Packet Trace Manipulation Framework for Test Labs

Andy Rupp
Ruhr-Universität Bochum
arupp@crypto.ruhr-uni-bochum.de

Holger Dreger
TU München
dreger@in.tum.de

Anja Feldmann
TU München
anja@in.tum.de

Robin Sommer
TU München
sommer@in.tum.de

ABSTRACT

Evaluating network components such as network intrusion detection systems, firewalls, routers, or switches suffers from the lack of available network traffic traces that on the one hand are appropriate for a specific test environment but on the other hand have the same characteristics as actual traffic. Instead of just capturing traffic and replaying the trace, we identify a set of packet trace manipulation operations that enable us to generate a trace bottom-up: our trace primitives can be traces from different environments or artificially generated ones; our basic operations include merging of two traces, moving a flow across time, duplicating a flow, and stretching a flow's time-scale. After discussing the potential as well as the dangers of each operation with respect to analysis at different protocol layers, we present a framework within which these operations can be realized and show an example configuration for our prototype.

Categories and Subject Descriptors: C.2.0 [Computer-Communication Networks]: General.

General Terms: Measurement.

Keywords: Network, measurement, trace generation, evaluation, network intrusion detection.

1. INTRODUCTION

When evaluating network components such as network intrusion detection systems, firewalls, routers, or switches there are two basic approaches, each of which has advantages as well as drawbacks: while capturing and replaying live traffic provides us with realistic traffic, its characteristics are fixed. On the other hand when generating traces artificially (e.g., by using a network simulator), tuning individual parameters is possible, but the resulting trace often does not show the wide variability of actual traffic. Therefore, we propose an alternative approach: generating traffic *bottom-up* from traces gathered from different sources (captured or crafted), by combining and adapting them. For this purpose we introduce a set of basic trace manipulation operations which try to minimize the impact of these modifications to the anticipated application.

This work originated in the context of evaluating network intrusion detection systems (NIDSs). To assess a NIDS, one often injects attack-traffic into high-volume background traffic to measure false-negative and false-positive rates (see, e.g., [13]). The main difficulty with this approach is that it is non-trivial to vary parameters, such as the attack rate, without changing the overall characteristics of the trace, indicating that one needs various degrees of flexibility for inserting attack traces into background traffic. Yet we also note that it is possible to decompose the complex *insert attack trace* operation into several more basic operations. These include *merging* of two traces into a single stream (the attack and the background trace); *adapting* one trace to another (the address space of the attack traffic to the background traffic's one); *stretching or compressing* flows (e.g., to change the length of the attack); *moving* a flow in time (e.g., an attack flow to the beginning); and *duplicating* flows (e.g., an attack or a background flow). While the complex operation *insert* is difficult to realize, these primitives are much easier to define and their impacts are easier to analyze.

Motivated by this application scenario, in this paper we examine packet trace manipulation in detail. First, we identify a set of basic operations and examine, with respect to our application scenario, whether they hinder specific kinds of analysis at the various protocol layers (e.g., is a NIDS able to detect the trace manipulation?). Knowing the impacts of our base operations might enable us to adapt real traffic in a way that cannot be recognized by the system under test. We present a general framework for composing complex traffic manipulations from these primitives: *Plug-ins* realize basic operations as well as mechanisms to interconnect them. The parameterization of the plug-ins and the set-up of the interconnections are done using a custom language. This provides us with fine-grained control capabilities for manipulating packet traces as well as their traffic characteristics. We believe that the framework will prove its usefulness in test-lab contexts different from our original motivation. For example, to evaluate traffic measurement tools, such as alternatives to Cisco NetFlow [4], or to stress routers [8]. Performance evaluations within such application scenarios require reproducable and scalable work-loads or put differently many different traces. However it depends on the application which aspect of the trace can or cannot be manipulated. Accordingly the application determines which operations are appropriate and which ones are inappropriate.

We note that, in general, unrealistic packet constellations cannot be avoided when manipulating traffic traces. Hence, we do not claim that we offer a complete solution. Rather we argue that we can often minimize the impact of manipulations if we understand what characteristics are modified and how this affects our concrete application. For exam-

IMC'04, October 25–27, 2004, Taormina, Sicily, Italy.

ple, if we evaluate tools for round-trip-time estimation, we should better leave inter-packet times untouched. On the other hand, to our knowledge there is no NIDS that takes RTTs into account. Thus, when evaluating NIDSs it does not do any harm to "stretch" a connection by multiplying the inter-packets gaps by a constant factor [1].

There exist other tools to process and manipulate traffic traces in test-lab environments. Libpcap [17] provides an OS-independent interface for processing packet traces. We utilize it as an input plug-in. NetDuDE [12] is a visual packet editor. While it provides a rich set of operations, its main focus is interactive use. We are considering to leverage parts of its general-purpose manipulation library, though. Click [11] is a modular router whose architecture is similar in spirit to our framework. It focuses on forwarding packets rather than changing traffic characteristics. Tcpreplay [18] and TCPivo [9] play traces back into a live network and thus provide us with output back-ends. Alternatively, we could use a DAG network interface card [6] to replay packets with accurate timings. Network simulators like NS [2] are useful for generating traces but are not the appropriate tools for manipulating captured or artificially crafted network traffic. When used as a network emulator NS interacts with the live network but lacks the necessary packet manipulation operations.

Simply using replay tools such as TCPivo to replay the traces and a router or switch to intermix the packets lacks reproducibility. This could be overcome by using a network emulator or by capturing the merged trace. Yet the resulting trace would have several drawbacks: first, one has no chance of adopting the trace to the test-bed environment; second, the inter-packet spacing of flows crucial for the analysis may not be preserved; third, packets can get lost if the switch/router does not have sufficient buffering; fourth, depending on the input trace the output trace may loose its variability (since the new utilization is always 100%). In this case removing some of the flows before the merge might have been more appropriate. Overall one should note that replay tools can be used to realize one basic operation. Tools such as libpcap filters allow us to realize part of another basic operation. Yet they do not provide us with the full flexibility of the framework proposed in this paper.

In Section 2, to motivate the work, we briefly discuss two application scenarios. In Section 3 we identify our basic trace manipulation operations and examine both their potential and their dangers. Section 4 presents our general framework and Section 5 demonstrates how to compose a complex trace using these operations. Finally, Section 6 briefly summarizes and discusses future work.

2. APPLICATION SCENARIO

To motivate our approach to manipulation of traffic traces, we discuss two example applications: evaluation of network intrusion detections systems and traffic measurement tools.

2.1 Network Intrusion Detection

A common approach to evaluating NIDSs uses captured packet traces to create realistic network work-loads. While this provides us with a reproducible setting, it is hard to record a "real-world" trace containing the exact characteristics one wants to test for. Therefore one usually modifies the captured trace to vary some of its parameters. For example, a straight-forward yet hard to realize operation is to insert separately recorded attack traffic into a trace that provides representative background traffic for the environment under test. Unfortunately, inserting one trace into another is much more complex than simply mixing packets.

Initially, the attack has to be adapted in order to fit seamlessly into the background traffic. While this ensures that the trace looks like being appropriate for the chosen environment (e.g., the changed IP addresses from the attack trace match those in the background trace) it is nearly impossible to identify all interactions caused by inserting the attack traffic into the background traffic. For example, if the injected attack is a successful attack on an FTP server this can imply that the FTP server is no longer available after the end of the attack. In this case, to preserve realism, one would have to remove all connections from the background trace that involve the FTP server after the attack took place. Such kinds of adaptations are especially crucial if one considers an anomaly-based system. Here the NIDS might detect some artifact caused by the trace insertions instead of actually recognizing the attack, implying that its performance looks much more impressive in the test-lab than in the real world.

Fortunately, not all trace interactions influence the test results. Yet, it is not obvious which artifact affects a NIDS's detection process or some other network traffic analysis. Therefore it is hard to judge whether a specific trace, generated from other traces, poses an acceptable loss of realism or not.

We note that modifications to traces can lead to artifacts at almost all layers of the protocol stack: at the link layer, some ARP requests may be missing or obsolete; at the network layer, IP-fragments may be missing or reordered; at the transport layer, an end-point's round-trip-time estimation may no longer be valid; at the application layer, it is possible to violate inter-connection semantics. The greater the complexity of the manipulations to the traffic the larger the difficulty of understanding their impact. Therefore, we propose to decompose the trace manipulations into basic operations, and analyze their potentials and dangers.

2.2 Traffic Measurement

When developing traffic measurement models and tools, evaluating their performance systematically is rather difficult. For example, while several new flow models have been proposed recently (see, e.g., [7, 15]), to assess their feasibility we need very fine-grained control over the characteristics of a packet stream. In particular we need the ability to tune the utilization of the link implied by such a stream. Unfortunately, given a packet trace it is non-trivial to predictably modify its link utilization without creating unrealistic packet constellations. A naive approach to increase the utilization is to merge-in a second trace packet-wise. As Figure 2.2 demonstrates, this very likely leads to short-term spikes that exceed a link's capacity. Figure 2.2 shows the bandwidth usage of two merged one-minute traces with an average bandwidth of 39 and 52 Mbit/s. While the average stays below the link's capacity of 100 Mbit/s there are plenty of peaks that cannot occur on a real 100Mbit/s link.

A more fine-grained mechanism to increase a link's utilization is the duplication of individual flows. When we operate on a single flow at a time, we can often avoid unrealistic

[1] By using large factors we may exceed certain time-outs of the NIDS but those are usually several orders of magnitude larger than a connection's RTT.

artifacts. If the duplication of a flow exceeds the bandwidth of the link, most of the time it is due to spikes amplifying each other. Then it may already help to either move the new flow slightly in time or to stretch its time-scale by increasing inter-packet gaps. For a flow-level model these operations are not observable as an unrealistic constellation (of course if we duplicate a flow we have to assign new IP addresses to the copy).

3. TRACE MANIPULATION

In this section we first identify a set of basic operations and then analyze their impact with regards to different layers of the protocol-stack. That is, assuming a system that performs per-protocol analysis, we try to identify whether it may perceive the modifications. For example, most NIDSs extract IP addresses on the network-layer, reassemble TCP streams on the transport-layer, and watch out for suspicious HTTP requests on the application-layer.

3.1 Basic operations

We start by defining basic operations for trace manipulation that can be applied either to a single packet or to the set of packets corresponding to some flow. A flow corresponds to a sequence of packets involving the same two end-points where the time difference between in-sequence packets is smaller than some timeout. An end-point consists of its IP address, the encapsulated protocol, and for TCP and UDP the port number.

Adapt: As highlighted by the example in Section 2 the first step of many trace manipulations is the adaptation of the packet-level headers to the particular environment. By changing the protocol header fields, such as MAC addresses (Ethernet link-layer) and IP addresses (network-layer), together with appropriate adjustment of the checksums, the packets become consistent with the test environment. Depending on the local configuration other protocol fields such as the time-to-live also have to be adapted in order to enable (or disable) packets to reach their "new" destination. In addition for some test environments it might be necessary to change the TCP or UDP port numbers.

Merge: If more than one recorded trace is involved a second basic operation is needed: merging. To preserve inter-packet times of interdependent traffic they should be merged according to their packet timestamps (i.e., in chronological order). Since the timestamps in the traces rarely overlap and a concatenation of traces usually is not sufficient, we propose to use relative timestamps for all packets and all traces. This normalizes the timestamp of first packet of each trace to zero. After choosing one of the traces as a base trace the remaining traces are injected at specified points. One problem inevitably arises from this method: how to maintain the nominal packet rate on the link (i.e., how to adhere to the minimal time-distances between adjacent packets). Otherwise the network link becomes overloaded. The following operations are helpful in addressing this issue.

Scale: One possibility to resolve time-distance conflicts is to compress or to stretch the traces. For compression, all inter-packet gaps of a trace (or of a flow) are multiplied by a constant factor smaller than one. Accordingly, stretching corresponds to multiplying the inter-packet gaps by a factor greater than one. Although this operation is inappropriate when applied to a whole trace [10], it is suitable for small adjustments as well as adjustments of individual flows.

Remove/Duplicate: In order to maintain the crucial minimal inter-packet gaps it may be sensible to remove one or more flows from a trace. On the other hand one could also increase the bandwidth of the resulting trace by duplicating flows.

Move: The move operation displaces single packets or flows. Besides being a useful standalone operation it provides another way of circumventing unrealistically small inter-packet gaps.

3.2 Problems with the basic operations

Each one of the basic operations introduces a number of *artifacts*. An artifact is a sequence of packets which is, by protocol or statistical analysis, distinguishable from the behavior of an actual interacting system. In the following we identify a number of artifacts and discuss them in the context of the network-layer within which their effects are perceivable. Note that this is not a complete list. Rather it serves the framework-user as a starting point for identifying relevant artifacts for his application. Most artifacts are caused either by disturbances to a state management at a specific protocol-layer or by rearrangement of dependent packets.

Network-layer: One main type of network-layer analysis are statistical summaries, e.g., number of packets per second and number of bytes per second. All trace operations, except adapt, change these characteristics. The extent of the change depends heavily on the parameterization and some bounds apply: for example, by stretching or compressing a trace the metric "packets per second" is influenced at most linearly by the multiplication factor, or by moving only 1% of the traffic the mean of the distribution should hardly change. Yet the extremes, e.g., 5th and 95th percentile might change drastically.

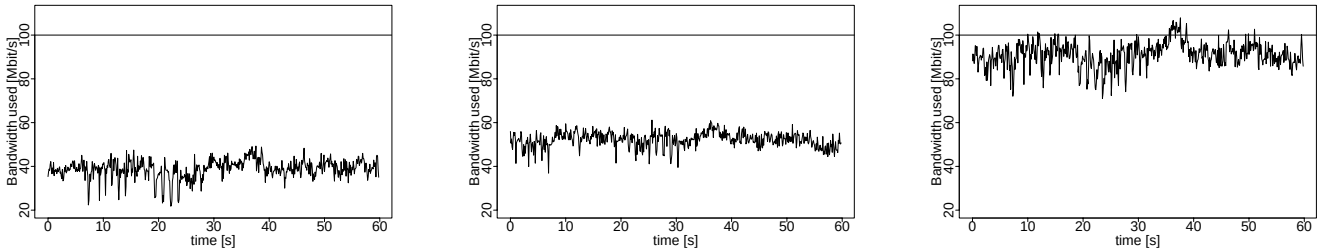
In an Ethernet-environment, our base operations can disturb the causal relationship between ARP and IP packets. Since the effects imposed by ARP are local in scope and usually not part of the protocol analyses, the current operations ignore them.

IP only needs to maintain state for IP fragment reassembly. Accordingly all protocol-based analysis at the network-layer should not be affected by the adapt, merge, and scale operations independent of whether they are applied to single packets or flows. The remove, duplicate and move operations also does not bias any protocol analysis when applied to flows. Yet, if applied to IP fragments, fragment reconstruction may no longer be possible and therefore the network-layer analysis suffers.

Transport-layer: One has to differentiate between UDP and TCP in order to decide which operations cause artifacts that are perceivable via transport-layer protocol analysis.

UDP is a stateless protocol and, hence, similar to IP. Therefore the same considerations that applied to IP (apart from IP fragmentation) apply to UDP. This implies that all of our basic operations do not affect UDP protocol analysis independent of whether they are applied to individual packets or to flows.

TCP, on the other hand, is state-full and offers a fault-tolerant, in-order, byte stream service. If applied to individual TCP packets, all of the operations are detectable if a real endpoint would have responded differently to the new packet sequence than it did in the trace. When applying the operations to individual flows this does not have to be the



(a) Trace 1 (b) Trace 2 Merged trace

Figure 1: Bandwidth usage (average bandwidth: trace 1: 39 MBit/s; trace 2: 52 MBit/s)

case. Let us start by assuming an ideal flow model: each flow corresponds to exactly one TCP connection. In this case one might presume it safe to apply the adapt, merge, remove, duplicate, and move operations to flows, since either all packets or no packet of the TCP connection is altered. Therefore the protocol analysis should yield comparable results. However, in a congested network, especially long-lived TCP connections adhere to congestion control and therefore compete for their fair share of the available bandwidth. When the operations disturb such an interaction it is infeasible to recreate or repair this interaction by the per-flow operations.

In contrast to the other operations, scaling changes the inter-packet spacing and therefore influences the analysis within a single TCP connection. For example, the initial retransmission timeout is computed by the end-system based on fixed OS-dependent values. By choosing a large stretch factor it is possible to excessively enlarge the initial inter-packet gap to the extent that the actual end-system would have experienced a timeout and therefore had retransmitted the packet.

In practice, due to resource limitations it is infeasible to use the above notion of an ideal flow. For example, in a trace we often do not see a complete 4-way-teardown of a connection (e.g., because we missed some packets while capturing; or one of the hosts has been turned off). Thus, although the TCP protocol allows arbitrary long idle times, in an online system at some point one has to terminate each connection. Otherwise, the state keeping can easily exhaust a system's resources [5]. Accordingly, we rely on a time-out-based definition of a flow: "data exchanged between two endpoints within some time". This causes further artifacts since it is now possible to apply operations on partial TCP-connections.

Application-layer: At the application-layer, the impact of each trace operation depends on the specific application and the transport protocol (i.e., TCP or UDP) that it uses. Since UDP-based application protocols often implement TCP-like error-recovery mechanisms which are perturbed by all basic per-packet operations they may perceive the output of these when applied to single packets as artifacts.

The merge operation, when applied to flows, is likely to cause the fewest artifacts at the application-layer: when the flows are disjoint in terms of address space the merge operation does not trigger artifacts for most application protocols. All other operations disturb the relationships between flows at the application-layer: consider a protocol such as FTP which uses a control connection and one or more separate

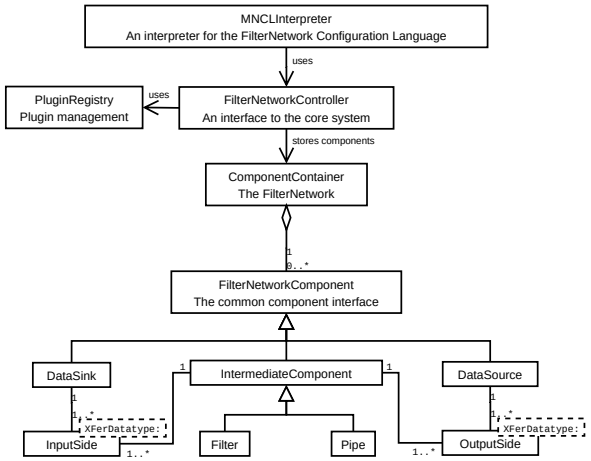


Figure 2: Advanced Pipes and Filters Architecture

data connections. As long as one adapts, scales, removes, duplicates or moves only one of these connections, one can easily cause artifacts in terms of missing expected connections and occurrence of unexpected connections (adaptation, merge, remove, duplicate); or wrong relative order of connections (scale, move). A different kind of artifact can be introduced by the scale operation: if one scales a whole trace, one does not only scale network delays but also computational and user-controlled delays.

4. SOFTWARE ARCHITECTURE

Section 3 identifies a set of basic trace manipulation operations that can be assembled to form arbitrary complex operations. In this section we describe a software system that provides us with a framework to realize such trace manipulation. Based upon a custom configuration language to conveniently specify a set-up it provides us with the ability to realize the basic operations and build more complex operations bottom-up using already specified ones as primitives.

Basic operations are implemented as small independent data-processing components, i.e., *filter plug-ins*, to facilitate reusability and enable step-wise development of complex manipulations. To enable the user to easily extend the system to read and write the trace data in arbitrary formats our design includes *input plug-ins* and *output plug-ins*. For example the user might write a input-plugin for reading trace data out of a specific database and a output plugin for releasing trace data onto a specific network.

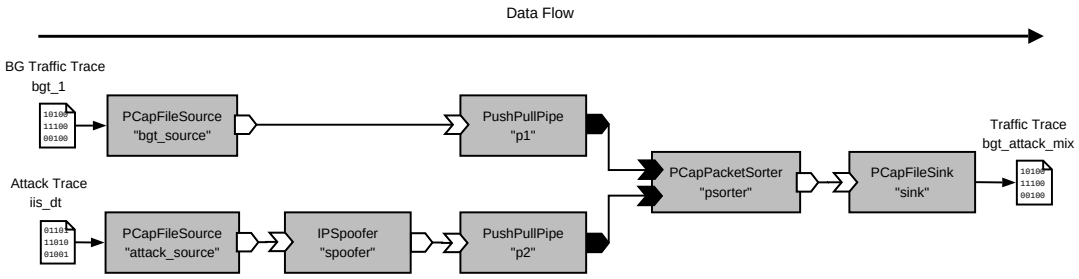


Figure 3: Example Application: FilterNetwork

The core of our software, which enables the complex operations, is an *Advanced Pipes and Filters Architecture* that is able to manage and control filter systems processing data of *arbitrary* types. Its design is mainly based on the *Pipes and Filters* and the *Pluggable Component* architecture pattern presented in [3] and [19]. The main parts of the resulting architecture are sketched in Figure 2.

The framework distinguishes between different types of data processing components:

- *DataSources* provide a filter system with input data.
- *DataSinks* save data modified by a filter system.
- *IntermediateComponents* like *Filters* realize some data manipulating operation(s) and *Pipes* can be used as optional data buffers between two components.

In our design we define a generic interface that all these components have to implement. A component may have an arbitrary number of input-side and output-side "ports" to exchange data and control-messages with other components. The architecture provides plug-in developers with two types of data transfer-mechanisms, i.e., push and pull. If a component uses the pull mechanism on one of its input ports the input data must be requested explicitly from the preceding component. In the other case data arrives without demand.

Each component can be developed independently from the core system. These are integrated by a *plugin registry* at runtime. The plug-in registry is responsible for the management of available plug-ins. From the connection of several component instances, a graph or network-like structure evolves which we call the *FilterNetwork* (aka filter system). The controller constitutes the most important part of the architecture. It provides all necessary methods to build and control the *FilterNetwork*. Thus, it becomes an interface (or facade) to the core functionality of the architecture.

In order to hand-over the full flexibility of the plug-in based design of the core to an end-user, the user interface of the architecture is a scripting language which allows to configure and interconnect the filter plug-ins as well as the input/output plug-ins. The result is user-composed filter systems that perform the desired manipulations on the chosen input data.

The second part of our software system is a set of plug-ins that realize some of the basic operations we defined earlier. This package currently consists of

- the *PCapFileSource* plug-in whose instances can supply a FilterNetwork with network packets from libpcap-compatible files, adjust packet timestamps[2] and map packets to (transport-layer) flows on the basis of user-supplied flow specifications.
- the *PCapFileSink* plug-in whose instances write packets that have been manipulated by the *FilterNetwork* back to a libpcap-compatible file.
- the *IPSpoofer* plug-in whose instances can adapt the addresses and ports of packet flows according to a set of user-supplied spoofing rules that have a BPF-like [14] syntax.
- the *PCapPacketSorter* plug-in whose instances merge packets from multiple *InputSides* chronologically.
- the *PushPullPipe* plug-in whose instances simply act as transfer-mechanism converters between the pull interfaces of *PCapPacketSorter* instances and the push interfaces of *PCapFileSource*, *IPSpoofer* or other *PCapPacketSorter* instances.

[2]Required by the merge operation.

5. EXAMPLE APPLICATION

In this section we discuss how our software system can be used to insert a trace into another originating from a different environment to test a NIDS. Assume that the trace `iis_dt` was recorded within a test-bed environment while executing an attack script that tries to exploit a vulnerability in a Web server software. The trace contains network packets exchanged between the attacking system `192.168.0.2` and the victim host `192.168.0.1`. Let us further assume that the trace `bgt_1` contains background traffic from the network `134.96.223.0/24` that is appropriate for the environment in which the NIDS is to be assessed. For example, this trace may consist of regular traffic from and to the Web server `134.96.223.242`.

To insert the attack into the background traffic, we

- adapt the attack to the environment: we pretend that the local Web server `134.96.223.242` is the victim of an attack originating from some randomly chosen host of the network `131.152.123.0/24`.
- merge the traces while preserving the inter-packet gaps of interdependent traffic.

For simplicity we assume that the attack has not been successful and thus no further traffic interactions need to be simulated.

Figure 3 shows a *FilterNetwork* that applies the necessary operations. It is configured based on the configuration shown in Figure 4. `Bgt_source` and `attack_source` provide the *FilterNetwork* with packets and flow information.

```
Components {
  # Background Traffic Source
  Source bgt_source {
    type PCapFileSource;
    Config { pcap_src_list = "bgt_1"; }
  }
  # Attack Traffic Source
  Source attack_source {
    type PCapFileSource;
      Config {
        pcap_src_list = "iis_dt [2.0, 5.0]";
      }
  }
  IComponent spoofer {
    type IPSpoofer;
    Config {
      spoofing_rules =
      "dst port 80 -> src net 131.152.123.0/24
                      dst host 134.96.223.242";
    }
  }
  IComponent psorter { type PCapPacketSorter; }
  Sink sink {
    type PCapFileSink;
    Config { pcap_dst_file = "bgt_attack_mix"; }
  }
  # Pipe between bgt_source and psorter
  IComponent p1 { type PushPullPipe; }
  # Pipe between spoofer and psorter
  IComponent p2 { type PushPullPipe; }
}

Connections {
  bgt_source[0]    -> p1[0];
  attack_source[0] -> spoofer[0];
  spoofer[0]       -> p2[0];
  p1[0]            -> psorter[0];
  p2[0]            -> psorter[1];
  psorter[0]       -> sink[0];
}
```

Figure 4: Example Application: Configuration

The attack starts 2–5 seconds after the first packet of the background traffic trace. The *IPSpoofer* instance is directly connected to `attack_source`. It rewrites the IP addresses of packet flows belonging to the malicious HTTP session. *PushPullPipe* instances (`p1` and `p2`) are interposed between the components `psorter`/`bgt_source` and `psorter`/`spoofer`. These simply serve as transfer-mechanism converter (push to pull). The *PCapFileSink* instance `sink` completes the *FilterNetwork* and stores manipulated packets in the libpcap-compatible file `bgt_attack_mix`.

6. SUMMARY

Motivated by the task of evaluating Network Intrusion Detection Systems, we identify a set of trace manipulating operations that aid in constructing packet traces for test-labs. We discuss whether and under which circumstances such operations generate irregularities that would not be present in captured real-world traffic. Furthermore we explore what kind of analysis at what protocol-layer is biased by such artifacts and might therefore draw conclusions based on the artifact rather than the trace content.

We propose a flexible and expandable software system for trace manipulations. The core of this software system is an abstract architecture that is able to manage and control filter systems. The current set of plug-ins implements some basic operations for trace manipulation. By means of a convenient configuration language users can specify which instances of these components should be interconnected in what fashion to build complex filter systems. First experience and initial experiments demonstrate that the architecture with the currently implemented set of components is a valuable tool for flexible, fine-grained trace manipulations. For a comprehensive description of the software (available at [1]) and a more detailed discussion of architecture as well as the artifacts we refer to [16].

7. REFERENCES

[1] Advanced pipes and filters architecture & TTM plugin package project. `http://www.net.informatik.tu-muenchen.de/~rdc/`.

[2] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, S. Shenker, K. Varadhan, H. Yu, Y. Xu, and D. Zappala. Virtual InterNetwork Testbed: Status and research agenda. Technical Report 98-678, University of Southern California, July 1998.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons Ltd, Chichester, England, 1996.

[4] Cisco Netflow. `http://www.cisco.com/warp/public/732/Tech/nmp/netflow/index.shtml`.

[5] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *Proc. 11th ACM Conference on Computer and Communications Security*, 2004.

[6] ENDACE measurement systems. `http://www.endace.com/`.

[7] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a Better NetFlow. In *Proc. of the ACM SIGCOMM*, 2004.

[8] A. Feldmann, H. Kong, O. Maennel, and A. Tudor. Measuring BGP pass-through times. In *Proc. of the Passive and Active Measurement Workshop (PAM)*, 2004.

[9] W. Feng, A. Goel, A. Bezzaz, W. Feng, and J. Walpole. TCPivo: A High-Performance Packet Replay Engine. In *Proc. of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*, 2003.

[10] P. Kamath, K. Lan, J. Heidemann, J. Bannister, and J. Touch. Generation of High Bandwidth Network Traffic Traces. In *Proc. International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2002.

[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 3(18), 2000.

[12] C. Kreibich. Design and Implementation of Netdude, a Framework for Packet Trace Manipulation. In *Proc. Usenix Technical Conference, Freenix Track*, 2004.

[13] R. Lippmann, R. K. Cunningham, D. J. Fried, I. Graf, K. R. Kendall, S. E. Webster, and M. A. Zissman. Results of the 1998 DARPA Offline Intrusion Detection Evaluation. In *Proc. Recent Advances in Intrusion Detection*, 1999.

[14] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In U. Association, editor, *Proc. Winter 1993 USENIX Conference*. USENIX Association, 1993.

[15] P. Phaal, S. Panchen, and N. McKee. sFlow, 2001. RFC 3176.

[16] A. Rupp. A Software System for Packet Trace Customization with Application to NIDS Evaluation. Master's thesis, Universität des Saarlandes, Germany, 2004.

[17] The `tcpdump/libpcap` project. `http://www.tcpdump.org/`.

[18] The `tcpreplay` project. `http://tcpreplay.sourceforge.net/`.

[19] M. Völter. PluggableComponent - A Pattern for Interactive System Configuration. In *Proc. of the 4th European Conference on Pattern Languages of Programming and Computing*, 1999.