FLAME: A Flow-Level Anomaly Modeling Engine

Daniela Brauckhoff
ETH Zurich, Switzerland
brauckhoff@tik.ee.ethz.ch

Arno Wagner
ETH Zurich, Switzerland
arno@wagner.name

Martin May
ETH Zurich, Switzerland
may@tik.ee.ethz.ch

Abstract

There are several remaining open questions in the area of flow-based anomaly detection, e.g., how to do meaningful evaluations of anomaly detection mechanisms; how to get conclusive information about the origin and nature of an anomaly; or how to detect low intensity attacks. In order to answer these questions, network traffic traces that are representative for a specific test environment, and that contain anomalies with selected characteristics are a prerequisite. In this work, we present flame, a tool for injection of hand-crafted anomalies into a given background traffic trace. This tool combines the controllability offered by simulation with the realism provided by captured traffic traces. We present the design and prototype implementation of flame, and show how it is applied to inject three example anomalies into a given flow trace. We believe that flame can contribute significantly to the development and evaluation of advanced anomaly detection mechanisms.

1 Introduction

In the last years numerous approaches for automated detection of network anomalies in flow data, such as DoS attacks, outages and scans have been proposed [12, 4, 3, 11]. In contrast to the vast number of anomaly detection systems researched, the effectiveness of these systems has not been well investigated. A common practice is to evaluate an anomaly detection system with one of the few publicly available network traffic traces, which are usually anonymized and sampled. A second common approach is to use private traces from campus networks or small ISPs. Both methods have significant disadvantages: Anonymization and sampling alter the traffic characteristics and thereby the evaluation results [1]. Manually labeled traces introduce bias and errors. Finally, the set of normal and anomalous traffic characteristics present in a short trace is limited. An alternative approach to using real traffic traces is simulation, which allows for perfect control over the environment. However, simulation struggles with its own problems. Realistic simulation of baseline traffic, i.e. traffic without anomalies, is largely an unsolved problem today. Available simulation traces are typically of low quality. For example the MIT's Lincoln Lab [6] traces, which are widely used as test data for anomaly detectors, have been shown to contain serious artifacts [5].

In this paper we present flame, a framework for injecting user-defined anomalous traffic into existing flow traces[1]. In flame, we combine simulation and real traces to get the best of the two worlds. We use real traces as background traffic and provide the means for simulating, and injecting realistic anomalous traffic into these. Thus, our approach inherits the control over anomalous traffic from the simulation-side, while at the same time the realism of background traffic is given from the trace-side. Moreover, with our approach anomalies are injected into the raw flow data, and not into some specific metric that is computed on it. Consequently, our approach is not targeted to a specific anomaly detector, but applicable to all detectors relying on flow data.

We are aware that our approach does not solve the problem of establishing ground truth, i.e., of identifying anomalies already present in the trace. Moreover, avoidance of injection artifacts has to be considered by the anomaly designer. Nevertheless, we believe that flame can play a significant role in the systematic evaluation of anomaly detection systems. It also represents a valuable tool for experimentation, especially, for tackling the remaining open issues in anomaly detection, i.e., localization and classification of anomalies, and detection of low intensity attacks, since it allows for challenging detection systems with hand-crafted anomalies. Note that we do not evaluate any anomaly models in this paper. The focus of the work presented herein are the injection

[1] With flow we refer to the common 5-tuple aggregation of packets according to IP addresses, ports, and protocol.

methodology and solution architecture.

In Section 2, we introduce our approach for modeling flow-level anomalies. We classify anomalies into three classes, additive, subtractive, and interactive, and for each class, we specify the actions that are required for anomaly injection. We present the design and implementation of our anomaly injection framework prototype in Section 3. In Section 4, we illustrate the applicability of flame by describing three use cases for the injection of an additive (network scan), subtractive (ingress shift), and an interactive (DoS) anomaly. Related work is presented in Section 5. Finally, we conclude our work in Section 6.

2 Modeling Network Traffic Anomalies

This section presents our methodology for modeling network traffic anomalies. We define three classes of anomaly models and show the modifications to an existing trace that are required for each anomaly class. We also outline how anomaly models are built. The goal of this work is to provide a framework for injecting realistic and parameterizable anomalies into existing flow traces. Unfortunately, an anomaly is an ill-defined concept: Usually the term is used to describe the effect of some unusual event on monitored network statistics (e.g., some spike or drop in the byte counts). This effect could be captured with a *descriptive* model. We use, however, a different approach that concentrates on the cause for an anomaly (e.g., an ongoing denial of service attack, or a flash crowd). Hence, we aim at building *constructive* anomaly models for a list of known events. The effect on monitored traffic statistics (e.g., the size of a peak) is only of secondary interest to us since it depends to a large extent on the existing background traffic, and is in that sense not generic. Another advantage of constructive modeling is that it is agnostic with respect to the traffic metrics used by the anomaly detector.

2.1 Classes of Traffic Anomalies

Each unusual event has a different effect on the observed network traffic. An outage event, for example, causes a loss of flows with specific characteristics, while a network scan generates additional traffic. Events of larger scale, such as Denial of Service attacks, might even impact the background traffic due to congestion, network element overload, or changes in user behavior caused by the anomaly. Accordingly, we distinguish three different classes of anomalies: Additive anomalies add flows, but do not interact with the baseline traffic. Examples for this type are alpha flows, network scans, or bot activity. Subtractive anomalies are represented by removal of specific flows from the baseline traffic. Examples for this class are outage events, and ingress shifts to other AS peerings. Interactive anomalies add flows that have an impact on the baseline traffic. An example for this type are denial of service attacks. Clearly, the transition between additive and interactive anomalies is blurred. Each event (e.g., a scan) will have side-effects on existing traffic, once it becomes large enough to consume all available resources such as bandwidth or processing power. If this is the case, side-effects should be included in the model describing the anomaly. Naturally, the size at which the transition from additive to interactive occurs depends on the existing background traffic and the underlying network.

2.2 Required Injection Operations

In the following we will outline the trace modifications that are necessary to inject each of the three identified classes of anomalies. Injection of *additive* anomalies requires the generation of additional flows, followed by merging with existing baseline flows. This requires a model describing the traffic characteristics of the anomalous flows for each particular anomaly. Additionally, it is important that the generated traffic matches the characteristics of the baseline used. Otherwise, detection systems may trigger on the mismatches in traffic characteristics (e.g., unusual IP addresses), which may cause unrealistic detection results. Injection of *subtractive* anomalies requires the removal of flows with defined characteristics from an existing flow trace. This requires a characterization of the flows to be removed, as well as an algorithm for the removal process (e.g., deterministic or probabilistic selection). Finally, injection of *interactive* anomalies requires first the identification and removal of flows that are changed due to the injected anomaly, then the generation of anomalous flows and the re-generation of the flows to be modified with different characteristics (e.g., at a slower rate if the anomaly causes congestion). These side-effects need of course to be included in the anomaly model. In a last step, the generated anomalous and normal traffic gets merged with the ”cleaned” existing background traffic.

2.3 Building Anomaly Models

We have identified two basic operations that the framework needs to provide for anomaly modeling: *flow generation and flow deletion*. Both operations are specific to the type of anomaly that is to be injected. Hence, for each type of anomaly we want to inject, a deleter model and/or a generator model need to be defined. Defining such anomaly models requires expert knowledge. A deep understanding of anomalies and close observation of real traffic traces are essential prerequisites for building accu-

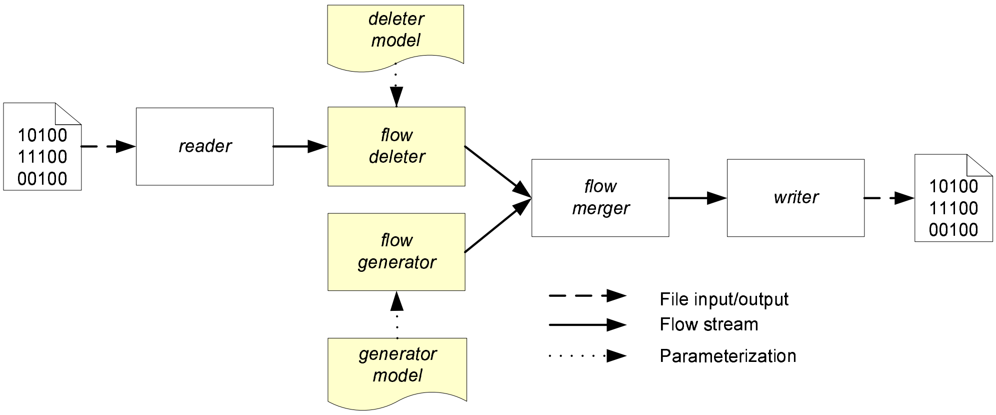

Figure 1: Basic design of the anomaly injection framework

rate and realistic anomaly models. Fortunately, we have access to a comprehensive NetFlow archive, which provides a large reservoir of anomalies to study. This flow archive, started in the DDoSVax project [2], records the network traffic on flow level from all border routers of a medium-sized ISP. We are continuously capturing this unsampled flow data since 2003. The framework will provide a basic set of anomaly models, but will also allow other users to define and share their own anomaly models. In section 4, we will present anomaly definitions for three examples, an ingress shift, a network scan, and a denial of service attack. The models we present target statistical anomaly detection systems with a temporal accuracy of minutes to hours (e.g., [3, 11]), and a spatial accuracy of single hosts (e.g., [12]) to Autonomous Systems (e.g., [4]). Consequently, the models we built will be accurate with respect to the granularity provided by the targeted anomaly detection system.

3 FLAME Prototype

In this section, we present the design and implementation of our flame prototype. We also evaluate the performance of the prototype.

3.1 Design

The basic design of our framework is presented in Fig. 1. It identifies the main functional blocks, the in- and outputs of the framework and shows the data flow through the system. The depicted configuration illustrates the case where all functional blocks are used, i.e., the injection of an interactive anomaly. In case of an additive anomaly the deleter component is not required, whereas injection of a subtractive anomaly requires only a reader/deleter/writer chain. The main functional blocks of the framework are:

- **Reader:** The reader takes a flow trace file as input, and converts it to an internal flow format.

- **Flow Deleter:** The deleter has two inputs, the deleter model that defines the flows to be deleted, and the stream of input flows. It outputs the remaining flows in an internal format. A detailed description of the flow deleter is presented in the next section.

- **Flow Generator:** The flow generator takes a generator model as input and outputs a stream of flows in an internal format. A detailed description of our flow generator is given in the next section.

- **Flow Merger:** The flow merger has two inputs, one stream of flows from the flow deleter and one from the flow generator. It outputs the merged stream of flows from both inputs. Merging is done based on flow packet timestamps.

- **Writer:** The writer takes flows in an internal format and writes them to a flow trace file.

While the flow generator and flow deleter are parameterized by the respective anomaly models, the writer and reader need to be selected according to the format of the input and output trace files.

3.2 Implementation

The individual data processing modules from Fig. 1 are implemented as separate processes. Communication is done via named pipes (Unix domain sockets). The data format used is always a stream of flows in an internal format inspired by Cisco NetFlow version 5. This allows removal and addition of individual components, for example the deleter module, with minimal changes. Basically only the set-up procedure, which is responsible for passing the individual parameterizations and the respective socket names to the components on start-up, has to be adjusted. Synchronization between components is done in a producer-consumer fashion, with buffering by the named pipes. This allows the use of multiple CPUs concurrently without the need for complicated synchronization mechanisms. One extension to the framework is to allow all internal connections between the components to use TCP sockets as alternative to Unix domain sockets. This facilitates distribution to a set of networked computers. Especially with complicated traffic generators, this possibility may prove valuable. The extension can also be propagated to reader and writer components.

Deleter Module

The task of the deleter module is to select which flows are passed onwards and which ones are dropped. Its internal structure is shown in Fig. 2. Flows are grouped into packets of flows (as specified by the NetFlow format and the IPFIX standard). The deleter module employs a *Flow Extractor* component to extract each individual flow and passes its fields to a decision function implemented in Python. The decision function has a logical value as result that directs the Flow Extractor to keep or delete each individual flow. In case of deletion, the number of flows in the packet (sequence number) is adjusted accordingly. If all flows from a flow packet are deleted, then it is dropped.

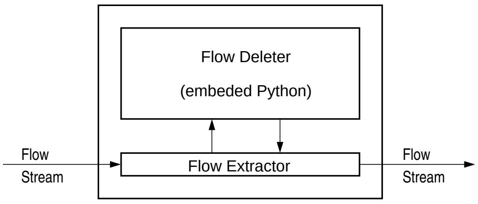

Figure 2: Deleter Module Structure

The choice of embedded Python for the actual decision procedure allows easy configuration and reconfiguration of the deleter functionality without recompilation. It also allows access to a wealth of Python modules that implement advanced statistical distributions and other useful helper functions for the decision process The downside is a negative impact on performance. For applications where performance is critical, the deleter module provides a well defined interface that allows the use of a deleter decision function implemented natively in C that replaces the Python script. However, use of this feature requires modification of the deleter source code and recompilation.

Traffic Generator

The task of the traffic generator module is to synthesize flow streams representing one or several anomalies. Its internal structure is given in Fig. 3. Anomalies can be generated on flow level or on packet level. If packet level generation is used, the packets have to be aggregated into flows. The structure of the packed-based anomaly generator substructure can be found in Fig. 4. The separation of the packet generation and flow aggregation step allows parametrization of the flow aggregation process (e.g., to set the timeouts for short-lived and long-lived flows used by Cisco routers). This facilitates adjusting the generated flow stream to the base data flow stream to minimize injection artifacts. Still, in some cases direct generation of flow-level anomalies might be accurate enough.

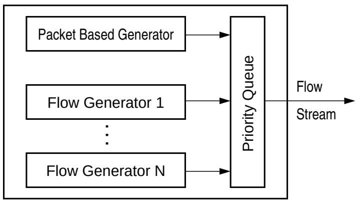

Figure 3: Complete Traffic Generator Structure

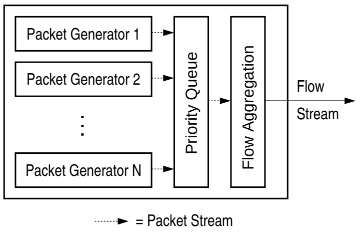

Figure 4: Packet-Based Traffic Generator Substructure

Each individual *Packet Generator* and *Flow Generator* is modeled as a system process. This again allows for coarse grained parallelism without complicated synchronization constructs. We use two priority queues that operate on timestamps, one for mixing the individual packet streams 1 to N, and one for mixing the flow streams 1 to N with the output of the packet-based generator. The advantage of this design is that the generator outputs are synchronized in a producer-consumer fashion, which facilitates their internal design significantly. Basically, packets/flows can be generated on-demand, or all packets/flows can be generated initially and stored in a large output buffer until consumption. It is also possible to read the packet or flow stream for a specific generator from file. This approach has advantages when a packet or flow stream is used repeatedly. Individual traffic generators can be hard-coded in C or use embedded Python for easy configuration, similar to the usage in the Deleter component.

3.3 Performance

In practical experiments with simple deleter functionality, for example a prefix-match, we have measured data processing rates of around 5'000 flow packets per second. This corresponds to roughly 140'000 individual flows per second, or around 500 million flows per hour[2]. The observed performance is high enough to support very large captured datasets as input data. Speed measurements were done on a 4-way SMP Linux system, featuring two dual core AMD Opteron 275 CPUs, running at 2200 MHz and 8GB of RAM.

4 Use Cases

We now present three use cases for flame, and discuss the artifacts created by the injection procedure.

4.1 Additive Anomaly

We illustrate the case of an additive anomaly with a *TCP SYN network scan*. For additive anomalies we need to define a *generator model*. Moreover, the tool chain configuration contains the reader, writer, generator, and merger module. TCP SYN scans are well known and we have a precise idea of how they work: The host doing the scan sends TCP SYN packets to a large number of IP addresses. Each scan may eventually result in a reply packet. This reply can be either a TCP SYN/ACK, a TCP RST, or an ICMP destination unreachable packet from the last-hop router. Consequently, our generator model gets a target IP range, a scanning rate, a start and end time, and a target port as parameters. It selects a destination IP address out of the given range and generates the TCP SYN packet header with the source IP address of the selected scanning host, a randomly chosen source port (> 1023), the targeted destination port, and a time stamp derived from the start time and the scanning rate. In this case, one could also generate the flows directly since each packet will form an individual flow anyway (no two packets have the same 5-tuple that defines a flow). A reply packet is generated for a given fraction of all sent TCP SYN packets. This fraction could also be configured by the user. Moreover, the type of reply packet sent is chosen according to a given distribution of TCP SYN/ACK packets, TCP RST packets, and ICMP destination unreachable packets. A TCP reply packet is generated by inversing the source and destination IP addresses and selecting a random source port. The target port is used as destination port, and the flag is either SYN/ACK or RST. For ICMP packets, the source IP address is set to the address of the last-hop router (e.g., same network but last 8 bits set to 1), a default source and destination port are used. The time stamp for the reply packet is set to the time stamp of the triggering TCP SYN packet plus a constant delay and a random offset. This is a basic model that makes several simplifying assumptions: 1) a given distribution of reply types, and 2) a constant round-trip time plus random offset. Whether this model is correct with respect to the accuracy required by target anomaly detection systems will be analyzed in future work. In particular, we see two possible sources for injection artifacts: first, the reply type might not be appropriate for internal machines, and second, the constant delay might not be realistic. In future work, we will provide the means to parameterize a model based on the information available in the existing background trace.

4.2 Subtractive Anomaly

We illustrate the case of a subtractive anomaly with the example of an *ingress shift*. For subtractive anomalies we need to define a *deleter model*. Moreover, the tool chain configuration contains the reader, writer, and deleter module. Ingress shifts are caused by routing changes in other Autonomous Systems that result in traffic shifts to a different PoP [10]. Here, we model the ingress shift at the PoP (router) where traffic is shifted away from. At the router where traffic is shifted to, an additive model would be used instead. Our deleter model for a ”negative” ingress shift is thus the following: It deletes each flow, which has its source IP address within the (configured) IP address range that has been shifted to another router. Moreover, the user needs to specify the start and duration of the ingress shift anomaly. With this type of anomaly we do not expect any injection artifacts.

4.3 Interactive Anomaly

Due to the limited space, we only briefly illustrate the case of an interactive anomaly with the example of a *TCP SYN flooding Denial of Service attack*. For interactive anomalies we need to define a *generator model* and a *deleter model*. Moreover, the tool chain configuration contains the reader, deleter, generator, merger, and writer module. The deleter model is parameterized with the start and end of the anomaly, the IP address of the attacked server, and the probability that replies are dropped by the server. It deletes each flow originating from the attacked server with the given probability. The generator model generates TCP SYN packets at a specified (constant or variable) flooding rate, and TCP

[2] We used Netflow v5 data, which aggregates 28-30 flows in a flow packet, for these tests.

SYN/ACK replies with a constant delay plus random offset, and a defined probability that the server generates a reply. We expect this model to create several injection artifacts since the drop rate is not coupled to the load on the server, and the load-induced slow-down of replies is not considered.

5 Related Work

This present work was very much inspired by the paper on packet trace manipulation from Rupp et al [8]. The authors present several basic manipulation operations for packet traces, namely, merging, adapting, stretching or compressing, moving, and duplicating. The main difference is that they work with packet traces, while we rely on flow traces. Moreover, they do not provide methods for generating anomalous traffic, but rely on existing packet traces that already contain attack traffic. In contrast, our focus is to simulate the anomalous traffic since this gives us full control about the characteristics of the anomalous traffic. A second source of inspiration was the work by Mirkovic et al [7] on modeling denial of service attacks and countermeasures. We will try to explore the knowledge about DoS attack characteristics gained in this work. Our approach differs from this work in two ways: first we use an existing trace as background traffic, and second we do not restrict ourself to denial of service attacks. Trident, developed by Sommers et al [9] is another tool for generating malicious and benign IP packet traffic traces. Benign traffic is generated using application-specific automata and a tool called Harpoon, which was developed earlier by the authors. Attack traffic is generated with MACE, which was extended to support 21 different attacks (e.g., Welchia, teardrop, synflood) for this work. Trident focuses on the evaluation of packet-level NIDS such as Bro, while our goal is to evaluate flow-based anomaly detection systems. In contrast to NIDS, these systems do not rely on individual packet header or payload characteristics but on link- or host-level statistics such as flow or packet counts.

6 Conclusion

We have presented flame, an anomaly injection framework that allows to combine recorded real traffic data with synthetic anomalous traffic for evaluating statistical anomaly detection systems. Flame uses a constructive anomaly modeling approach that simulates the cause of an anomaly rather than its effects. We have described our anomaly injection methodology, which relies on two modification operations applied to existing background traces: flow deletion and flow generation/merging. We also presented a prototype implementation of flame, discussed architectural aspects, and gave performance figures. Finally, we presented three use cases for flame: the injection of a network scan, an ingress shift, and a denial of service attack, and discussed possible injection artifacts. Future work includes development and evaluation of basic anomaly models, and the application of flame to evaluate different anomaly detection approaches. Of further interest is also the use of flame for an investigation into the effects of data reduction techniques, such as sampling and anonymization, on anomaly detection approaches. The flame prototype implementation is available under the GNU GPL upon request from the authors.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Program (FP7/2007-2013) under grant agreement no. 216585 (INTERSECTION Project).

References

[1] Brauckhoff, D., Tellenbach, B., Wagner, A., May, M., and Lakhina, A. Impact of packet sampling on anomaly detection metrics. In *IMC 2006* (2006).

[2] DDoSVax. `http://www.tik.ee.ethz.ch/~ddosvax/`.

[3] Dübendorfer, T., Wagner, A., and Plattner, B. A Framework for Real-Time Worm Attack Detection and Backbone Monitoring. In *IWCIP 2005* (2005).

[4] Lakhina, A., Crovella, M., and Diot, C. Mining Anomalies Using Traffic Feature Distributions. In *ACM SIGCOMM 2005* (2005).

[5] Lippmann, R., Fried, D., Graf, I., Haines, J., Kendall, K., McClung, D., Weber, D., Webster, S., Wyschogrod, D., Cunningham, R., and Zissman, M. Evaluating Intrusion Detection Systems: The 1998 DARPA Offline Intrusion Detection Evaluation. In *DARPA Information Survivability Conference* (2000).

[6] McHugh, J. The 1998 Lincoln Laboratory IDS Evaluation. In *RAID 2000* (2000).

[7] Mirkovic, J., Wei, S., Hussain, A., Wilson, B., Thomas, R., Schwab, S., Fahmy, S., Chertov, R., and Reiher, P. DDoS benchmarks and experimenter's workbench for the DETER testbed. In *Tridentcom 2007* (2007).

[8] Rupp, A., Dreger, H., Feldmann, A., and Sommer, R. Packet trace manipulation framework for test labs. In *IMC 2004* (2004).

[9] Sommers, J., Yegneswaran, V., and Barford, P. A framework for malicious workload generation. In *IMC 2004* (2004).

[10] Teixeira, R., Duffield, N. G., Rexford, J., and Roughan, M. Traffic matrix reloaded: Impact of routing changes. In *PAM 2005* (2005).

[11] Wagner, A., and Plattner, B. Entropy based worm and anomaly detection in fast IP networks. In *WET ICE 2005* (2005).

[12] Xu, K., Zhang, Z.-L., and Bhattacharyya, S. Profiling Internet backbone traffic: behavior models and applications. In *SIGCOMM 2005* (2005).