services*, Silicon Valley, CA, July, **37**. IEEE Computer Society.

[55] Dreger, H., Feldmann, A., Mai, M., Paxson, V. and Sommer, R. (2006) Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. *Proc. 15th USENIX Security Symp.*, Vancouver, BC, Canada, July–August, pp. 257–272. USENIX.

[56] The Whisker Project (2004) *Libwhisker: a Perl module for HTTP testing*. http://sourceforge.net/projects/whisker.

[57] Tool, T.S. (2004). *SideStep: IDS evasion tool*. http://www.robertgraham.com/tmp/sidestep.html.

[58] Inc, F.. (2007). *FortiGate: an anti-virus and intrusion prevention system*. http://www.fortinet.com.

[59] Chen, Z., Wei, P. and Delis, A. (2008) Catching remote administration trojans. *Softw. Pract Exp.*, **38**, 667–703.

[60] Sanfilippo, S. (2008) *Hping: an active network security tool*. http://www.hping.org.

[61] Staniford, S., Hoagland, J.A. and Mcalemey, J.M. (2002) Practical automated detection of stealthy portscans. *J. Computer Secur.*, **10**, 105–136.

[62] MIT Lincoln Laboratory (2008) *DARPA intrusion detection evaluation data sets*. http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html.

APPENDIX 1: TOMAHAWK

`Tomahawk` is a command-line IPS testing tool developed by TippingPoint for testing IPS devices as well as network and security performance evaluation [15]. Each test machine in `Tomahawk` is equipped with three NICs: two cards (*eth0* and *eth1*) connect to the internal and external ports of the IUT while the third NIC acts as management and control channel. `Tomahawk` employs a trace-driven method to conduct IPS tests that honors packet orders in traces during the feeding process. To determine the injection direction for a packet, `Tomahawk` divides trace packets into those initiated by the internal network and the rest originating from the external network; the former is replayed via NIC *eth0* while the latter via *eth1*. Packets are parsed by the `Tomahawk` sequentially and partitioned exclusively based on their appearance order in traffic traces. An IP address is considered to be external if it acts as an source address in its very first appearance in a trace. Similarly, an IP is treated as an internal address, if it is first encountered as a destination address in a trace. `Tomahawk` re-transmits a packet after a default 0.2 s timeout period elapses in order to deal with packet-loss due to actions taken by the IUT. To ensure that injected packets are forwarded and subject to security inspection by IUTs, `Tomahawk` rewrites the MAC addresses of replayed packets on-the-fly. Moreover, each packet's source/destination IP addresses are also rewritten and the packet's checksum is updated accordingly. `Tomahawk` uses either pipelining or parallel replay to generate high volume traffic when IPS stress-testing takes place. It also provides mechanisms to accurately control the bandwidth consumed by each test machine and concurrent connections.

`Tomahawk` offers a testbed that helps conduct basic tests for IPS performance evaluation. However, `Tomahawk` cannot test IPS functionalities when the IUT function in routing mode as it provides no address resolution capability on MAC and IP associations. Its simplistic traffic partitioning method tends to generate packets that are un-forwardable to IUTs. Suppose that the first three packets of the Cyberkit in Table 3 is (67.117.243.204, 67.117.44.225), (67.119.190.203, 67.117.243.205), and (67.117.243.204, 67.119.190.203). Here, IP address pair (IP_{src}, IP_{dst}) represents a packet from IP_{src} to IP_{dst}. After processing the first two packets, `Tomahawk` establishes that both IP addresses 67.117.243.204 and 67.119.190.203 belong to the internal network. When replaying packet 3, `Tomahawk` rewrites it to have identical source and destination MAC addresses simply due to the fact that both its source and destination IP addresses are bound to the internal network. Hence, the IUT declines to forward packet 3 and imposes no security inspection on it causing a false negative. Moreover, `Tomahawk` features no IP de-fragmentation mechanism and consequently it cannot evaluate the capabilities of IPSs with respect to traffic normalization and evasion resistance. Finally, the `Tomahawk` provides no capabilities for manipulating replayed packets so that the ensued traffic can be shaped to display characteristics such as specific traffic intensity, protocol mixture, and attack density.

APPENDIX 2: TCPREPLAY

`Tcpreplay` is a suite of utilities that help in the testing of network devices such as IDSs/IPSs [19]. Following a trace-driven methodology, Tcpreplay injects a captured trace to devices under testing through either one or two NICs. In dual NIC replay mode, packets in a trace are classified into client or server initiated according to their origin. Before replay, some protocol fields at data-link, network and transport layers can be rewritten so that the resulting data streams are forwarded and inspected by the IUT. The main utilities of Tcpreplay suite include: (i) *tcpprep*: a tool that determines the origin of a packet and classifies packets into two groups—client-and server-initiated, (ii) *tcprewrite*: an editor for traffic traces that can rewrite some protocol fields in TCP/IP packet headers and (iii) *tcpreplay*: a utility that feeds IUTs with traffic traces via the two NICs at arbitrary speeds. Utility *tcpreplay* also takes into account the manipulation effects on the packet streams by other tools such as *tcpprep* and *tcprewrite*.

To ensure that client-initiated traffic indeed goes through the IUT in one direction while server-originated traffic traverses the opposite, the *tcpprep* resorts to heuristic rules. For a replayed packet to be processed correctly and subject to security inspection by the IUT, it has to be IUT-forwardable. *Tcprewrite* helps in this direction as it can change the source and destination MAC addresses of packets in traces. Furthermore, *tcprewrite* also allows to map IP addresses from one subnet to another subnet. The utility *tcprewrite* supports limited TCP/UDP editing as far as ports, packet sizes, and checksums are concerned. With the help of *tcpprep* and *tcprewrite*, *tcpreplay* may replay the rewritten trace with a specified speed. The trace can be injected as quickly as the network infrastructure of the test environment permits, at fixed pace (packets or bits per second), or at rates proportional to its original speed. To control the replayed period, *tcpreplay* can be instructed to replay the same trace multiple times.

Compared to the Tomahawk, the packet partitioning method of Tcpreplay may generate more viable packet classifications and yields more packets that are IUT-forwardable. In addition, Tcpreplay may rewrite some protocol fields before a packet is replayed. However, Tcpreplay employs ‘send-without-receive’ replay policy and provides no mechanism to record the security performance of IUTs. Thus, it cannot accurately evaluate attack coverage, detection/prevention accuracy, and traffic normalization of IUTs without heavy manual intervention. Similar to Tomahawk, the Tcpreplay does not perform IP de-fragmentation and NAT. This lack in capability renders Tcpreplay ineffective when it comes to testing the resistance of IUTs to evasive attacks. Tcpreplay cannot derive test cases or attack variants based on existing traffic traces. This puts a burden on testers who have to manually generate and capture all test-cases in real-world environments. Finally, Tcpreplay can only test IUTs when the latter work in switching mode as it does not have any address resolution functionality.

APPENDIX 3: GROUP-BASED TESTING

The population of attacks and their variants expands exponentially every year. For instance, *CVE* dictionary contains 15 107 vulnerabilities and exposures in 2005, but increases to 30 000 in 2007. Similarly, the number of attack signatures employed in *Snort-Inline* also enlarges steadily and it is 4 637 in version *v.2.3.2*. In our testbed, we attempt to use group-based testing method to generate test cases instead of the traditional enumeration-based method as the latter has become impractical. By classifying attacks into groups and testing IPSs with representative attacks selected from each group rather than the entire attack repertoire, we expect to reduce the number of test cases and consequently improve testing efficiency. In the group-based testing, the n attacks are first classified into k groups with each group n/k attacks; an attack is selected from each group and used to test the IPS. If it successfully detects the attack, the IPS is considered to be able to identify other attacks in the group. In case that the IPS fails to detect the selected attack, it is further tested by using every attack in the group.

To compute the number of test cases N generated in the group-based testing method, we assume that the IPS can detect a given attack with probability p. We further assume that each attack trace is used X times on average, then X is a random variable with probability distribution q: $q = p$ when $X = k/n$ and $q = 1 - p$ when $X = (n + k)/n$. The expectation of X is $E(X) = pk/n + (1 - p)(n + k)/n = 1 - \mathrm{p} + k/n$, therefore, $N = nE(X) = (1 - p)n + k$. When enumeration-based testing method is employed, the number of test cases is n as each attack is used for once. The group-based method is more efficient than enumeration-based method when $N < n$, which can be easily manipulated into $k < pn$. For instance, when $p = 0.9$ and $n = 15\,107$, group-based testing method generates fewer test cases as long as attacks are classified into less than 13 597 groups. In our testbed, the number of groups k can be adjusted with the help of our proposed hierarchical classification scheme for attack categorization.