A Tool for Offline and Live Testing of Evasion Resilience in Network Intrusion Detection Systems

Leo Juan[1], Christian Kreibich[2], Chih-Hung Lin[1], and Vern Paxson[2]

[1] Institute For Information Industry, Taipei City, Taiwan
{lichou,chlin}@nmi.iii.org.tw
[2] International Computer Science Institute, Berkeley, USA
{christian,vern}@icir.org

Abstract. Network intrusion detection systems (NIDS) face a difficult, fundamental problem in the degree to which attackers can exploit ambiguities present when monitoring network traffic in order to undermine the correctness of the NIDS's analysis to evade detection. However, many of today's NIDSs lack the additional mechanisms required to resist different forms of evasion, because the underlying problems are subtle and—critically—*not visible* to the customers who purchase these systems.
Remedying this common shortcoming of modern NIDS functionality requires the widespread availability of *test suites* oriented towards probing the degree to which a NIDS exhibits evasion vulnerabilities. In this work we undertake the creation of a framework to facilitate the development of such test suites. Our prototype system, `idsprobe`, takes as input a packet trace and from it constructs a configurable set of variant traces that introduce different forms of ambiguities that can lead to evasions. Our test harness then uses these variant traces in either an *offline configuration*, in which the NIDS under test reads traffic from the traces directly, or a *live* setup, in which we employ replay technology to feed traffic over a physical network past a NIDS reading directly from a network interface, and to potentially live victim machines. Summary reports of the differences in NIDS output tell the analyst to what degree the NIDS's results vary, reflecting sensitivities to (and possible detections of) different evasions. We sketch the overall architecture of the framework, discuss the technical components it uses, demonstrate its use for two popular open-source NIDSs, and explore areas for future work to further refine the approach and increase its power.

1 Introduction

Network intrusion detection systems (NIDS) monitor network traffic in order to determine the significance of the observed activity in terms of potential threats and successful exploits. However, such monitoring faces a difficult, fundamental problem: the traffic as observed by an intermediary such as a NIDS does *not* necessarily appear to the recipient in the same semantic terms. Instead, the recipient may either observe a *different* pattern of traffic (particularly, a subset if the network or the recipient's kernel discards some of the packets), or may impose an alternative

interpretation on ambiguous traffic (such as the arrival of two packets spanning the same sequence range in a TCP flow, but offering different payload bytes for that sequence).

Attackers can exploit such ambiguities to confuse the operation of the NIDS, rendering it prone to either missing attacks or reducing the precision of the NIDS's analysis from pinpointing a specific attack to simply noting that the traffic stream includes an ambiguity. Since unfortunately such ambiguities also arise in traffic streams for benign reasons, their presence generally does not constitute an "actionable" determination: the analyst must at a minimum spend considerable effort attempting to ascertain whether the condition constitutes an actual threat.

Thus, evasion attacks leverage an inherent analysis difficulty present in observing network traffic from a location other than one of the endpoints. These ambiguities render it hard, or even impossible, for a NIDS to correctly interpret skillfully crafted packet sequences in the same fashion as the end host receiving them. Such attacks can exploit differing interpretations of traffic at multiple protocol levels. From the application layer's point of view, it is generally not possible to pinpoint the exact location in the protocol stack where the ambiguity was introduced: for a web server, it might have been within HTTP itself, but could just as well have occurred due to TCP retransmissions (layer 4) or IP fragmentation (layer 3).

In a seminal paper [1], Ptacek and Newsham describe several network- and transport-layer attacks that lead to different payload streams perceived by the end-system and the NIDS. Approaches that alleviate the problem exist (e.g., normalization [2] and active mapping [3]), but have not seen large-scale deployment, and do not remedy the problem in its full generality.

Given the fundamental significance of evasion attacks for network intrusion detection, it is striking how little has been documented regarding the efficacy with which modern NIDS address the threat. Users considering which NIDS products to purchase can find extensive performance testing results regarding the breadth of attacks detected by a given system, or its ability to process line-rate traffic, but little information regarding its resilience to evasion—and, in particular, insight into a system's architectural strengths and weaknesses in this regard. Thus, vendors feel little market pressure to enhance their products' strengths in this regard. Because evasion constitutes a fundamental problem, however, for vendors to ignore it risks building a "house of cards": their products increas-

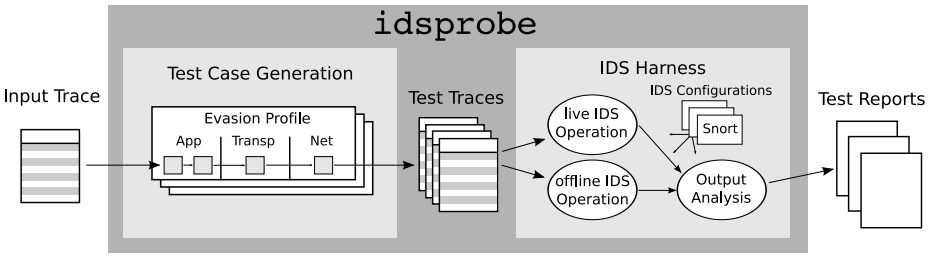

Fig. 1. Architecture of the `idsprobe` framework.

ingly provide more of an appearance of security than a reliable foundation.

Recently, third-party testing of NIDS products has begun to include an assessment of evasion vulnerabilities [4]. This testing environment, however, is proprietary: it is not available for inspection, modification and extension by others. In this work, we argue that there is significant utility for the network security community at large to have an easy-to-use, transparent, open-source environment for testing NIDS for resilience in the presence of evasion.

To this end, we have designed and implemented a framework, termed `idsprobe`, to facilitate the creation of evasion test-cases in a pluggable fashion, coupled with fully automated testing of different NIDS on the resulting test-cases. In the next section, we describe the requirements that guided our system development. In § 3 we present the architecture of the overall framework, and in § 4 some initial experimental results obtained with using it. We discuss related work in § 5, and offer final thoughts as well as a look at important future work in § 6.

2 Requirements

For our evasion-testing environment we consider two sets of requirements: creating test cases, and then applying those test cases to evaluate a given NIDS.

For test-case creation, we have the following considerations:

– The framework should support both trace-based test cases (that is, we construct packet trace files that we then analyze with a NIDS in an off-line fashion) and live network operation.

Trace-based test cases offer very large advantages in terms of repeatability, portability, and ease of inspection and verification of correctness. Thus, when possible we prefer use of traces to use of live traffic. However, some forms of evasion testing *require* live testing. These include: *(i)* NIDS that when responding to possible evasions incorporate information they gain from end systems, either by proactive probing [3], direct feedback from the hosts [5], or passive fingerprinting [6]; *(ii)* NIDS that employ some form of traffic *modification* to remove ambiguities to prevent evasions from exploiting them [2, 7]; and *(iii)* evasion attacks that rely on *resource exhaustion* such as overwhelming the NIDS's available processing or memory resources, thus causing it to drop packets and consequently miss an attack.

- The framework should provide modular, pluggable building blocks for creating complex evasive patterns out of elementary traffic transformations. Related to this, we desire that the framework encourage others in the network security community to contribute test components. By emphasizing modularity and separate plug-ins, we can sustain a low "barrier to entry" for others to contribute to the system and its suite of test cases.
- The framework needs to accommodate elementary traffic transformations across the relevant layers of the protocol stack. In addition, these transformations should be amenable to *composition*. For example, a single test case might include network-layer (e.g., fragmentation), transport-layer (e.g., ambiguous TCP retransmissions) and application-layer (e.g., ambiguous HTTP character encoding) evasions all together, as a way of detecting when a NIDS's evasion countermeasures suffer from feature-interaction.

To use the resulting test cases for evaluating a NIDS, we desire:

- Reusability of the generated test traces for live testing. On-the-fly introduction of evasive actions to live traffic is complicated by the fact that such evasions have to be both selective about the packets they are applied to, and carefully sequenced in order to work. The ability to leverage input traces containing ready-made evasions in live environments therefore yields substantial reduction of effort and improved reliability.
- Automation of the process of executing the NIDS and capturing its full set of outputs (log files, alarms, *stdout*, *stderr*).

– Automated generation of summaries of the differences in NIDS behavior given a non-evasive "base case" versus an evasive test case, as evidenced in the outputs it produces.
– Suitable postprocessors to inspect these differences to highlight patterns corresponding to susceptibility to or thwarting of evasion attempts, particularly to shed light on *architectural* issues reflected in the results (such as whether a given NIDS lacks sufficient state to detect inconsistent retransmissions in full generality, even though it can detect certain instances that require less state to track).

Given these requirements, we now turn to an architecture that attempts to meet them to a large degree.

3 Framework Architecture

Figure 1 illustrates the current architecture of the `idsprobe` framework, which accommodates both offline and live testing.

3.1 Overview

For simplicity, we limit the presentation of the framework to reflect a single set of related test cases. (In general, we use the framework to construct multiple such sets.) The process begins with a single, non-evasive trace (shown on the left) which contains some attribute, such as a particular payload string in a particular context, for which we can configure a NIDS to detect its presence. We then repeatedly apply a series of *transformation profiles* to copies of this trace to yield a set of variants, each of which reflects a particular potential evasion. After generating these traces, we then employ a "test harness" to run a set of NIDS-under-test against the traces (including the original, unmodified trace), capturing their outputs, from which we then construct a set of reports summarizing the NIDS's behavior in the presence of different evasions.

Given this overview, in the next three subsections we discuss in more detail the generation of test cases in the `idsprobe` framework, followed by the testing process based on the resulting traces.

3.2 Test Case Generation

To support modularity, we encapsulate a set of elementary transformations in scripts that can be individually invoked and then subsequently

```
@load http-reply

redef rewriting_http_trace = T;

event http_request(c: connection, method: string,
                   original_URI: string, unescaped_URI: string,
                   version: string)
        {
        rewrite_http_request(c, method, gsub(original_URI, /\//,
                                             "slash"), version);
        }
```

Fig. 2. Example of altering application-level trace contents using Bro's trace-transformation framework.

composed. Each script takes as input (from a file or *stdin*) a `libpcap` trace and produces as output a new trace (to a given output file or *stdout*). In addition, the `idsprobe` framework transparently manages any temporary storage a script requires to perform the transformation, which facilitates chained application of transformations.

We currently provide tools for the following transformations:

Application layer. We support rewriting of application-layer contents using the framework developed in our previous work [8] built upon the Bro intrusion detection system [9]. This framework allows application-level specification of trace transformations that are then reflected down to the transport layer (adjustment of sequence numbers, checksums, and acknowledgments) and network layer (original packetization preserved if possible, new packets inserted if necessary).

For example, Figure 2 shows a Bro script that changes any occurrence of a slash ("`/`") to the text "`slash`".

Transport layer. This level currently supports adjustment of relevant header control bits (e.g., TCP SYN/FIN/RST/ACK/PSH/URG), payload modifications, and correction (or miscorrection) of checksums. We implement these using plug-ins for *Netdude* [10].

Network layer. Our current support for network-layer modifications—also based on Netdude plug-ins—support:

- Modification of arbitrary header fields.
- Duplication/insertion/removal of individual packets.
- IP fragmentation.
- Checksum correction.

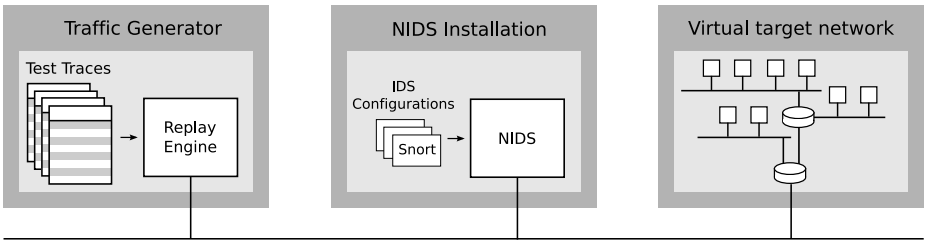

Fig. 3. `idsprobe` setup for live testing.

Trace file manipulation. We provide additional plug-ins to *(i)* adjust packet timestamps in trace files, *(ii)* correct the flow of time (sort packets with non-monotonic timestamps), and *(iii)* recombine multiple sets of packets/traces into a single trace file.

While our implementation of these transformations leverage Bro and Netdude, we emphasize that the scripting interface to the transformation tools can readily accommodate other tools that can provide trace manipulation at different semantic levels.

Finally, we also note a somewhat subtle point regarding composition of different evasions. When generating test cases that combine multiple types of evasions, it is important to apply these "top down" in terms of network protocol layering. That is, we must first apply application-layer transformations, then transport-layer, and finally network-layer. The reason for this is that tools that manipulate one layer generally assume that the lower layer is unambiguous (and thus the tool is free to rewrite it accordingly). If we apply transformations in a different order, we might lose evasions introduced at a lower layer when we later rewrite a set of packets at a higher network layer. (Ideally, our framework would detect such violations of top-down transformation, but at present it does not.)

3.3 Offline Evasion Testing

Once a set of test traces have been generated, the `idsprobe` framework then enables automating of assessing a number of NIDS against the suite. Adding a NIDS is a simple process: all that is required is to provide a shell script that will invoke the NIDS given a number of environment variables including, among others, the trace file to be analyzed.

The test harness then invokes the script repeatedly to execute the NIDS across each of the traces in the variant set, storing the files gen-

erated by each such execution in a separate per-trace directory. After execution, `idsprobe` invokes file-differencing (via the Unix *diff* utility) to determine the degree to which the NIDS's behavior changed for given variants. We pattern this process on that used for Bro's regression testing, which includes notions of output files to skip and canonicalizations to apply to files prior to differencing to remove insignificant differences.

Once differenced, the results currently require manual inspection to assess their significance. For some tests, the NIDS may correctly generate different output—for example, a warning about a possible evasion, or the correct suppression of an alert if by some other means (such as passive fingerprinting) the NIDS has determined that the attempted evasion has rendered the attack ineffective.

3.4 Live Evasion Testing

As mentioned in Section 2, some forms of evasion testing require live tests. To facilitate these, we extended the `idsprobe` framework to function in live environments, while allowing us to re-use the evasive test traces generated for offline testing whenever possible. The architecture of our live testing setup is shown in Figure 3. It consists of three components, connected via a physical link: *(i)* a traffic generator, which establishes connections to the victim machine(s) and drives the data exchange; *(ii)* a NIDS installation which monitors the link; and *(iii)* a virtual target network which hosts the victim machines, responding to the traffic sent by the traffic generator. We now describe the technical realization of each of those components in turn.

The key challenge for the traffic generator is enabling re-use of the existing test traces. Involving packet traces in live environments inevitably requires replay technology; moreover, we require *adaptive* replay which uses the input trace as a guideline for the traffic originator, driving the data exchange with the victim while robustly aligning the victim's traffic with the recorded responder traffic in the input trace. Our approach is to rely on the causality of exchanged application data units (ADUs) at the application level and to ignore the actual content of the responder's ADUs, while patching up the sequence and acknowledgement numbers in the input trace's packets to keep the TCP exchange working. We used the `scapy` packet processing tool [11] to build this replay functionality.

The NIDS installation uses the same IDS configurations as used for offline testing and output collection, except for the obvious difference of sniffing live traffic instead of reading packets from an input trace file.

```
08:00:09.143206 IP 10.48.0.1.2010 > 10.48.0.81.80: S 2345:2345(0) win 32768 <mss 1024>
08:00:09.143306 IP 10.48.0.81.80 > 10.48.0.1.2010: S 865241303:865241303(0) ack 2346 win 5840 <mss 1460>
08:00:09.161454 IP 10.48.0.1.2010 > 10.48.0.81.80: . ack 1 win 32768
08:00:09.176192 IP 10.48.0.1.2010 > 10.48.0.81.80: . 1:14(13) ack 1 win 32768
        0x0000:  4500 0035 f178 0000 4006 7499 0a30 0001  E..5.x..@.t..0..
        0x0010:  0a30 0051 07da 0050 0000 092a 3392 88d8  .0.Q...P...*3...
        0x0020:  5010 8000 f192 0000 4745 5420 2f31 6162  P.......GET./1ab
        0x0030:  6364 6566 67                             cdefg
08:00:09.176192 IP 10.48.0.1.2010 > 10.48.0.81.80: . 2:14(12) ack 1 win 32768
        0x0000:  4500 0034 f178 0000 4006 749a 0a30 0001  E..4.x..@.t..0..
        0x0010:  0a30 0051 07da 0050 0000 092b 3392 88d8  .0.Q...P...+3...
        0x0020:  5010 8000 8942 0000 4554 202f 3161 6263  P....B..ET./1abc
        0x0030:  6465 6667                                defg
08:00:09.176264 IP 10.48.0.81.80 > 10.48.0.1.2010: . ack 14 win 5840
08:00:09.176846 IP 10.48.0.81.80 > 10.48.0.1.2010: P 1:455(454) ack 14 win 5840
08:00:09.176912 IP 10.48.0.81.80 > 10.48.0.1.2010: F 455:455(0) ack 14 win 5840
08:00:09.368323 IP 10.48.0.1.2010 > 10.48.0.81.80: . ack 456 win 32768
08:00:09.396305 IP 10.48.0.1.2010 > 10.48.0.81.80: F 14:14(0) ack 456 win 32768
08:00:09.396339 IP 10.48.0.81.80 > 10.48.0.1.2010: . ack 15 win 5840
```

Fig. 4. *tcpdump* output for the TC2 input trace. A single TCP segment contains the relevant application-layer content, "`GET /1abcdefg`" at sequence number 2346 (0x092a). It is followed by a duplicate segment that misses the first byte, but starts at sequence number 2347 (0x092b), thus benignly overlapping right-aligned with the first segment. (We have omitted full packet contents of other packets for brevity.)

We used `honeyd` [12] to realize the virtual target network. `honeyd` provides the major benefits of allowing easy adjustment of the network topology (for example in order to introduce additional routers for reachability evasions relying on the IP TTL field), while providing flexible victim responder configurations ranging from simple shell scripts to forwarding to live external systems via `honeyd`'s subsystem mechanism.

4 Initial Experimental Results

As a preliminary evaluation of the `idsprobe` framework, we developed an initial set of 8 different types of test cases. We evaluated each against the *Snort* [13] (version 2.6.1.4) and *Bro* [9] (version 1.2.1) NIDSs.

4.1 Test Cases

In all test cases, we use a set of traces of entire, full-packet TCP connections. Each contains a single HTTP request with lengths ranging from 8 to 256 bytes, and a corresponding HTTP response. The main objective is to determine whether the NIDS under test can match a signature that we *know* is present in the generated, evasive traffic, while also checking for any signs of evasion or other unusual activity that the NIDS might signal. `idsprobe` automatically generated 196 test traces based on 5 input traces.

The test cases cover the following sets of transformations:

– TC1: We introduce a single, consistent, and immediate retransmission of a TCP segment carrying the signature-bearing application-

layer payload. I.e., we locate the packet in the trace, and add an exact copy of it with a timestamp immediately (1 μsec) following. Such duplications occur in actual network traffic for benign reasons and do not represent any sort of actual threat or ambiguity. Thus, this test case checks whether the NIDS performs a simple form of TCP stream reassembly correctly.

- TC2: Like TC1, but the retransmission consists of only part of the original TCP segment. More precisely, we retransmit a right-aligned part of the original segment with correct checksum, as follows: a segment containing at least 2 bytes of payload is duplicated to immediately follow the original, shortened by 1 byte, and its sequence number incremented by 1. This constellation likewise presents neither threat nor ambiguity. Figure 4 illustrates this arrangement.
- TC3: Like TC1, but we change the TCP payload on the first (subtest TC3a) or the second (subtest TC3b) variant of the duplicated packet, respectively, without any checksum corrections. For a correctly functioning NIDS, this test does not pose any actual ambiguity; it should simply discard the variant with the invalid checksum.
- TC4: Like TC3, except now the checksums are corrected. Our payload modification is *careless*, thus leading to a different checksum value. This test case represents the first truly ambiguous traffic. The NIDS needs to decide which version of the byte stream to analyze, and ideally should note the inconsistency. The semantics of this connection are not well-formed, and it would be reasonable for a reactive NIDS to terminate the connection.
- TC5: Like TC4, but we change the TCP payload *carefully*, leaving the checksum unchanged. We achieve this by swapping 16-bit fields, though one could derive more complex modifications due to the incremental nature of the checksumming algorithm. As with TC3, this presents a real ambiguity, though the NIDS will have to compare the actual payloads in order to notice the difference.
- TC6: We duplicate one of the IP datagrams in the TCP flow, setting its IP fragment offset to a non-zero offset value (adjusting the IP header checksum to reflect the change) on the first (subtest TC6a) or second (subtest TC6b) variant, respectively. This test case creates an ambiguous, malformed fragment.

  One interpretation the NIDS might use is to treat the zero-offset (and complete) version of the datagram as correct, and to eventually discard the second version as an incomplete datagram. Another interpre-

tation would be to treat the two datagrams as overlapping, inconsistent fragments.
- TC7: We duplicate one of the IP datagrams in the TCP flow and set its IP TTL value to a number of different values (again with header checksum updated) on the first (subtest TC7a) or second (subtest TC7b) variant, respectively. This test case does not introduce a serious ambiguity (since the contents of the datagrams remain the same), but can confuse NIDS evasion detection that examines TTL values for anomalies.
- TC8: We consistently fragment one of the IP datagrams in the TCP flow carrying the signature-bearing payload, using various different fragment sizes. This test case tests whether the NIDS correctly processes well-formed fragments.
- TC9: Like TC8, but we duplicate one of the fragments, and alter its payload in the first (subtest TC8a) or second (subtest TC8b) variant, respectively. The alteration is again careless, i.e., reassembly of the datagram using the modified payload leads to an incorrect TCP checksum for the full datagram.
- TC10: Like TC9, but with a careful payload alteration, i.e., reassembly of the datagram using the modified payload leaves the TCP checksum unchanged. Figures 5 and 6 present the workings of TC10 in detail.

4.2 NIDS Configurations

NIDS provide varying degrees of configurability. Therefore, assessors need to carefully consider the configurations they wish to evaluate, as these can have quite differing effects on the performance of the NIDS in the presence of possible evasions.

For the Bro NIDS, we used its default configuration settings. We instructed it to monitor all TCP traffic (`-f tcp`) and loaded the `mt`, `frag`, and `signatures` analyzers. We configured signatures the HTTP requests in the input traces, with each signature matching exactly one of the HTTP requests.

Snort's default configuration file needed more editing before Snort would accept it (i.e., run without error). We disabled database logging, which was enabled by default but caused start-up failures since our environment does not have a database set up. We removed the large list of signature file `include` directives, since none of the listed rule sets

```
00:01:37.391023 10.48.0.1.2013 > 10.48.0.81.80: S 2345:2345(0) win 32768 <mss 1024>
00:01:37.391107 10.48.0.81.80 > 10.48.0.1.2013: S 943214166:943214166(0) ack 2346 win 5840 <mss 1460> (DF)
00:01:37.411887 10.48.0.1.2013 > 10.48.0.81.80: . ack 1 win 32768
00:01:37.427628 10.48.0.1.2013 > 10.48.0.81.80: . 1:158(157) ack 1 win 32768
0x0000   4500 00c5 7566 0000 4006 f01b 0a30 0001        E...uf..@....0..
0x0010   0a30 0051 07dd 0050 0000 092a 3838 4e57        .0.Q...P...*88NW
0x0020   5010 8000 0fc2 0000 4745 5420 2f31 6162        P.......GET./1ab
0x0030   6364 6566 6768 696a 6b6c 6d6e 6f70 7172        cdefghijklmnopqr
0x0040   7374 7576 7778 797a 3261 6263 6465 6667        stuvwxyz2abcdefg
0x0050   6869 6a6b 6c6d 6e6f 7071 7273 7475 7677        hijklmnopqrstuvw
0x0060   7879 7a33 6162 6364 6566 6768 696a 6b6c        xyz3abcdefghijkl
0x0070   6d6e 6f70 7172 7374 7576 7778 797a 3461        mnopqrstuvwxyz4a
0x0080   6263 6465 6667 6869 6a6b 6c6d 6e6f 7071        bcdefghijklmnopq
0x0090   7273 7475 7677 7879 7a35 6162 6364 6566        rstuvwxyz5abcdef
0x00a0   6768 696a 6b6c 6d6e 6f70 7172 7320 4854        ghijklmnopqrs.HT
0x00b0   5450 2f31 2e31 0d0a 484f 5354 3a6e 6f6e        TP/1.1..HOST:non
0x00c0   650d 0a0d 0a                                   e....
00:01:37.427676 10.48.0.81.80 > 10.48.0.1.2013: . ack 158 win 6432 (DF)
00:01:37.428396 10.48.0.81.80 > 10.48.0.1.2013: P 1:575(574) ack 158 win 6432 (DF)
00:01:37.428499 10.48.0.81.80 > 10.48.0.1.2013: F 575:575(0) ack 158 win 6432 (DF)
00:01:37.601154 10.48.0.1.2013 > 10.48.0.81.80: . ack 576 win 32768
00:01:37.619894 10.48.0.1.2013 > 10.48.0.81.80: F 158:158(0) ack 576 win 32768
00:01:37.619929 10.48.0.81.80 > 10.48.0.1.2013: . ack 159 win 6432 (DF)
```

Fig. 5. *tcpdump* output for the TC10 input trace. A single TCP segment contains the relevant application-layer content, "`GET /1abcdef`".

```
00:01:37.391023 10.48.0.1.2013 > 10.48.0.81.80: S 2345:2345(0) win 32768 <mss 1024>
00:01:37.391107 10.48.0.81.80 > 10.48.0.1.2013: S 943214166:943214166(0) ack 2346 win 5840 <mss 1460> (DF)
00:01:37.411887 10.48.0.1.2013 > 10.48.0.81.80: . ack 1 win 32768
00:01:37.427628 10.48.0.1.2013 > 10.48.0.81.80: [|tcp] (frag 30054:8@0+)
00:01:37.427629 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@8+)
00:01:37.427630 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@16+)
0x0000   4500 001c 7566 2002 4006 d0c2 0a30 0001        E...uf..@....0..
0x0010   0a30 0051 0fc2 0000 4745 5420                  .0.Q....GET.
00:01:37.427631 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@24+)
0x0000   4500 001c 7566 2003 4006 d0c1 0a30 0001        E...uf..@....0..
0x0010   0a30 0051 2f31 6162 6364 6566                  .0.Q/1abcdef
00:01:37.427632 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@24+)
0x0000   4500 001c 7566 2003 4006 d0c1 0a30 0001        E...uf..@....0..
0x0010   0a30 0051 2f31 6364 6162 6566                  .0.Q/1cdabef
00:01:37.427633 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@32+)
0x0000   4500 001c 7566 2004 4006 d0c0 0a30 0001        E...uf..@....0..
0x0010   0a30 0051 6768 696a 6b6c 6d6e                  .0.Qghijklmn
00:01:37.427634 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@40+)
00:01:37.427635 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@48+)
00:01:37.427636 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@56+)
00:01:37.427637 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@64+)
00:01:37.427638 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@72+)
00:01:37.427639 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@80+)
00:01:37.427640 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@88+)
00:01:37.427641 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@96+)
00:01:37.427642 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@104+)
00:01:37.427643 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@112+)
00:01:37.427644 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@120+)
00:01:37.427645 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@128+)
00:01:37.427646 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@136+)
00:01:37.427647 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@144+)
00:01:37.427648 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@152+)
00:01:37.427649 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@160+)
00:01:37.427650 10.48.0.1 > 10.48.0.81: tcp (frag 30054:8@168+)
00:01:37.427651 10.48.0.1 > 10.48.0.81: tcp (frag 30054:1@176)
00:01:37.427676 10.48.0.81.80 > 10.48.0.1.2013: . ack 158 win 6432 (DF)
00:01:37.428396 10.48.0.81.80 > 10.48.0.1.2013: P 1:575(574) ack 158 win 6432 (DF)
00:01:37.428499 10.48.0.81.80 > 10.48.0.1.2013: F 575:575(0) ack 158 win 6432 (DF)
00:01:37.601154 10.48.0.1.2013 > 10.48.0.81.80: . ack 576 win 32768
00:01:37.619894 10.48.0.1.2013 > 10.48.0.81.80: F 158:158(0) ack 576 win 32768
00:01:37.619929 10.48.0.81.80 > 10.48.0.1.2013: . ack 159 win 6432 (DF)
```

Fig. 6. *tcpdump* output of resulting TC10 evasive trace file. The TCP segment shown in Figure 5 has its application-layer content rewritten, fragmented into 24 8-byte fragments, with a duplicate fragment with the original TCP stream content inserted after the third fragment. The sensitive payload is now spread across three IP datagrams. The payload variation preserves the TCP checksum's validity. Finally, `idsprobe` patches the packet timestamps to preserve chronological ordering.

were actually included in the Snort distribution. We also verified that the `frag3` and `stream4` preprocessors were enabled, and that evasion-related alerts would be generated.

	Output	Test Cases									
		TC1	TC2	TC3a/b	TC4a/b	TC5a/b	TC6a/b	TC7a/b	TC8	TC9a/b	TC10a/b
Bro	Sig. match	✓	✓	✓	✗/✓	✗/✓	✓	✓	✓	✗/✓	✗/✓
	Evasion				①	①				②	②
	Other			③					④	④	④
Snort	Sig. match	✓	✓	✓	✗/✓	✗/✓	✓	✓	✓	✗/✓	✗/✓
	Evasion	⑤	⑥		⑤			⑤		⑦	
	Other										

[1] Bro signaled a "RetransmissionInconsistency".
[2] Bro signaled a "WeirdActivity" of type "fragment_inconsistency".
[3] Bro signaled bad checksums in weird.log.
[4] Bro signaled a "WeirdActivity" of type "excessively_small_fragment" for fragments of 32 bytes or less.
[5] Snort signaled "possible evasive FIN detection" with nonsensical parameters.
[6] Snort signaled "TCP checksum changed on retransmission".
[7] Snort signaled "fragmentation overlap".

Table 1. Summary of the results produced by `idsprobe` on 10 test cases with the Bro and Snort NIDS. The first line for each of the NIDS summarizes whether the signature was detected or not, the second reports any evasion-related alerts or messages, and the third lists any other output.

```
[**] [111:24:1] (spp_stream4) possible EVASIVE FIN detection [**]
03/19-00:00:09.176188 10.48.0.1:2010 -> 10.48.0.81:80
TCP TTL:240 TOS:0x10 ID:0 IpLen:20 DgmLen:77
***AP*** Seq: 0x92A  Ack: 0x33928A9F  Win: 0x8000  TcpLen: 20
```

Fig. 7. Erroneous evasion alert produced by Snort 2.6.1.4 in 4 of the test cases.

4.3 Findings

Output of `idsprobe`-generated traces Table 1 summarizes our findings based on the `idsprobe`-generated evasive packet traces. Overall, Bro and Snort performed similarly as far as signature detection is concerned. They differ, however, in the amount of detail delivered in addition to the relevant alerts. To the extent of our testing, both NIDS deal well with IP fragmentation as well as TCP stream reassembly. More basic issues such as broken IP/TCP checksums are also properly recognized by both. In their default configurations, both Bro and Snort follow a first-seen-first-delivered policy when dealing with IP fragments as well as TCP segments, causing signature detection to fail in those cases where

```
08:00:09.176192 IP 10.48.0.1.2010 > 10.48.0.81.80: . 1:13(12) ack 1 win 32768
        0x0000:  4500 0034 f178 0000 4006 749a 0a30 0001  E..4.x..@.t..0..
        0x0010:  0a30 0051 07da 0050 0000 092a 3392 88d8  .0.Q...P...*3...
        0x0020:  5010 8000 4582 0000 4745 5420 2f31 6162  P...E...GET./1ab
        0x0030:  6364 7878                                cdxx
08:00:09.176194 IP 10.48.0.1.2010 > 10.48.0.81.80: . 11:14(3) ack 1 win 32768
        0x0000:  4500 002b f178 0000 4006 74a3 0a30 0001  E..+.x..@.t..0..
        0x0010:  0a30 0051 07da 0050 0000 0934 3392 88d8  .0.Q...P...43...
        0x0020:  5010 8000 80f0 0000 6566 67              P.......efg
```

Fig. 8. *tcpdump* output of an inconsistent, partially overlapping retransmission that is not reported by Snort 2.8.0.1.

`idsprobe`-generated evasive traffic hides the sensitive payload in the most recently arrived fragments/segments. Both NIDSs allow changing this behavior: while Snort allows the user to specify preferences at the granularity of destination networks,[3] Bro allows policy specification using end-host databases automatically generated via active mapping [3].

Our most striking result is Snort's unreliable reporting of evasion attempts. It both substantially over-reports non-issues while also failing to report real indications of evasive activity.

After excluding from file-differencing Bro's `.state` directories (which remain empty) and Snort's *tcpdump* log files, the total amount of difference in Bro's output amounts to 1,665 lines, as opposed to 17,018 for Snort. After manually inspecting some of the differences reported for Snort, we found that the majority are due to the verbose summary output reported on *stdout* and *stderr*. To aid in analysis of the significance of the differences, we decided to ignore these files in the file-differencing pass and focus on Snort's remaining output files. This change brought Snort's differential data volume down to 1,329 lines.

In TC2, Snort erroneously reported a TCP checksum change on a retransmission, where in fact no divergent payload was transferred. In the event of careful payload alterations that do not affect the TCP checksum (TC5/TC10), however, Snort fails to notice the (rather likely) evasion attempt. Bro handled both cases correctly, remaining silent on the former but alerting on the latter case. Snort also generated a total of 60 potential evasive TCP FIN detections in 4 of the test cases. Figure 7 illustrates one of these. We note that a number of the values reported in these alerts are nonsensical—there were no IP TTL values of 240 in the input traces, nor IP ToS fields with values 0x10, nor IP IDs of 0. In addition, none of the traces reflects an ambiguous TCP FIN packet.

We see that Bro generates significantly more (correct) output regarding the presence in the inputs of possible evasions. These include TCP

[3] Prior to version 2.8, only a global flag was supported.

retransmission inconsistencies, IP fragment inconsistencies, the presence of bad checksums, and the presence of excessively small IP fragments. For Snort, the only correct evasion-related output concerns IP fragmentation overlap.[4]

During the course of our work, new releases of Snort appeared. We experimented with the latest release available, Snort version 2.8.0.1, to see how its behavior might have changed. We noticed that the erroneous evasive FIN alerts have been repaired. However, the new stream reassembly module `stream5` introduced new issues, as evidenced by the fact that a partially overlapping retransmission that extends into sequence number space (as shown in Figure 8) is not reported at all, while Snort 2.6.1.4 did report a changed TCP checksum on the retransmission.

Output after long-term operation The difference in output volume and quality demonstrated by Snort and Bro on `idsprobe`'s test traces suggests that reliably identifying evasion attempts is not yet a feature set that users can assume will be fully and correctly provided by mainstream NIDSs. To better understand the usability of evasion/anomaly-related events reported by different IDSs, we ran Bro 1.2.1 along with Snort 2.6.1.4 and 2.8.0.1 on a 24-hour, 21 GB trace recorded at ICSI on 16 March 2007. In contrast to typical NIDS configurations, the NIDSs were not configured to detect attacks, but only to report anomalous or potentially evasive activity.

Table 2 summarizes our findings. The absence of consensus in the reported events is striking, particularly between Bro and the Snort versions, but to a lesser degree even between two different Snort releases. TCP SYNs with payload data seem a rare case where there is near-consensus, with the three NIDSs reporting 460, 458, and 461 instances, respectively. Bro reports a single retransmission inconsistency (which we have verified to be correct, but it does not reflect a malicious evasion). Snort 2.6 reports this as one of 36,873 "possible EVASIVE RST detection" events, and Snort 2.8 as 3 of the 5 "Data sent on stream after TCP Reset" events recorded. For the 22,137 flow reassembly issues reported by Bro ("ContentGap" and "AckAboveHole"), which have direct significance for content-based analysis, there is no apparent corresponding alert in either of the Snort logs. These events account for the main reason why

[4] Even that is not the best description of the problem, since IP fragments can overlap for rare-but-benign reasons. Better would be to highlight that the overlap is inconsistent.

Snort 2.8 reports fewer events than Bro, whose output volume is almost an order of magnitude below Snort 2.6's.

These results clearly demonstrate the importance of reliable evasion detection in a NIDS, lest the volume of alerts can render the reported events operationally useless as evidence of actual evasions.

5 Related Work

The fundamental problem of NIDS evasion was first framed in the seminal paper by Ptacek and Newsham [1]. They considered ambiguities attackers can use to *insert* or *delete* traffic such that the NIDS's view of activity differs from the recipient's, as well as the threat of attackers imposing denial-of-service on the NIDS itself. Some of these issues also appear in the discussion of the Bro system [9], particularly in the context of inconsistent TCP retransmissions.

In response to the threat of evasion, researchers have developed several types of countermeasures. One approach concerns "normalization" [2] or "scrubbing" [7] of traffic, by which an active network element modifies traffic flowing through it in order to remove classes of ambiguities. These approaches can address many network- and transport-layer evasion threats, but have more difficulty addressing application-layer threats.

A different strategy, *active mapping*, seeks to proactively determine how recipients of network traffic (or the network traffic between them and a NIDS) will actually resolve ambiguities [3]. The NIDS conducts periodic probing of the site it protects in order to construct a database that it then consults in the presence of ambiguous traffic in order to determine the correct interpretation to assume. A similar scheme avoids the need to conduct probing by instead using passive inference, such as deducing operating system type and then resolving ambiguities in the manner known *a priori* to be used by the given OS [6]. A limitation of active mapping is the need to maintain an up-to-date database in the presence of churn and end systems changing their network-layer identity, as well as a degree of incompleteness since some ambiguities do not provide externally visible manifestations of their resolution by end systems. A limitation of passive fingerprinting concerns the degree to which passively observable information such as OS type provides relatively coarse-grained information from which to draw inferences. (For example, different OS variants may

BRO 1.2	
14,591	ContentGap
7,546	AckAboveHole
2,249	window_recision
735	bad_TCP_checksum
460	SYN_with_data
311	possible_split_routing
290	data_before_established
98	bad_ICMP_checksum
85	above_hole_data_without_any_acks
35	connection_originator_SYN_ack
30	bad_TCP_header_len
18	inappropriate_FIN
15	SYN_seq_jump
15	premature_connection_reuse
9	active_connection_reuse
8	data_after_reset
3	SYN_inside_connection
3	SYN_after_reset
3	bad_SYN_ack
2	TCP_christmas
1	RetransmissionInconsistency
1	FIN_advanced_last_seq
1	bad_UDP_checksum
26,509	

SNORT 2.6	
161,862	(spp_stream4) possible EVASIVE FIN detection
36,873	(spp_stream4) possible EVASIVE RST detection
27,384	(spp_stream4) TCP CHECKSUM CHANGED ON RETRANSMISSION
1,933	(spp_stream4) Possible RETRANSMISSION detection
458	(spp_stream4) DATA ON SYN detection
67	(snort_decoder) WARNING: ICMP Original IP Header Truncated!
30	(snort_decoder) WARNING: TCP Data Offset is less than 5!
18	(snort_decoder): Truncated Tcp Options
12	(snort_decoder): Experimental Tcp Options found
2	(snort_decoder): Tcp Options found with bad lengths
1	(snort_decoder) WARNING: ICMP Original IP Fragmented and Offset Not 0!
228,640	

SNORT 2.8	
4,844	TCP Timestamp is outside of PAWS window
2,058	Data sent on stream not accepting data
807	Bad segment, adjusted size $<= 0$
461	Data on SYN packet
67	(snort_decoder) WARNING: ICMP Original IP Header Truncated!
30	(snort_decoder) WARNING: TCP Data Offset is less than 5!
18	(snort_decoder): Truncated Tcp Options
12	(snort_decoder): Experimental Tcp Options found
5	Data sent on stream after TCP Reset
2	(snort_decoder): Tcp Options found with bad lengths
1	(snort_decoder) WARNING: ICMP Original IP Fragmented and Offset Not 0!
8,305	

Table 2. Aggregate summaries of anomalies and evasion-related events reported by the NIDSs under test, on the 24h ICSI trace.

exhibit different ambiguity resolutions even though their passive fingerprints are indistinguishable.)

A third class of defense is to acquire information on ambiguity resolution directly from the end systems as the ambiguity arises. For example, Dreger et al. describe incorporating host-based context into the Bro system, for which (among other functionality) they demonstrate instrumenting an Apache web server to send to Bro the URLs that the server ultimately resolves (after all OS and server preprocessing has completed) [5].

Several tools have been developed for testing NIDS for vulnerabilities to evasion. Fragrouter[5] implements some network-layer evasions based on IP fragmentation. Unlike our framework, it modifies live traffic only. The libwhisker[6] library provides basic functionality for testing HTTP implementations. Nikto[7] leverages the library, adding HTTP content obfuscation techniques. Both tools primarily target live-traffic operation.

Regarding systematic evaluation of NIDS in the presence of possible evasions, Vigna and colleagues present a framework for NIDS testing based on traffic transformation [14]. Rather than testing the NIDSs' awareness of evasion, they emphasize evaluating the robustness of individual signatures used by such NIDSs. Their system takes as input an attack trace, to which it applies semantically invariant transformations and then and monitors for changes in the alerts generated by the NIDSs. Similarly, Rubin et al. developed a framework to facilitate traffic transformations on different network layers [15], again aiming to produce variants of a specific attack.

In contrast to these efforts, our framework does not assume the existence of an attack, but instead determines the general effects of traffic transformations. This allows us to separate the NIDS's specific attack detection logic from its architectural analysis limitations. In addition, the work of Rubin et al. develops a formal model of possible transformations, which allows them to exhaustively test a NIDS against attack variants. Our work, on the other hand, aims to facilitate a public, open-source effort for developing NIDS evasion test suites, with a related emphasis for our framework on modularity and a plug-in architecture.

[5] Per `http://www.securityfocus.com/tools/176`, nominally available at `http://www.anzen.com/research/nidsbench/`, but in fact that location no longer resolves.

[6] `http://www.wiretrip.net/rfp/libwhisker/`

[7] `http://www.cirt.net/code/nikto.shtml`

6 Discussion and Future Work

It is important to recognize that our `idsprobe` framework does *not* attempt to provide "turnkey" evaluation of NIDS evasion vulnerabilities. Rather, our aim is to provide the means for an experienced assessor to more readily construct good test cases, and more efficiently apply those test cases in a repeatable fashion across a set of NIDS under consideration. We emphasize that we also do not strive to ourselves provide a *comprehensive* set of evasion tests; rather, we aim to facilitate that others—in particular, the network security community as a whole—can *collectively* work towards such a goal. These considerations motivate our open-source, modular/plug-in approach.

The elements of our framework that focus on packet trace transformation are now fairly mature. The focus of our immediate future work is to further develop and refine the components of our framework that support live testing (i.e., evaluation based on dynamically generated, actual network traffic). As reported above, we have the basic building blocks for such testing in place. The next steps are then to devise methodologies for assessing evasions based on overloading NIDS resources (CPU and memory), which aim to induce a NIDS to either crash or perform diminished analysis no longer capable of detecting an attack that accommodates the stress traffic; and to assess the efficacy of on-line anti-evasion technology, such as traffic normalizers like `norm` [2], techniques that probe end-systems (e.g., *active mapping* [3]), and network-host hybrids, where hosts communicate information to the NIDS (e.g., [5]). For all of these, correct assessment of their effectiveness cannot use trace-based testing, but rather requires live testing.

7 Summary

We have designed and implemented the `idsprobe` framework to facilitate the creation of offline as well as live evasion test-cases in a pluggable fashion, coupled with fully automated testing of different NIDS on the resulting test-cases. We aim for the system to encourage extension and broad use by the community, and to this end will provide the software to others upon request, and ultimately aim to maintain it in as a public open-source resource.

8 Acknowledgments

This work was partially supported by the iCAST project sponsored by the National Science Council (NSC), Taiwan, under Grants 95-3114-P-001-002-Y02, 95-3114-P-307-003-Y, 96-3114-P-001-002-Y, 95-2221-E-017-007, 95-3113-P-017-001, as well as by the Ministry of Economic Affairs, Taiwan, under Grant 96-EC-17-A-31-F1-0824. The opinions expressed in this work are solely those of the authors and should not necessarily be considered to be the opinions of any government, funding agency, or other organization.

References

1. Ptacek, T., Newsham, T.: Insertion, evasion, and denial of service: Eluding network intrusion detection. Secure Networks, Inc., Jan (1998)
2. Handley, M., Paxson, V., Kreibich, C.: Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. Proc. USENIX Security Symposium (2001)
3. Shankar, U., Paxson, V.: Active mapping: resisting NIDS evasion without altering traffic. Proc. Symposium on Security and Privacy (2003) 44–61
4. Group, N.: Network IPS Testing Procedure (V4.0) (2006) `http://www.nss.co.uk/certification/ips/nss-nips-v40-testproc.pdf`.
5. Dreger, H., Kreibich, C., Paxson, V., Sommer, R.: Enhancing the accuracy of network-based intrusion detection with host-based context. In: Proc. Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA). (2005)
6. Taleck, G.: Ambiguity resolution via passive os fingerprinting. In: Proc. Conference on Recent Advances in Intrusion Detection (RAID). (2003) 192–206
7. Watson, D., Smart, M., Malan, G.R., Jahanian, F.: Protocol Scrubbing: Network Security through Transparent Flow Modification. IEEE/ACM Transactions on Networking **12**(2) (April 2004) 261–273
8. Pang, R., Paxson, V.: A High-Level Programming Environment for Packet Trace Anonymization and Transformation. In: Proceedings of the ACM SIGCOMM Conference. (August 2003)
9. Paxson, V.: Bro: A system for detecting network intruders in real-time. Computer Networks **31**(23-24) (1999) 2435–2463
10. Kreibich, C.: Design and Implementation of Netdude, a Framework for Packet Trace Manipulation. Proc. USENIX Technical Conference, FREENIX track (2004)
11. Biondi, P.: Scapy, a powerful interactive packet manipulation program `http://www.secdev.org/projects/scapy/`.
12. Provos, N.: A Virtual Honeypot Framework. Proceedings of the 13th USENIX Security Symposium (2004) 1–14
13. SourceFire: Snort, the Open Source Network Intrusion Detection System `http://www.snort.org/`.
14. Vigna, G., Robertson, W., Balzarotti, D.: Testing network-based intrusion detection signatures using mutant exploits. Proceedings of the 11th ACM Conference on Computer and Communications Security (2004) 21–30
15. Rubin, S., Jha, S., Miller, B.: Automatic Generation and Analysis of NIDS Attacks. Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)-Volume 00 (2004) 28–38