# OpenSAFE: Hardware-Based Network Monitoring Using Software Control

Aaron Gember, Jeffrey R. Ballard, Brian Kroth and Aditya Akella
*University of Wisconsin–Madison*

## Abstract

Administrators of today's networks are highly interested in monitoring traffic for purposes of collecting statistics, detecting intrusions, and providing forensic evidence. Unfortunately, network size and complexity can make this a daunting task. Aside from the problems in analyzing the network traffic itself for this information—an extremely difficult task on its own—a more fundamental problem exists: how to direct the traffic for network analysis and measurement in a flexible, high performance manner.

Current solutions fail to fully address the challenges of directing traffic for both on- and off-path monitoring. In this paper, we propose OpenSAFE, a system for enabling the arbitrary direction of traffic for security monitoring applications at line rates. Flexible policies are specified in ALARMS, a flow specification language that greatly simplifies management of network monitoring appliances. Finally, we demonstrate our OpenSAFE implementation using both live network traffic and replayed traces. Analysis shows that our OpenSAFE implementation handles higher traffic volumes than our existing monitoring infrastructure.

## 1 Introduction

Networks are traditionally monitored for many purposes including performance optimization, usage tracking, security and intrusion detection, compliance verification, and forensic analysis. There are two common approaches to monitoring and measurement today: 1) on-path middleboxes and 2) off-path traffic mirrors. The former set of approaches directly receive and affect network traffic before forwarding it on to its destination, whereas the latter employs a copy or span of network traffic at interesting points in the network to monitor without affecting it. In addition, growing link speeds and network fan-out are making effective network monitoring even more challenging. Unfortunately, both sets of approaches suffer from key issues pertaining to performance and flexibility.

Firstly, achieving complex on- or off-path monitoring functionality is quite tricky today. Each of the above application domains places different constraints on monitoring techniques. Unfortunately the constraints can work against each other, making the monitoring functions ill-suited for one underlying platform. Furthermore, hardware limitations often prevent the ability to provide multiple traffic monitoring ports on a single router. Thus, operators are often forced to daisy-chain multiple monitoring devices to achieve complex monitoring functionality, which is a rigid and failure-prone configuration. The introduction of new monitoring mechanisms often requires difficult physical rewiring and complex reconfiguration of monitoring hardware/software, resulting in outages. Secondly, the arriving network traffic can quickly overwhelm the monitoring computer, rendering it useless. To overcome these two problems, network administrators can deploy special purpose load-balancing or traffic-splitting mechanisms. Hardware traffic-splitters (e.g., SPANIDS [20] and Xinidis et al. [25]) are fast, but they are expensive and difficult to program. Software-based load-balancers (e.g., those based on Click [15]) are inexpensive and flexible as they are easy to reprogram based on observed traffic patterns, but they impose latency and throughput penalties.

Our goal is to design a monitoring framework that has the flexibility and ease of configuration realizable through software-based control along with the speed of approaches based on hardware forwarding. To this end, we introduce *OpenSAFE*, which uses a commodity software-programmable switch to direct traffic in flexible ways to meet complex monitoring requirements.

Several inexpensive unoptimized monitoring devices, subsets of which may be performing a given monitoring function, can be "plugged into" OpenSAFE to process programmatically-defined substreams of data. We also introduce a high-level language, ALARMS, to enable network administrators to flexibly express rich policies to control how traffic is distributed to various monitoring devices. The policy is expressed at a logically central monitoring controller which then instantiates policy constraints in the form of forwarding entries in the programmable switch's flow table. In effect, OpenSAFE couples a high-performance hardware-based monitoring dataplane with a flexible software-based configuration/control plane, resulting in a highly effective, yet low-cost, monitoring framework.

OpenSAFE introduces design abstractions for monitoring elements and functionality, such as mirror ports, monitoring devices of different kinds, and various mechanisms for controlling traffic. Our ALARMS language expresses OpenSAFE's abstractions in a simple policy language syntax. OpenSAFE and ALARMS are designed to allow operators to instantiate and/or update very sophisticated monitoring setups with relative ease. The abstractions and policy language in Open-

SAFE/ALARMS are motivated by those in Click [15], but are more monitoring-specific. In theory, Click can also enable software-programmable network monitoring, but OpenSAFE offers key flexibility and performance advantages. OpenSAFE's architecture allows it to orchestrate traffic distribution amongst multiple monitoring locations via a simple policy specified at the (lone) controller, enabling richer and more flexible monitoring functionality. In contrast, Click and other solutions require configuration at each monitoring point in the network. Click's reliance on CPU processing of packets imposes significant latency penalty, whereas OpenSAFE's hardware fast path removes this overhead for an overwhelming fraction of packets.

We conduct a thorough evaluation of OpenSAFE using both live production traffic and replayed traces. We implement OpenSAFE and ALARMS using the NOX/OpenFlow platform [12, 17]. We show that it outperforms an existing, highly-optimized monitoring device in a multiple day head to head test. We examine 54% more packets and generate 30% more security alerts during our four day test using an *unoptimized* team of machines and OpenSAFE. Furthermore, we show that OpenSAFE scales with increasing bandwidth and significantly outperforms an in-kernel Click configuration.

## 2 Challenges and Design Requirements

Modern networks may employ both on-path or off-path monitoring. On-path middleboxes are used commonly because of their ability to manipulate live traffic. Unfortunately, additions, deletions or modifications of middleboxes lead to outages and reconfiguration of network gear. This results in network interruptions and performance loss. On-path middleboxes also need to be capable of processing all traffic traversing a particular network link, even if the processing of some packets only involves simple forwarding. High traffic volumes can inundate on-path middleboxes and degrade network performance. OpenSAFE can help address all of these challenges in on-path monitoring.

The alternative to on-path middleboxes is to take a *mirror* (or tap) of traffic at a border point and examine the traffic off-path. While an off-path monitoring device does not impact production traffic due to reconfiguration outages or traffic overload, hardware limitations often prevent the ability to provide multiple traffic mirrors. This limits the number of monitoring devices that can effectively participate. For example, the Cisco Catalyst 6000 series is limited to two mirror ports per device. Making it worse, enabling multicast on a Cisco FireWall Services Module (FWSM), consumes one of the mirror ports–leaving only one for monitoring. Commonly, network operators connect this single mirror port into an expensive computer that has been heavily optimized to move traffic very fast (for example, by using PF_RING [7]), leaving little room for error. The heavy tuning often results in brittle software configurations. At times even slightly different revisions of software make a huge impact in these monitoring computers.[1] Even with high tuning, a single off-path monitoring device can become overwhelmed with traffic and lose large amounts of data by randomly dropping packets. A random drop policy affects the ability for the monitoring device to both fully examine all traffic and accurately reassemble network flows. OpenSAFE can address these challenges in off-path monitoring as well.

### 2.1 Design Requirements

To address the challenges discussed above, OpenSAFE must meet the following design requirements:

- **The ability to handle high-speed links** ensures the monitoring infrastructure can process *all* traffic at contemporary link speeds of 1 Gbps, 10 Gbps, or higher. The system must also support multiplexing traffic amongst multiple parallel monitoring devices which process traffic at slower link speeds.
- **Support for flexible configurations** is required to meet the diverse, and often conflicting constraints, of multiple monitoring domains. The ability to monitor specific subsets of traffic and process traffic with multiple monitoring devices is essential.
- **No service interruptions** should be caused by configuration changes or monitoring device failures. Production traffic should remain unaffected.
- **Compatibility with existing monitoring solutions** ensures admins can integrate their existing intrusion detection, traffic capture, packet counting, and other monitoring tools into the monitoring infrastructure.
- **Easy management** avoids the configuration complexities of todays highly tuned systems, allowing admins to focus on analyzing monitoring output instead of managing monitoring infrastructure.

### 2.2 Existing Systems

Many commercial and research systems have been developed for monitoring enterprise networks in either an on-path or off-path fashion. However, as shown in Table 1, existing systems still lack some key design features.

The NIDS Cluster by Vallentin et al. [24] and the stateful intrusion detection system by Kruegel et al. [16] both use commodity PCs and flow-based hashing to distribute traffic from a mirror or a link to a set of parallel monitoring devices all performing the same function. However, the link speed both systems can handle is limited

[1]For example, in our initial tests we found that a minor revision of our IDS software dropped almost 50% more packets on our production monitoring system.

| System | Forwarding Hardware | Data Plane | Control Plane | Monitoring Devices | Distribution Basis |
|---|---|---|---|---|---|
| OpenSAFE | Programmable network fabric | Hardware | Software | Any configuration | Flows (see Section 3.2) |
| NIDS Cluster [24] | Commodity PCs | Software | Software | Parallel sensors | Flows (hashing) |
| Stateful Intrusion Detection [16] | Commodity PCs | Software | Software | Parallel sensors | Flows (filters) |
| SPANIDS [20] | FPGA | Hardware | Hardware | Parallel sensors | Flows (hashing) |
| Active Splitter [25] | Programmable network processor | Hardware | Hardware | Parallel sensors | Flows (hashing) |
| Click Modular Router [18] | Commodity PC | Software | Software | Any configuration | Packets |

Table 1: Existing systems for multi-device on- or off-path monitoring

by the software-based data planes they employ. Additionally their use is limited to a set of monitoring devices performing the same function in parallel.

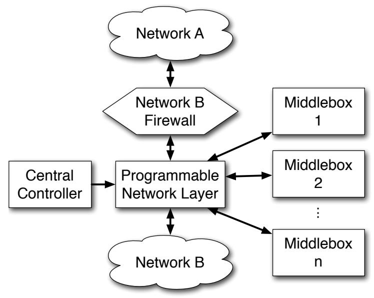
SPANIDS by Schaelicke et al. [20] and the active splitter architecture by Xinidis et al. [25] employ a hardware-based data plane. SPANIDS employs FPGA hardware and Xinidis et al. use programmable network processors. Fields in the packet headers are hashed to load balance flows amongst multiple monitoring devices operating in parallel. Unfortunately, both systems are still limited to parallel identical monitoring devices. Additionally, both the data plane and control plane must be programmed into hardware using low-level assembly, making it difficult to modify the systems' behavior.

The Click Modular Router [18] is most similar to OpenSAFE because of its ability to support arbitrary configurations of monitoring devices. Its element-based model and simple specification language is similar to ALARMs, except that ALARMS is more narrowly focussed on monitoring and includes constructs that are missing in Click. Despite the similarity, new challenges arise in realizing OpenSAFE/ALARMS because of the fundamental difference in the platforms over which forwarding (of traffic to various monitoring devices) is realized: at a powerful CPU in Click, vs. directly at the forwarding tables of dumb switches in OpenSAFE (see next section).

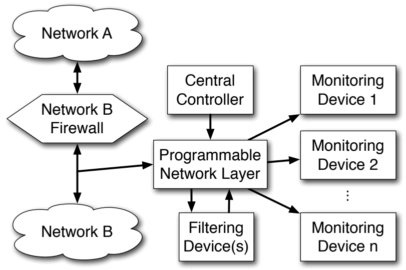
A Click based approach has key disadvantages as well. A crucial disadvantage of Click is its use of a software-based data plane which can be overwhelmed by the high-speed links present in contemporary enterprise networks. In particular, even with recent enhancements [9], CPU-based packet processing in Click imposes undesirably high per-packet latencies. Another downside is that Click does not allow easy simultaneous control over multiple monitoring locations, as each location needs to be independently configured. In contrast, by design, OpenSAFE allows a single network controller to use simplistic policies to centrally orchestrate multiple network monitoring locations, thereby enabling richer and more flexible monitoring functionality.

## 2.3 Using programmable network fabrics

The key problem in monitoring is not being able to exercise fine-grained control over network traffic. We observe that by inserting a programmable network fabric at the monitoring point we can dramatically increase the



(a) Monitoring live traffic on-path.



(b) Monitoring mirrored traffic off-path.

Figure 1: The desired dynamic layer.

utility of network traffic and the flexibility of monitoring, while at the same time dramatically reducing the effort needed to engineer and manage the monitoring functionality. An on-path and off-path example of this are shown in Figure 1. The framework, which we call OpenSAFE, demultiplexes high-bandwidth packet streams into several lower-bandwidth flows that are directed to different monitoring devices. A central controller determines how the fabric is configured. We briefly outline the key advantages of OpenSAFE.

(1) Our approach allows for *flexible, fast and scalable monitoring*. This has at least two different aspects to it. First, when applied to off-path monitoring, our approach allows for flexible sharing of a single mirror port across multiple devices. Second, with a programmable network fabric, network flows are directed on demand, and new monitoring functions can easily be added/reconfigured without incurring outages. Ideally this will leverage the support of an intuitive declarative language to control traffic. Related to this is the issue of scaling the throughput of monitoring functions, particularly those that are very intensive. In such cases, OpenSAFE allows multiple devices to work on disjoint subsets of traffic in paral-

lel thereby improving the throughput significantly.

(2) Another benefit of using a programmable network fabric is that it works *per-flow*. Since decisions are made per-flow in hardware—unlike other packet distribution techniques [15, 21, 22]—the scale-up of network monitoring is facilitated in an more manageable way. Fundamentally, software performing intrusion detection or deep packet inspection will reassemble out of order packets to correctly process streams. This process is dramatically streamlined, however, by having per-flow operations in the programmable switch fabric. In addition, when seeing all the packets from a flow as opposed to a random subset of packets (e.g. due to some round-robin policy), the monitoring software is generally able to more accurately collect useful data.

(3) Our approach is *cost-effective*. Programmable network fabrics have become available for commodity prices today; for example, OpenFlow[17]—an approach to program a switch's flow table—is supported on commodity networking hardware via a firmware upgrade. In addition, the ability to demultiplex traffic into multiple low-bandwidth flows has the immediate advantage of allowing the use of commonly-available, inexpensive, and easy to manage 1 Gbps NICs on the monitoring devices rather than 10 Gbps NICs.

## 3 OpenSAFE

We propose OpenSAFE (Open Security Auditing and Flow Examination), a unified system for network monitoring and measurement. Leveraging a programmable network fabric, our system can direct network traffic in a flexible (programmable) fashion without sacrificing latency or throughput. OpenSAFE consists of three components: a set of design abstractions for codifying the flow of network traffic; ALARMS, a policy language for easily specifying traffic paths (Section 4); and a controller that implements the policy using OpenFlow (Section 5). Unless otherwise specified, we describe OpenSAFE assuming an off-path monitoring configuration; the same design can be applied for on-path setups.

### 3.1 Paths

OpenSAFE is designed around several simple primitives to make the direction of network flows for network monitoring both flexible and easy. We use the notion of a *path* as the basic abstraction for describing the selection of traffic flows and the direction these flows should take. Fundamentally, we wish to support the construction of paths that allow desired traffic to enter the system on a particular network port and be directed to one or more network monitoring systems, regardless of physical configuration. A basic example of this is shown in Figure 2, where mirrored HTTP traffic is sent first through a counter appliance and then to a TCP dump appliance.

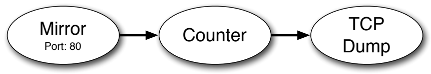


Figure 2: A basic monitoring path.

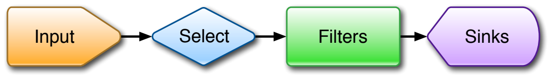


Figure 3: Abstractions to describe monitoring paths.

Paths are composed of several components: *inputs*, *selects*, *filters*, and *sinks*. At a high level, each path begins with an input, applies an optional select criteria to select a desired subset of network flows, directs matching traffic through zero or more filters, and ends in one or more sinks (Figure 3). Inputs can only produce traffic, sinks can only receive traffic, and filters must do both.

If we take Figure 2 and view it with these abstractions, it becomes Figure 4. This shows traffic entering on a mirror port (input) matching our criteria of port 80 (select), passing through a counter (filter), and ending at a TCP dump (sink). The same abstractions apply when OpenSAFE is used on-path: live traffic enters through an input and returns to the production network at a sink. Figure 5 shows a more complicated on-path example involving multiple filters, demonstrating how paths can be extended and applied to live traffic. A typical OpenSAFE configuration consists of multiple paths treated in aggregate.

### 3.2 Parallel Filters and Sinks

To monitor large networks at line rates it is possible (and quite likely) that a single middlebox or monitoring device will not be able to cope with all the network traffic. To address this problem, we allow traffic to be sent to multiple filters or sinks operating in parallel within a path. Figure 6 shows such a path for a mirror-based setup, with HTTP traffic sent to multiple IDS appliances.

The division of traffic between multiple filters or sinks is handled using *distribution rules*. Rules are applied on a per-flow basis. Existing parallel monitoring systems (e.g. [20, 24, 25]) only support distribution based on hashing. OpenSAFE supports five methods of distribution amongst a set of parallel components:

- *ALL*—send a flow to every component in the set
- *RR*—alternate flows between components of the set using Round Robin
- *ANY*—randomly select a component from the set
- *HASH*—apply a hash function on the first packet of a flow to select a component
- *PROB*—apply a probability function to load balance flows amongst components

Distribution rules (except for *ALL* rules) are considered *dynamic*—the path a particular flow follows is de-

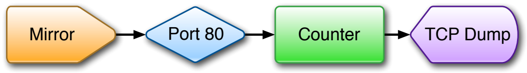


Figure 4: A basic logical monitoring path (Figure 2) with coded abstractions.

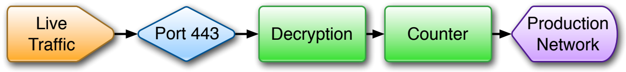


Figure 5: A logical monitoring path with multiple filters.

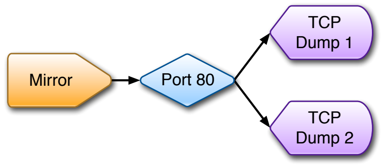


Figure 6: A monitoring path with parallel sinks.

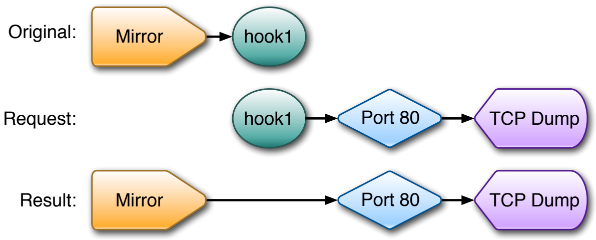


Figure 7: An example of a hook (top line). The middle line is a hook request made by a device and the bottom line is the resulting path implemented by OpenSAFE.

termined at runtime when the first packet of a flow traverses the distribution rule. Hooks, described below, are also dynamic. In contrast, the other portions of a path are considered *static*—the path taken by all flows is constant. The difference between *static* and *dynamic* rules has implications for how paths are implemented in the programmable network fabric, as described in Section 5.

### 3.3 Hooks

One issue that can arise when splitting monitoring traffic among multiple devices is that flows from a particular host (or a potential adversary) can be directed to separate machines. While information about the flows can be aggregated after the fact, it may be useful for monitoring software to examine all future traffic from a host after suspicious activity is detected. This requires the capability to add new paths at runtime. In OpenSAFE, *hooks* provide this functionality.

Monitoring devices can make hook requests at runtime to have new paths added to the current OpenSAFE configuration. A hook request effectively duplicates the path containing the hook and appends the path specified by the monitoring device. For example, Figure 7 shows a path with a hook and Figure 2 shows a potential resulting path based on a hook request to send HTTP traffic to *counter* followed by *TCP dump*.

### 3.4 Overall Design

The overall design of OpenSAFE is shown in Figure 8. The input is a connection from the chosen network aggregation point to a port on our programmable switch. Some number of filters (i.e. middleboxes) are used, attached to various switch ports. Finally, output is directed to some number of sinks.[2] Optionally, multiple switches can be used, assuming they are directly connected; paths can be defined between ports on any of the switches.

All switches are controlled by a logically central controller. This allows administrators to manage the entire monitoring infrastructure from a single point. In contrast, Click [15] and existing load balancing techniques [20, 25] require configuration changes to be made at each Click router or load balancer. A single policy at the controller not only eases management, but also helps ensure consistent monitoring across the network. Installing backup controllers helps avoid a single failure point.

[2]An on-path setup typically has a single sink to return traffic to the production network.

## 4 ALARMS: A Language for Arbitrary Redirection for Measuring and Security

To enable network administrators to easily manage and update their monitoring infrastructure, we introduce ALARMS, a language to enable the arbitrary redirection of network flows for measurement and security purposes. ALARMS represents the abstractions mentioned in Section 3 in a simple policy language syntax. Each component is defined with a name and parameters, and paths are defined between the named components. In this section we present the syntax of ALARMS; details about the implementation of policies are described in Section 5.

Most of the examples in this section and Section 5 are intended for mirror-based off-path monitoring. However, the same language constructs and implementation details apply to on-path monitoring, unless otherwise noted.

### 4.1 Component Declarations

In ALARMS, all components of a path are given unique types and names. Specifically, the policy file declares the following components: *Switches*, *Inputs*, *Sinks*, *Selects*, *Hooks*, and *Waypoints*. We describe the language specification and parameters for each of these below.

**Switches** Each switch is declared with a unique name and the identifier of the programmable switch fabric:

```
switch sw = 0x00000021;
```

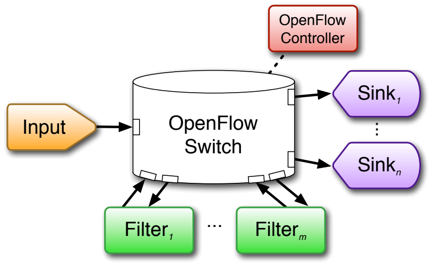


Figure 8: The overall design of OpenSAFE, using our abstractions.

**Inputs and Sinks** Inputs and sinks are simply named switch ports (as in Figure 8), declared like so:

```
input mirror = sw:0;
sink tcpdump = sw:1;
```

Since inputs can only transmit traffic and sinks can only receive traffic, each named input or sink is restricted to a single port. Traffic can be directed to multiple sinks using distribution rules (Section 4.2.2). A special default sink named `discard` drops all traffic sent to it. For on-path monitoring, an input provides live traffic and a sink sends live traffic back into the production network.

**Filters** Middleboxes within an OpenSAFE network are called filters. A filter is a combination of an input and a sink, shown as the third item in Figure 3. As such, filters are declared with a single `tofrom` switch port (to both receive and transmit on the same port) or both a `to` and a `from` port (to delegate receiving and transmitting, respectively, to separate ports):

```
filter to counter = sw:2;
filter from counter = sw:3;
```

**Selects** Selects are named criteria used to limit traffic flows based on fields in packet headers. ALARMS supports selecting on any of 9-different header fields: Ethernet source and destination addresses, EtherType, VLAN identifier, network source and destination addresses, transport protocol, and transport source and destination ports. Additionally, an arbitrary number of bits can be declared as wildcards for network source and destination addresses to provide for CIDR-like address ranges. Limited boolean logic (AND, OR) can be used in a select definition to specify criteria on multiple header fields. Any header fields not specified in the select are treated as wildcards. The example select below yields only traffic whose source or destination port is 80:

```
select http = tp_src: 80 || tp_dst: 80;
```

**Hooks** Path requests made at runtime are facilitated via hooks. In ALARMS they are declared with only a name:

```
hook hook1;
```

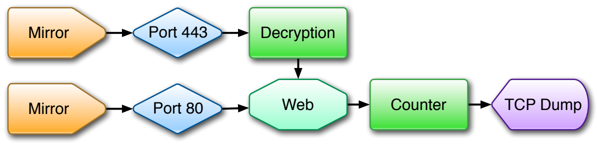


Figure 9: A logical monitoring path with a waypoint.

**Waypoints** In a system of reasonable size, it is probable to have multiple paths configured with common attributes. For instance, suppose an administrator wants to perform some degree of processing on one of two sets of traffic, then send the results of both to the same filter and sink. This quickly becomes a maintenance problem as modifying the common end-components of the paths may involve editing many paths.

The final component type in ALARMS is an abstraction added as a convenience to ease the creation and management of multiple, semi-redundant paths. *Waypoints* serve as "virtual destinations" and "virtual sources" allowing administrators to aggregate paths and reduce repetition. A path using a waypoint is displayed in Figure 9, where HTTP and HTTPS traffic is sent to a `web` waypoint before being passed to a counter filter and TCP dump sink. Declaring waypoints requires only a name:

```
waypoint web;
```

## 4.2 Paths

After all named components have been declared, we can connect these components to form paths. The declaration of paths is similar to the language used in Click [15], except OpenSAFE paths are designed to direct flows rather than individual packets. Paths in ALARMS must conform to the following specification:

1. Paths begin with an input, waypoint, or filter.
2. Paths end with a sink, waypoint, filter, hook, or rule.
3. Selects can be applied to any connection between components.

The path in Figure 4 can be written in ALARMS as:

```
mirror[http] -> counter -> tcpdump;
```

### 4.2.1 Paths with Selects

A select limits the traffic seen by all components in the path downstream from the select. In the path above, the filter (`counter`) will only see HTTP traffic coming from the input (`mirror`) and the sink (`tcpdump`) will only see HTTP traffic leaving the filter. Each connection in the path is limited to having one select.

If a path has multiple selects, the selects downstream further restrict upstream selects, with the downstream select taking precedence in the case where both specify criteria for the same header field(s). For example, a revised path with an additional select will result in the filter still seeing all HTTP traffic from the mirror port, while the sink now sees only HTTP traffic for a particular server:

```
select webserver = nw_src: 10.0.0.1
    || nw_dst: 10.0.0.1;
mirror[http] -> counter[webserver] -> tcpdump;
```

#### 4.2.2 Distribution Rules

The distribution of traffic between multiple components (excluding inputs) is handled by distribution rules, applied on a per-flow basis.

The first three distribution methods, `ALL`, `RR`, and `ANY`, each take a list of components to act on. In the on-path example below, the rule will round-robin new HTTP flows between two counter middleboxes before returning traffic to the production network:

```
live[http] -> {RR, counter1, counter2}
    -> production;
```

`HASH` and `PROB` rules take an additional argument—the name of the hash or probability function—and rely on the output of this function to determine the destination. Probability rules are designed to allow OpenSAFE to distribute traffic based on the current load of distribution components, so the user must also provide a way for the function to receive load information from components. For example, the following policy instructs OpenSAFE to use a user defined hash function `myhash` to distribute new flows between two counter filters before they proceed to a tcpdump sink. This could be more desirable than a `RR` rule since it can be deterministic.

```
mirror[http]
    -> {HASH(myhash), counter1, counter2}
    -> tcpdump;
```

Distribution rules to distribute traffic amongst parallel sinks should only be utilized in mirror-based monitoring environments. Policies for on-path monitoring should, in most cases, send all traffic to a single sink which reintroduces the traffic to the production network.

## 5 Programming the Fabric

Direction of traffic is realized by programming the network fabric based on an ALARMS policy file. Programming the fabric consists of three tasks:

1. Parse the policy file written in ALARMS.
2. Install static flows when a new switch connects to the controller.
3. Install dynamic flows when a packet is received by the controller, or upon hook request.

Our network fabric consists of an OpenFlow [17] switch and NOX controller [12]. An OpenFlow switch forwards packets in the data plane based on a programmable flow table. The flow table consists of entries that contain values for up to ten different packet header fields (known as the *OpenFlow 10-tuple*).[3] Any fields in the 10-tuple for which values are not specified are treated as wildcards. Each entry also contains an action that should be applied to packets matching that entry: drop, output to one or more ports, or send to the controller. When a packet arrives at an OpenFlow switch, it is matched against the entries in the flow table. The actions of the highest priority matching entry are applied to the packet. If the packet does not match any entry in the flow table, it is forwarded to the controller for a decision to be made. While other all-ASIC options exist (such as ones from GigaMon[10]), OpenFlow is readily available today on commodity hardware.

### 5.1 Policy Parsing

ALARMS policy parsing performs two key actions on paths. First, each path is checked to verify it meets the three criteria outlined in Section 4.2. Second, overlapping paths are identified and combined. Combining paths avoids overwriting the flow table entries for an earlier path with flow table entries for an overlapping path defined later in a policy. A set of paths with the same first component and select criteria are internally combined by applying the following transformation rule: *Given a set of paths* $\alpha \rightarrow \beta_1, ..., \alpha \rightarrow \beta_n$, *where* $n \geq 2$, *remove the existing paths and add new paths* $waypoint_{\alpha_1} \rightarrow \beta_1, ..., waypoint_{\alpha_n} \rightarrow \beta_n$, $\alpha \rightarrow \{ALL, waypoint_{\alpha_1}, ..., \ waypoint_{\alpha_n}\}$. For example, the set of overlapping paths

```
mirror -> counter -> tcpdump1;
mirror -> counter -> tcpdump2;
```

are internally combined to form non-overlapping paths

```
counter -> {ALL, waypoint1, waypoint2};
waypoint1 -> counter -> tcpdump1;
waypoint2 -> counter -> tcpdump2;
```

### 5.2 Static Flow Installation

The process to program the network fabric based on an ALARMS policy begins with a fundamental observation: hardware is faster than software. In OpenFlow, forwarding a packet which matches an existing flow table entry is faster than sending a packet to the controller. To preserve high performance, we pre-compute as many routes as possible and install them in the flow table of the OpenFlow switch on startup. This avoids the need to contact the controller for every new flow and prevents the controller from being overloaded with traffic—a distinct possibility when operating at high line rates. We call these pre-computed flow table entries *static flows* since they remain in the switch's flow table the entire time OpenSAFE is running. Static flow table entries are installed for *static* path components (see Section 3.2) and to send traffic for *dynamic* components to the controller.

[3]The allowable header fields are the nine fields specifiable in selects defined in ALARMS, plus a field for incoming switch port.

### 5.2.1 Default Drop

By default, an OpenFlow switch automatically sends to the controller any traffic for which there is no matching flow table entry. In contrast, ALARMS specifies paths for only certain traffic, assuming all other traffic is discarded. To reconcile the differences between ALARMS and OpenFlow, we install low-priority wildcard rules to drop all traffic entering the switch from inputs or filters. These drop rules avoid the overhead of sending unwanted packets to the controller. All paths defined in the ALARMS policy file are installed with higher priority so desired traffic is not dropped.

When employing OpenSAFE on-path, traffic not examined by monitoring devices should simply be forwarded. Since ALARMS assumes default drop, administrators need to include in the policy file a simple forwarding path from live traffic input to production network sink. More specific paths (i.e. paths with selects) will still take precedence over this simple forwarding path because OpenFlow gives precedence to matching flow table entries with fewer wildcard fields before selecting an entry with more wildcard fields.

### 5.2.2 Input Paths

Static flow installation processes each path that begins with an input. At the beginning of a path, it is assumed that all flows (i.e. a 10-tuple of all wildcards) will traverse the path. The 10-tuple becomes more specific as each component and selection in the path is processed. The input port (*in_port*) field is updated when processing an input or a filter. A selection adds new tuple items or overrides existing values.

New flow table entries are typically installed at the switch for each transition (i.e. arrow, `->`) in a path. For example, the path in Figure 2, written in ALARMS as

```
mirror[http] -> counter -> tcpdump;
```

results in two flow entries for the first transition

$\{tp\_src=80,\ in\_port=0\} \rightarrow output{:}2$
$\{tp\_dst=80,\ in\_port=0\} \rightarrow output{:}2$

and two flow entries for the second transition

$\{tp\_src=80,\ in\_port=3\} \rightarrow output{:}1$
$\{tp\_dst=80,\ in\_port=3\} \rightarrow output{:}1$

### 5.2.3 Filters and Waypoints

Paths ending with a filter require "expanding" the path to also process all paths succeeding from the filter. In the example below, processing the path ending with the `counter` filter prompts the processing of the second path:

```
mirror[http] -> counter;
counter -> tcpdump;
```

When processing filter paths, the 10-tuple that existed at the end of the original path is used as the starting 10-tuple for each filter path. If we did not use the 10-tuple from the end of the original path, an incorrect flow table entry would have been installed:

$\{in\_port=3\} \rightarrow output{:}1$

The only traffic that should leave `counter` is the HTTP traffic that came in. Therefore, we start processing the second path with a 10-tuple specifying port 80 traffic.

Paths ending in waypoints are treated similarly to paths ending in filters. The waypoint is "expanded" and processing continues along each path beginning with that waypoint. The difference is that waypoints are merely conceptual and do not correspond to any physical ports on the OpenFlow switch. Flow entries are installed originating from the component preceding the waypoint in the original path, to the component(s) following the waypoint in the waypoint path(s). For example, the paths

```
mirror[http] -> web;
web[webserver] -> tcpdump;
```

result in only one set of flow entries

$\{tp\_src=80,\ nw\_src=10.0.0.1,\ in\_port=0\} \rightarrow output{:}2$
$\{tp\_src=80,\ nw\_dst=10.0.0.1,\ in\_port=0\} \rightarrow output{:}2$
$\{tp\_dst=80,\ nw\_src=10.0.0.1,\ in\_port=0\} \rightarrow output{:}2$
$\{tp\_dst=80,\ nw\_dst=10.0.0.1,\ in\_port=0\} \rightarrow output{:}2$

It is important to note that the current 10-tuple is first limited by the `http` select when the input `mirror` is processed, and the set of flows is further limited by the `webserver` select when the `web` waypoint is "expanded." If no path begins with a particular waypoint, uses of the waypoint are "expanded" to have a destination of the implicit `discard` sink.

### 5.2.4 Distribution Rules

The static flow table entries installed for distribution rules vary depending on the distribution method. An *ALL* rule can be treated statically—no packets need to be sent to the controller and all flow table entries can be installed at startup. For example, the path

```
mirror[http] -> {ALL, tcpdump1, tcpdump2};
```

results in the flow entries

$\{tp\_src=80,\ in\_port=0\} \rightarrow output{:}1,\ output{:}4$
$\{tp\_dst=80,\ in\_port=0\} \rightarrow output{:}1,\ output{:}4$

If more components exist in the path following the rule, path processing continues along the path as normal.

The other methods of distribution (*RR*, *ANY*, *HASH*, and *PROB*) require packets to be sent to the controller for the appropriate dynamic flow entries to be installed. For these rules, static flow entries are installed with the action of sending to the controller. For example, the path

```
mirror[http] -> {RR, tcpdump1, tcpdump2};
```

results in the flow entries

$\{tp\_src=80,\ in\_port=0\} \rightarrow controller$
$\{tp\_dst=80,\ in\_port=0\} \rightarrow controller$

When packets matching these entries are sent to the controller, it is necessary to know which rule should be applied. Therefore, we store the 10-tuple and the associated distribution rule at the controller for later reference.

## 5.3 Dynamic Flow Installation

Dynamic flow entries are installed for distribution rules (excluding *ALL* rules) and upon receipt of a hook request. These flow entries cannot be pre-computed at OpenSAFE startup because the destination switch ports are unknown until flows arrive or requests are received.

### 5.3.1 Distribution Rules

Flow entries for dynamic rules are installed when a new flow matches an entry whose action is "send to controller." The controller receives the first packet in the flow and determines which rule should be applied to the flow. Only the matching rule needs to be processed; the rest of the path containing the rule is already processed during static flow installation. For *HASH* or *PROB* rules, the controller calls user specified code to select one or more destination components. A destination is selected at random for an *ANY* rule, and the next component in the list of possible destinations is selected for a *RR* rule. Dynamic flow entries for distribution rules contain a fully-specified 10-tuple with all values populated from the packet headers. Entries are also installed for flows going in the reverse direction to ensure that both halves of a flow traverse the same path. OpenSAFE uses a default timeout of 30 seconds for dynamic flow entries.

### 5.3.2 Hooks

The controller listens on a network socket for hook requests. Monitoring devices send an XML fragment which contains the name of the hook, the name of the component to which traffic should be sent, values for one or more fields in the *10-tuple*, and the duration the hook entry should last. The controller installs a high-priority flow entry with the appropriate *10-tuple*, timeout, and output action. The *in_port* value in the flow table entry is determined based on the component that precedes the hook component in the hook path.

Dynamic flow entries installed for hooks do not consider the rest of the paths specified in an ALARMS policy. If a hook request overlaps with an existing path, the hook request takes precedence. For example, assume the following set of paths:

```
mirror[http] -> tcpdump1;
mirror -> hook1;
```

A request for `hook1` to send all HTTP traffic to `tcpdump2` will result in all HTTP traffic going to `tcpdump2` instead of `tcpdump1` for the duration of the request. After the hook request times out, the flow entry for the first path will again take effect.

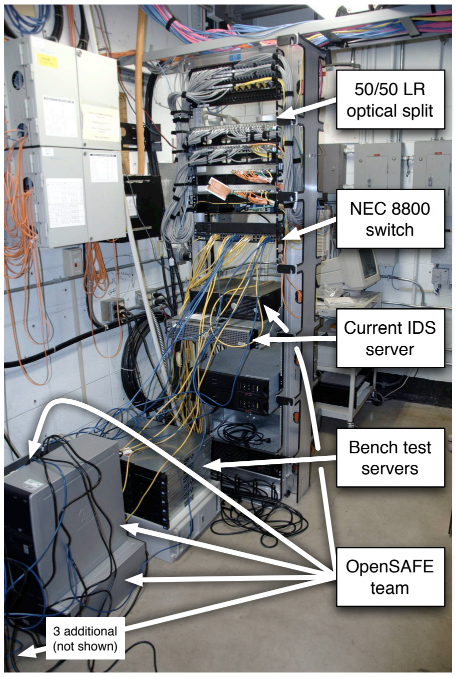


Figure 10: The head-to-head test platform.

## 6 Evaluation

OpenSAFE needs to handle traffic volumes at high line rates to be able to serve as a feasible network monitoring system. We verify OpenSAFE meets this requirement by measuring its performance using both live and replayed real-world traffic. First, we compare our implementation against an existing monitoring infrastructure and show that OpenSAFE loses less traffic (Section 6.1). Second, we run our implementation with varying rules sets using a constant set of traffic traces (Section 6.2). We demonstrate that OpenSAFE handles sustained amounts of high traffic volume and scales with increasing path sizes. Lastly, we compare OpenSAFE against an in-kernel Click [15] configuration (Section 6.3).

Our OpenSAFE implementation uses an OpenFLOW 0.8.9 enabled NEC IP8800 10 gigabit switch. The controller is written as a Python module for NOX 0.6.0.

## 6.1 Comparison to Existing Infrastructure

In this section, we compare OpenSAFE against an existing mirror-based monitoring system. We describe the existing setup in the College of Engineering at the University of Wisconsin—Madison, present our OpenSAFE setup, and discuss the test results. We observe that OpenSAFE analyzes more traffic and generates more security alerts than the existing system.

### 6.1.1 Test Setup

The existing production monitoring setup has been highly optimized with technologies such as PF_RING [7]

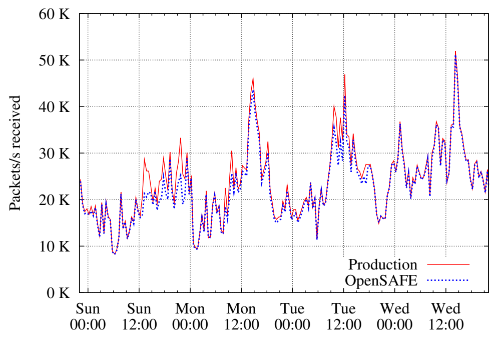


Figure 11: Packets per second received by the optimized production IDS system compared versus the OpenSAFE IDS team.

and TNAPI [8]. PF_RING is a special network socket that avoids excessive kernel memory copy operations, reducing packet loss during high bandwidth captures. TNAPI is a threaded network device polling method that makes interrupt handling more efficient on multi-cored machines. Both technologies require special kernel and application modifications that can make the system quite brittle. However, both have been shown to improve standard packet capture techniques by up to 100% [6, 7, 8].

The existing production system runs three pieces of monitoring software: Suricata [23], Barnyard2 [3], and nProbe [19], each compiled with PF_RING support. Suricata is a multi-threaded content matching IDS like Snort that uses the same rules and logging. Barnyard2 reads IDS alert logs and consolidates them in a remote location like a BASE [4] database. nProbe is a tool for collecting flow data in a distributed sensor fashion and reporting the data back to a collector. The production system's hardware is comprised of a single Dell PowerEdge 2950 with a 2.0 GHz Dual Core Xeon 5130 CPU and a 10-Gigabit Intel 82598EB XF LR server fiber adapter.

Our OpenSAFE monitoring setup was composed of six old or spare desktop machines attached to the NEC OpenFlow switch. The six machines' hardware specs included two HP xw4300s (Pentium 4 3.4 GHz), one Dell GX620 (Pentium 4 3.4 GHz), two Dell Optiplex 755s (3.0 GHz Core2 E8400), and one HP dc5800 (2.6 GHz Core2 Q9400). We refer to this group of machines as the "OpenSAFE team." Each machine was setup with the same monitoring software, configurations, and rules sets as the production system with the exception that we did not use PF_RING or TNAPI. While the comparative data we present comes solely from Suricata, we included the other monitoring software to maintain a fair test and to illustrate the diversity in monitoring techniques.

All traffic from the border router was sent to both the production machine and OpenSAFE using a 50/50 optical splitter on the single mirror port available on the router. Figure 10 is a picture of the head-to-head testing platform in the MDF in one of our buildings. We configured OpenSAFE to further split the traffic amongst our IDS desktops by statically partitioning the college's local subnets between the machines. Since neither our machines nor subnets are of equal capacity we used the traffic counts at configuration time as well as the load average of the individual IDS machines to attempt to manually balance the traffic. Given that traffic fluctuates over time this configuration was almost certainly suboptimal. A portion of our ALARMS policy file is below.

```
### define switches
switch switch1 = 0x12f2c720cc;

### define input ports
input mirror = of1:0;

### define sink ports
sink ids1 = of1:1;
...
sink ids6 = of1:6;

### define selects
select vlan1 = nw_dst: 10.0.1.0 && nw_dst_n_wild: 8
    || nw_src: 10.0.1.0 && nw_dst_n_wild: 8;
...
select vlan36 = nw_dst: 10.0.36.0 && nw_dst_n_wild: 8
    || nw_src: 10.0.36.0 && nw_dst_n_wild: 8;

### define rules
mirror[vlan1]  -> ids1;
mirror[vlan2]  -> ids1;
...
mirror[vlan31] -> ids6;
mirror[vlan36] -> ids6;
```

We initially hoped to use a CPU load reporting tool with a `PROB` distribution rule to dynamically load-balance the traffic. However, the NEC switch's flow table is limited to 3000 entries. The college border sees an average of 330 new flows/second, so we rapidly overflow the NEC's flow table in a matter of seconds. As noted above, we avoided this issue by using a limited set of solely static rules. An alternative solution is to decrease the timeout for dynamic rule entries. State is reclaimed faster allowing the flow table to keep pace with the frequency of new flows. However, if flow entries are removed too quickly, packets will frequently need to be directed to the controller, increasing latency and resulting in poor performance. This remains an open issue in OpenSAFE (Section 7.1).

According to Antonatos et al. [1, 2], drop count (i.e. the number of packets received versus the number of packets examined) is the most useful comparison metric for content matching IDS software. We compare Suricata's reports of the number of packets it processed to the number and of packets the device or PF_RING saw. We combine the counts from all of the OpenSAFE team and summarize the data into 30 minute averages.

### 6.1.2 Results

We ran our test over four days including one weekend. Figure 11 shows the average number of packets/second

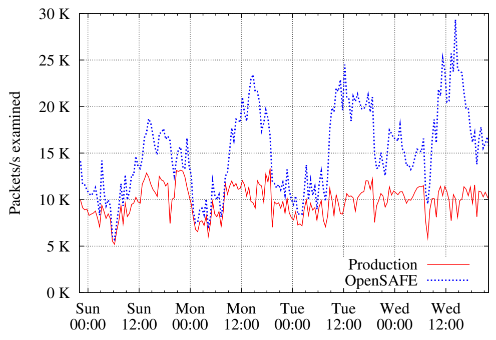


Figure 12: Packets per second examined by the production IDS system software versus the OpenSAFE IDS team.

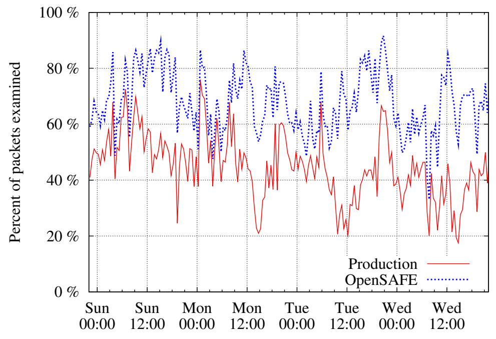


Figure 13: Percentage of packets received that were actually examined by the IDS software (note: the IDS does not examine all types of packets).

received by the existing system compared with the OpenSAFE team. Here we see a typical diurnal traffic pattern. The OpenSAFE team received almost the same amount of traffic as the existing system (96% in total). The major exception occurs on Sunday afternoon when one of the older desktops (an HP xw4300) missed a large subset of traffic. On Monday evening we examined the data, determined the machine was experiencing hardware problems and replaced it with an HP dc7900 (3.0 GHz Core2 E8400) on Tuesday evening. During our replacement we simply updated the ALARMS policy file to direct those subnets to the new machine. We include this event in part to illustrate the flexibility of our system in action.

Figure 12 shows the average number of packets/second examined by the IDS software. Unlike the OpenSAFE team, which is able to closely follow the number of packets received, their is a clear threshold past which the production system cannot keep up. The OpenSAFE team examined 54% more packets in total. In addition, we saw that after only the first day the OpenSAFE team had registered 4,926 more alerts, 30% more than the production system registered during the same time. This demonstrates that the additional parallelism obtained through multiple lower bandwidth IDS machines allows OpenSAFE to scale better.

Figure 13 shows the number of a packets examined as a percentage of all those received. Here we note that although the OpenSAFE team outperforms the existing system, it still does not reach 100%. We believe this is caused by the IDS software only considering certain packets for examination (e.g. only TCP, UDP, ICMP).

## 6.2 Synthetic Loads

To verify that OpenSAFE does not introduce excessive latency, we replayed multiple real-world traces from our previous comparison in a hypothetical on-path setting. We are able to show that increased network load has almost no effect on individual packet latency.

### 6.2.1 Test Setup

We used six Dell PowerEdge R210 servers to generate synthetic traffic loads and measure per-packet latency. The servers were equipped with 2.4 GHz Quad-core Xeon CPUs and two 1 Gbps NICs. Each machine replayed traffic using a user-level Click [15] configuration. We modified packet payloads in our trace data to assign each packet a unique identifier, allowing us to match sent and received packets for calculating the time required to pass each packet through OpenSAFE.

We configured OpenSAFE for a typical on-path monitoring setup. The replayed traces sent by each server represented live traffic input. Flows were directed through a patch cable connecting two switch ports—a setup similar to passing flows through a middlebox but without any latency overhead. Lastly, packets were directed back to their originating server, similar to employing a sink to direct packets back to a production network. Figure 14 shows this setup. The corresponding ALARMS policy fragment used for each server is below.

```
input poweredge2out = of:2;
sink poweredge2in = of:2;
filter to patch2 = of:0;
filter from patch2 = of:1;
poweredge2out -> patch2 -> poweredge2in;
```

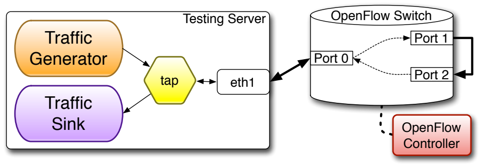


Figure 14: Synthetic traffic generation and measurement setup

Directing packets back to their originating server allowed us to capture both the send and receive time of packets on the same server, which increases our timing accuracy since we avoid clock skew between machines.

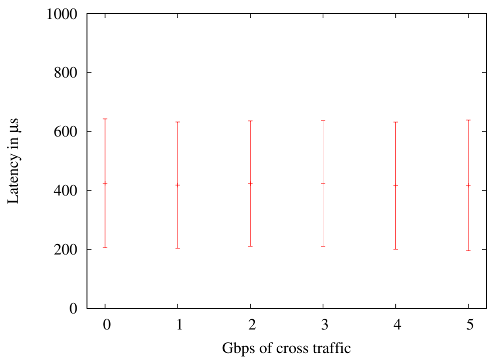


Figure 15: Gigabits per second of cross traffic versus average individual packet latency in microseconds.

We ran `tcpdump` on each server to capture the time packets were sent and received. Using `tcpdump` allows us to measure the overall system latency from kernel to kernel without any userspace queuing effects.

### 6.2.2 Results

We replayed 18.6 million packets from one of the servers, with between zero and five other servers replaying cross traffic. Each test was run for three iterations. Figure 15 shows the average per-packet latency, and standard deviation, for each test. As can be seen from the graph, there is almost no change in the average latency or jitter as cross-traffic volume increases. From this we see that when used with static rules OpenSAFE does not impose any additional performance overhead, making it suitable for both on- and off-path monitoring.

## 6.3 Comparison to Click

We verify the benefit of using a programmable network fabric by comparing OpenSAFE to an in-kernel Click [15] configuration. We show that OpenSAFE's hardware-based forwarding yields lower per-packet latencies than Click's software-based forwarding.

### 6.3.1 Test Setup

We used an in-kernel Click setup to provide a fair comparison of a software-based monitoring setup. The machine was a Dell Precision T5500 workstation with a 2.26 GHz Dual-socket Quad-core Nehalem CPU, 12 GB of RAM, and three 1 Gbps Ethernet ports. We installed Click 1.7.0 in the 2.6.24. 7 Linux kernel running in CentOS 5.3.

Our Click configuration was a combination of our setup from our head-to-head comparison test and synthetic load latency test. Traffic was received from one or more NICs, passed to an IP classifier to divide traffic by subnet, and queued for output on the appropriate NIC. A portion of the Click configuration file is below.

```
### define elements
inEth1 :: FromDevice(eth1, PROMISC true);
classify :: IPClassifier (net 10.0.1.0/24,
                          ...
                          net 10.0.36.0/24,
                          -);
outQueue1 :: Queue(1000);
outEth1 :: ToDevice(eth1);

### define paths
inEth1 -> classify;
classify[0] -> outEth1;
...
classify[36] -> Discard;
outQueue1 -> outEth1;
```

Our comparative OpenSAFE configuration used the same division of traffic by subnets. The ALARMS policy used inputs, sinks, and selects to define similar paths.

The same R210 Dell servers from our previous test were used to replay the same real-world traces and to measure packet send and receive times using `tcpdump`. The subnet divisions and paths in both Click and OpenSAFE were carefully chosen to ensure packets were directed to the same server from which they originated.[4]

### 6.3.2 Results

| | **Click** | **OpenSAFE** |
|---|---|---|
| **1 Gbps** | 837 $\mu$s | 664 $\mu$s |
| **2 Gbps** | 855 $\mu$s | 694 $\mu$s |

Table 2: Average per-packet latency in microseconds using similar configurations in Click and OpenSAFE.

We ran the same two tests with both OpenSAFE and Click: one test with one server generating 1 Gbps traffic and a second test with two servers generating traffic. Table 2 shows the average per-packet latency for each test. We observe that using Click incurs on average over 150 $\mu$s of extra latency per-packet. This extra latency can result in dropped packets on high-speed links.

We attempted to measure Click's performance with more than 2 Gbps of traffic but encountered kernel panics when using more than one NIC. This experience highlights Click's fragility. Furthermore, one NIC typically has at most four ports, so high fanout would require many interconnected Click routers. In comparison, our OpenFlow switch is equipped with 48 x 1 Gbps ports and 2 x 10 Gbps ports. In summary, employing Click yields suboptimal monitoring performance and requires significant additional hardware for high fanout setups.

## 7 Discussion

OpenSAFE is designed for both flexibility and high performance. As our results show, it outperforms an existing monitoring infrastructure and scales with increasing bandwidth. Our major concerns are state exhaus-

[4]A null filter, implemented as a patch cable between two switch ports, was included in the paths in our OpenSAFE setup because OpenFlow does not allow immediately sending a packet back to the same port from which it originated.

tion, matching ability, and flow insertion latency. Further performance improvements and extensibility depends on new capabilities in the programmable switch and resolution of some unique implementation quirks.

## 7.1 Dynamic Rule Latency

Latency to send packets to the controller is an important concern for dynamic rules. The OpenFlow version 0.8.9 specification does not have explicit hashing functions, requiring OpenSAFE to utilize the controller for emulating certain ALARMS rules. Packets destined for an `ANY`, `RR`, `HASH`, or `PROB` rule need to be sent to the controller for the appropriate function to be applied. After computing the destination, the controller needs to send messages to the switch to install flow entries based on the outcome of the function. This results in a relatively long round-trip time to the controller for each new flow that traverses a dynamic rule. In addition, the controller has the potential of being overwhelmed if large numbers of packets are sent for dynamic flow installation.

As we have stated, we avoid these problems by carefully constructing OpenFlow entries that minimize the number of flows that are sent to the controller. Additional study should be done in the area of pre-computing more dynamic distribution rules. It is possible that a particular hash function could be covered by a specific set of static OpenFlow rules; this is obviously not general to all hash functions, but it could be used to improve performance in some cases. Additionally, simple distribution methods that do not require state, like `ANY`, could be added to the OpenFlow specification to reduce controller activity.[5]

Click's dynamic programmability allows hashing and other distribution methods to be calculated directly at the router, without the need to contact another machine. This provides the benefit of faster processing of new flows traversing dynamic distribution rules. However, Click will incur the cost of computing the hash for all subsequent packets in the flow because it does not maintain any per-flow state. A custom Click element could be written to maintain flow tables similar to OpenFlow, but OpenFlow's hardware-based flow lookup will likely still exceed the performance of a software-based element.

## 7.2 The Packet Ordering Problem

When employing OpenSAFE on-path with dynamic rules, the first packet of a new flow is held at the switch while the controller makes a decision on its destination. The packet is not forwarded until flow entries have been installed, effectively delaying all subsequent packets for the flow. We will let $RTT_{controller}$ denote the time from packet arrival to flow entry installation time.

[5]Some dynamic distribution support is expected to be included in the OpenFlow 1.1 specification.

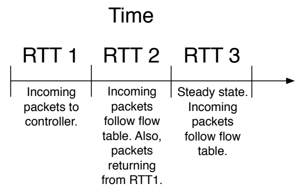

Time
RTT 1
RTT 2
RTT 3
Incoming packets to controller.
Incoming packets follow flow table. Also, packets returning from RTT1.
Steady state. Incoming packets follow flow table.

Figure 16: For a particular flow, packets may arrive out of order.

When using OpenSAFE with dynamic rules in a mirror-based setting, the first packet of a new flow incurs the same $RTT_{controller}$ delay. However, since OpenSAFE is handling a copy of network traffic when used off-path, the subsequent packets of a flow are not delayed. More packets from the flow may arrive at the switch regardless of whether flow entries are installed and the first packet has been forwarded. If packets arrive before the flow entry is installed, these packets will also be sent to the controller. As shown in Figure 16, during the first $RTT_{controller}$ all packets are sent to the controller. During the second $RTT_{controller}$ packets from the first RTT return from the controller *while* new incoming packets are routed per the flow table in the switch. Beginning with the third $RTT_{controller}$, packets will be forwarded directly.

During the second $RTT_{controller}$ packets may be forwarded out of order. Some newly arriving packets are forwarded per the flow table in the switch, while older packets may still be being processed by the controller. This issue is minimized in two ways. The easiest way is to minimize the $RTT_{controller}$ and thereby drastically reducing the number of packets sent to the controller while the first packet is still being processed. The other minimizing factor is that monitoring software already gracefully handles unassisted reordering of packets. In particular, to correctly decode a TCP stream, reassembly is required. If desired a filter could be added that could provide the reassembly.

It is important to note this problem does not occur when employing OpenSAFE on-path or when all flows are precomputed (as $RTT_{controller}$ is effectively zero).

## 8 Related Work

Some existing systems for on- and off-path monitoring were already discussed in Section 2.2. In this section, we discuss works featuring alternative network policy languages and mechanisms for distributing monitoring infrastructure throughout the network.

Casado et al. describe using Ethane [5] switches to enforce middlebox policies and propose a language, *Pol-Eth*, for describing these policies. However, their work on *Pol-Eth* is primarily designed around reachability and the idea that middleboxes can be placed anywhere along the logical path of a flow. In Our work differs in that our policy language, ALARMS, is geared toward directing monitoring traffic at a specific point in the network.

Joseph et al. propose an architecture similar to Ethane in their work on policy-aware switching [14]. However, they do away with OpenFlow's concept of a centralized controller, instead relying on each switch to individually determine the next hop and forward packets immediately. This improves throughput, especially with large quantities of brief flows (where the overhead of contacting the controller is significant), but makes some network management more difficult, as no single entity has a complete view of the network. Additionally, the policy specification language in their work is still centered around deciding appropriate paths for a flow, rather than a higher-level concept of what network monitoring needs to be applied.

A Flow-Based Security Language (FSL) [13] for expressing network policy has been suggested by Hinrichs et al. FSL, a variant of Datalog, allows specification of policies such as access controls, isolation, and communication paths. This specification is flexible and fast, capable of performing lookup and enforcement at high line rates. Again, however, the language is generally focused on end-to-end reachability and path selection, without specific thought to network monitoring.

Flowstream, proposed by Greenhalgh et al [11] considers using OpenFlow as the connecting fabric on a middlebox. Conceptually, Flowstream has similar ideas to OpenSAFE, however, it does not described a high-level policy language.

## 9 Conclusion

Network security monitoring in today's large-scale networks is a difficult task. Rather than attempting to solve all parts of the problem, including how to analyse network traffic, we focused on how to route traffic to monitoring appliances. Current solutions for routing monitored traffic are expensive, difficult to manage, and have problems scaling to high line rates.

OpenSAFE is a cost-effective approach which allows for flexible, fast, and scalable monitoring. It uses widely available OpenFlow-enabled switches to direct traffic through the monitoring infrastructure and scale to line rates. Management is facilitated by ALARMS, a language to enable the arbitrary redirection of network flows for measuring and security purposes. We showed OpenSAFE outperforms an existing, finely-tuned monitoring setup, examining 54% more packets. We also showed OpenSAFE scales to meet the demands of increasing bandwidth and outperforms an in-kernel Click router.

OpenSAFE makes monitoring large scale networks easier than before. It can be combined with other security monitoring improvements to efficiently and effectively monitor high traffic volumes. Most importantly, it lays the groundwork for monitoring infrastructures to meet the changing and growing demands of enterprise networks.

## References

[1] S. Antonatos, K. Anagnostakis, and E. Markatos. Generating realistic workloads for network intrusion detection systems. *ACM SIGSOFT Software Engineering Notes*, 29(1):207–215, 2004.

[2] S. Antonatos, K. Anagnostakis, E. Markatos, and M. Polychronakis. Performance analysis of content matching intrusion detection systems. In *IEEE/IPSJ Symposium on Applications and the Internet (SAINT)*, 2004.

[3] Barnyard2: Snort Output Spool Reader. `http://www.securixlive.com/barnyard2/index.php`.

[4] BASE: Basic Analysis and Security Engine. `http://base.secureideas.net/`.

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *SIGCOMM*, 2007.

[6] G. Cascallana and E. Lizarrondo. Collecting Packet Traces at High Speed. In *IEEE Workshop on Monitoring, Attack Detection and Mitigation (MonAM)*, 2006.

[7] L. Deri, N. S. P. A, V. D. B. Km, and L. L. Figuretta. Improving passive packet capture: Beyond device polling. In *International System Administration and Network Engineering Conference (SANE)*, 2004.

[8] L. Deri and F. Fusco. Exploiting Commodity Multicore Systems for Network Traffic Analysis, 2009.

[9] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *SOSP*, 2009.

[10] GigaMon GigaVue Switch. `http://www.gigamon.com/gigavue-2404.php`.

[11] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy. Flow processing and the rise of commodity network hardware. *ACM SIGCOMM Computer Communication Review*, 39(2):20–26, 2009.

[12] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[13] T. Hinrichs, N. Gude, M. C. andJ ohn Mitchell, and S. Shenker. Expressing and Enforcing Flow-Based Network Security Policies. Technical report, University of Chicago, 2008.

[14] D. A. Joseph, A. Tavakoli, and I. Stoica. A Policy-aware Switching Layer for Data Centers. In *SIGCOMM*, pages 51–62, 2008.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18:263–297, August 2000.

[16] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful intrusion detection for high-speed networks. In *IEEE Symposium on Security and Privacy*, pages 285 – 293, 2002.

[17] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[18] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM SIGOPS Operating Systems Review*, 33(5):217–231, 1999.

[19] nProbe: An Extensible NetFlow/IPFIX Network Probe. `http://www.ntop.org/nProbe.html`.

[20] L. Schaelicke, K. Wheeler, and C. Freeland. Spanids: a scalable network intrusion detection loadbalancer. In *Computing Frontiers (CF)*, pages 315–322, New York, NY, USA, 2005.

[21] S. Snapp, J. Brentano, G. Dias, T. Goan, L. Heberlein, C. Ho, K. N. Levitt, B. Mukherjee, S. Smaha, T. Grance, et al. DIDS (distributed intrusion detection system)-motivation, architecture, and an early prototype. In *National Computer Security Conference*, pages 167–176, 1991.

[22] T. Sproull and J. Lockwood. Distributed Instrusion Prevention in Active and Extensible Networks. *Lecture Notes in Computer Science*, 3912:54, 2007.

[23] Suricata Open Source Intrusion Detection and Prevention Engine. `http://www.openinfosecfoundation.org/`.

[24] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Recent Advances in Intrusion Detection (RAID)*, Berlin, Heidelberg, 2007.

[25] K. Xinidis, I. Charitakis, S. Antonatos, K. Anagnostakis, and E. Markatos. An active splitter architecture for intrusion detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 3(1):31 – 44, 2006.