IRMA via SDN: Intrusion Response and Monitoring Appliance via Software-Defined Networking

Benjamin E. Ujcich, Michael J. Rausch, Klara Nahrstedt, and William H. Sanders
Coordinated Science Laboratory, University of Illinois at Urbana-Champaign
Urbana, Illinois USA
{ujcich2, mjrausc2, klara, whs}@illinois.edu

ABSTRACT

Recent approaches to network intrusion prevention systems (NIPSs) use software-defined networking (SDN) to take advantage of dynamic network reconfigurability and programmability, but issues remain with system component modularity, network size scalability, and response latency. We present IRMA, a novel SDN-based NIPS for enterprise networks, as a network appliance that captures data traffic, checks for intrusions, issues alerts, and responds to alerts by automatically reconfiguring network flows via the SDN control plane. With a composable, modular, and parallelizable service design, we show improved throughput and less than 100 ms average latency between alert detection and response.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management, Network monitoring*

Keywords

Software-defined networking, enterprise network, network intrusion prevention system, NIDS, NIPS, OpenFlow

1. INTRODUCTION

Network intrusion detection systems (NIDSs) and prevention systems (NIPSs) monitor and detect potentially malicious or undesirable network traffic, but implementation of such systems within an enterprise network is often a complex matter [5]. The static nature of these systems often requires human intervention to implement responses, and actionable events may not necessarily be executed quickly enough to mitigate real-time threats [4].

A software-defined networking (SDN) approach provides more flexibility by allowing for scalable monitoring and for dynamic reconfiguration of the network as a response mechanism [3]. However, if we just take a simple extension of existing NIPS by using SDN networks for the underlying data traffic without any close coordination between the NIPS and SDN network, this will be insufficient for solving problems such as NIPS modularity, scalability, latency, and responsiveness.

We propose the Intrusion Response and Monitoring Appliance (IRMA) with its related system architecture for monitoring and response. IRMA represents a novel and flexible NIPS appliance designed in coordination with the SDN-based network architecture, whose design is:

- *modular*, in terms of individual system components to enable reusability and component evolution without requiring redesign of the underlying architecture;
- *scalable*, in terms of the number of monitors and switches beyond a trivial network size; and
- *latency-aware*, to be able to respond quickly to threats.

Our contribution includes the IRMA design, which incorporates 1) a "divide and conquer" hierarchical alert approach that first collects raw alerts from individual zones of a partitioned enterprise network and then analyzes and aggregates them into higher-level alert data collection for scalability reasons; 2) asynchronous communications, parallel processing, and pipelining of modular service components to achieve reduced latency and higher throughput; and 3) service component design to achieve software implementation scalability and modularity.

We validated IRMA both through simulation and through experimentation on the GENI networking testbed [2]. We simulated incoming alert data on the scale of a campus enterprise network to show how the design scales through zoning and parallelization. We created a small SDN topology on GENI to show component modularity and demonstrated that the average latency between detection of new alerts and response through SDN mechanisms is less than 100 ms.

2. ASSUMPTIONS & RELATED WORK

Our network model assumes a medium- to large-scale campus enterprise network, such as one found in a university or corporation. End hosts connect at the edge layer of switches, with the edge layer connected by a core layer whose primary role is to forward traffic. Our threat model assumes that some subset of traffic is malicious (intrusive) or undesired.[1] Malicious end hosts may seek to attack other hosts within the network (e.g., for intelligence gathering or as staging for larger future attacks) or outside of the network (e.g., botnets). To mitigate the ability to evade detection, strategically

[1] Undesired traffic, such as BitTorrent traffic, may not be a security threat but could still negatively disrupt network performance.

placed *monitors* (e.g., Snort) perform three functions: they receive copies of traffic entering edge layer switches, detect potential intrusions, and generate *alerts* on suspicious traffic. Challenges lie in collecting these alerts in a scalable way and implementing network responses dynamically and quickly.[2]

SDN provides new opportunities for monitoring and response through its global view of and control over the network and its programmability. Combining existing NIDS solutions with the responsive actions of SDN in a novel way allows for a new type of intrusion prevention system that is automatic, dynamic, programmable, and agile.

The authors of [3] propose an "active security" methodology for programmatic control over a network through protection of infrastructure, sensing of alerts from multiple monitors, adjustment of the network to respond accordingly, collection of data for further forensic analysis, and coordination of counterattacks.

SnortFlow [6] and SDNIPS [7] propose SDN-based response mechanisms for cloud computing networks, but both designs focus on processing performance rather than system scalability or latency. MalwareMonitor [1] proposes an SDN framework for detecting malware at the border gateway of a campus network by slicing subsets of traffic and dynamically provisioning monitors, though placement of monitoring and response at the gateway limits the ability to provide monitoring and response on intranet traffic and may cause a bottleneck if inspection of traffic is required before it enters or leaves the network's connection to the Internet.

3. IRMA DESIGN

To the best of our knowledge, IRMA is the first network appliance in an SDN enterprise-scale network context to coordinate the monitoring, intrusion detection, and analysis of data traffic with responses assisted by the SDN control plane.

3.1 Design goals and choices

Our primary design goals for IRMA are to consider *modularity* in system components to promote reusability and independent component evolution, *scalability* of the size of the network in terms of switches and monitors, and minimization of *latency* (when possible) between detection of malicious traffic and prevention through responsive actions.

Modularity: We incorporate modularity to logically separate functionality, to further extend the system's capabilities in incremental and evolutional changes, to decouple bindings between key system components, and to pass information through standardized interfaces decoupled from the underlying component's functionality.

Latency: To minimize latency, we made the design choice to let monitoring be of a higher priority than response. IRMA passively monitors mirrored data plane traffic and performs analysis and responses in the background of otherwise normal control plane operations.

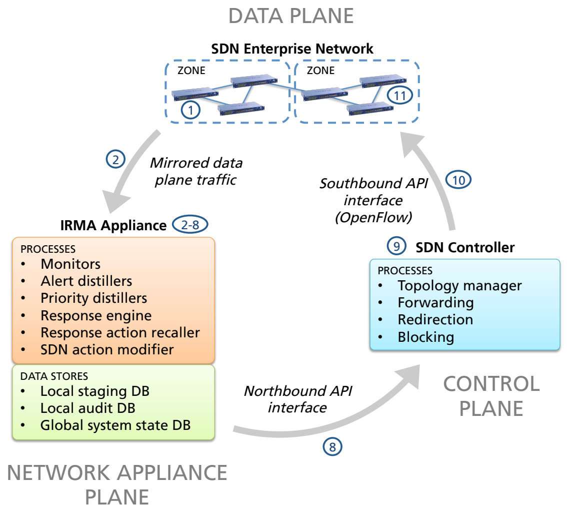


Figure 1: Network architecture for IRMA (numbered items described in Section 3.2).

Modularity, scalability, and latency cannot all be achieved equally, however. We prioritize modularity and scalability, noting that doing so may conflict with minimization of latency in certain instances. For that reason, we attempt to minimize latency when it can reasonably be minimized.

End hosts: Importantly, we note that our design does not require end hosts to be modified for monitoring purposes, as is required in [3, 6, 7]. Monitoring occurs transparently from the perspective of end hosts,[3] and such design allows for use in "open" enterprise networks such as those of universities.

Zoning: We partition the network into locally manageable *zones* to improve monitor alert collection scalability and to "divide and conquer" the alert retrieval and analysis.

3.2 Information flow & functional service components

In the SDN control plane, we define a *flow entry* as a combination of *flow match attributes* (e.g., source and destination IP address, source and destination port[4]), an associated *flow entry action* (e.g., "forward", "block", "redirect"), and other metadata (e.g., incoming port). A switch contains *flow tables*, each comprising multiple flow entries.

Figure 1 shows the high-level information flow among the IRMA appliance, the SDN controller, and the SDN enterprise network. Figure 2 shows the detailed information flow within the IRMA appliance through its components, processes, and data stores. IRMA, as shown in Figure 2, is partitioned into composable service components that re-

[2] A third challenge lies in false positive alerts that inadvertently trigger network responses. We do not address this issue in this paper but note its importance in a production system setting.

[3] Though we do not attempt in this paper to prove the integrity of the alert data received in the IRMA architecture, we note that removal of monitors from the end hosts favors a more secure system design by limiting the potential attack surface and restricting what monitoring components end hosts (users) can access or modify.

[4] At that granularity, flow entries can define end-to-end services.

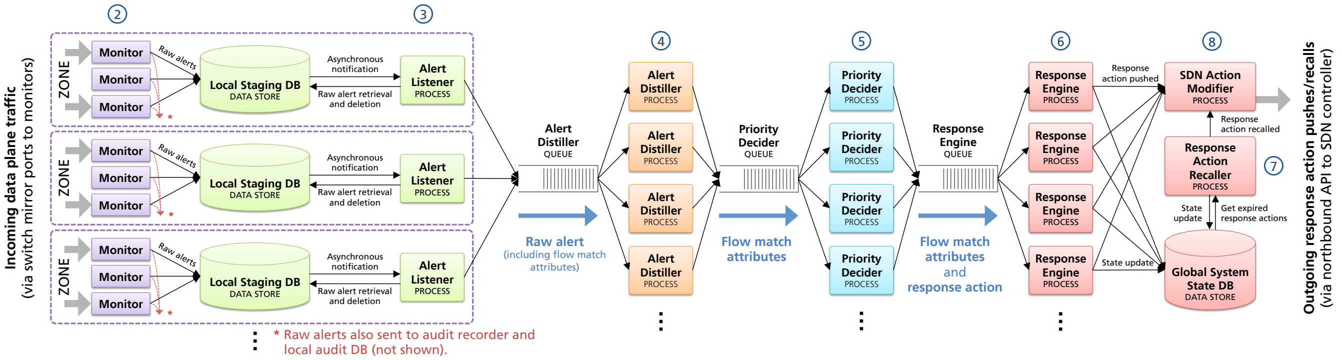


Figure 2: Components, processes, and data stores in the IRMA appliance (numbered items described in Section 3.2).

flect a highly modular design. The modularity means that each functional service entity can be replaced with another implementation of the same functionality.

We outline that information flow, with emphasis on IRMA and its functional service components. The numbered items below correspond to numbered elements in Figures 1 and 2.

1. Incoming network traffic packets in the data plane are matched against switches' flow tables based on the flow entries' flow match attributes, and a corresponding flow entry action is performed (e.g., "forward").
2. Mirrored copies of the packets are passively sent to a *monitor* (e.g., Snort) that detects intrusions and generates raw alerts. *Raw alerts* include packet headers and payload, and they correlate with packets from a specific flow entry defined by the flow match attributes. (Packets that do not generate alerts are discarded.)
3. An *alert listener* asynchronously retrieves (and later removes) raw alerts from the local staging database (see Section 3.3) in the network's respective zone. An *audit recorder* records the alerts to a local audit database.
4. An *alert distiller* filters the raw alert and removes unnecessary details, leaving the flow match attributes.
5. A *priority decider* determines the severity of the threat and sets a *response action* (e.g., "block", "redirect", "isolate") to be sent along with the flow match attributes.
6. A *response engine* references the global system state (see Section 4.1). It 1) inserts or updates global system state entries based on the incoming flow match attributes and response action, and 2) determines if the response action should be sent to the SDN controller (i.e., if the response action should be "enabled" via the SDN action modifier).
7. A *response action recaller* also references the global system state (see Section 4.1). If it determines that response actions need to be recalled, it tells the SDN action modifier to recall them.
8. An *SDN action modifier* interfaces directly with the SDN controller via the northbound API of the SDN architecture (see Section 3.4).

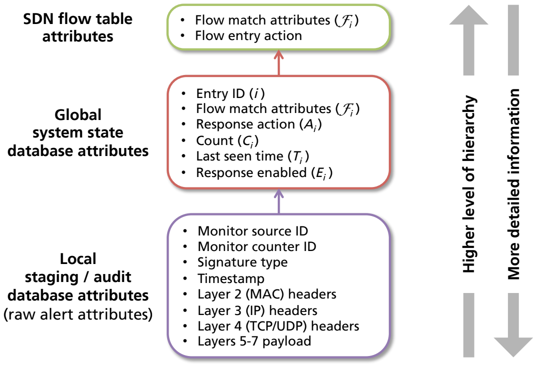

SDN flow table attributes
• Flow match attributes ($\mathcal{F}_i$)
• Flow entry action
Global system state database attributes
• Entry ID (i)
• Flow match attributes ($\mathcal{F}_i$)
• Response action (A_i)
• Count (C_i)
• Last seen time (T_i)
• Response enabled (E_i)
Local staging / audit database attributes (raw alert attributes)
• Monitor source ID
• Monitor counter ID
• Signature type
• Timestamp
• Layer 2 (MAC) headers
• Layer 3 (IP) headers
• Layer 4 (TCP/UDP) headers
• Layers 5-7 payload
Higher level of hierarchy
More detailed information

Figure 3: IRMA hierarchical alert data model.

9. From the northbound API, the SDN controller receives the flow match attributes and response action.
10. The SDN controller pushes changes to the switches via the southbound API using the OpenFlow protocol.
11. In the switches, flow entries matching the flow match attributes have their flow entry action either set to the response action (when pushing a response action) or set to the previous flow entry action (when recalling a response action).

Administrators can set policies both through 1) rule matching in monitors and 2) severity and response action decisions in priority deciders.

3.3 Hierarchical alert data model

A trade-off occurs between the extent of alert data collection and the definition of when and how that alert data should be used. We separate storage into *local* and *global* databases in a hierarchical alert data model as shown in Figure 3.

Processing, storage, and auditing of large numbers of raw alerts is a per-zone, parallelizable process that can be implemented in many local databases and processes at a low level of the hierarchy. Only the significant attributes need to be filtered and sent to a higher level in the hierarchy. *Local staging databases* include detailed information about raw alerts generated from monitors. The databases serve as

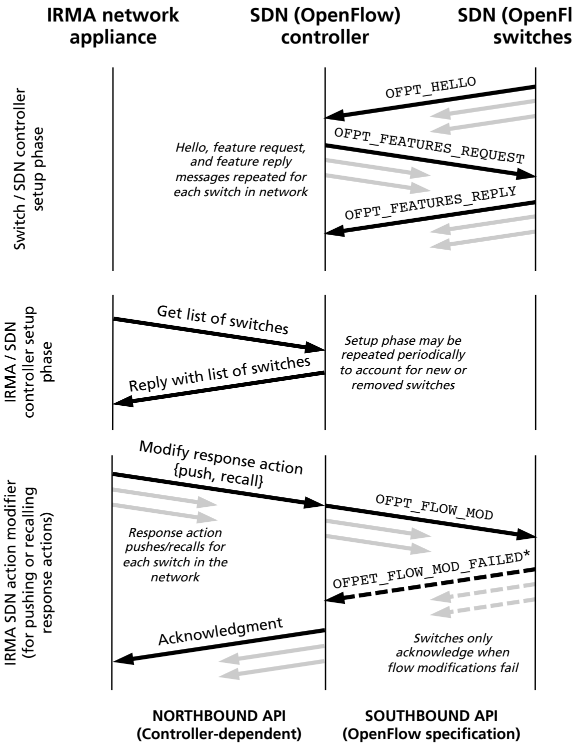

IRMA network appliance
SDN (OpenFlow) controller
SDN (OpenFlow) switches
Switch / SDN controller setup phase
OFPT_HELLO
Hello, feature request, and feature reply messages repeated for each switch in network
OFPT_FEATURES_REQUEST
OFPT_FEATURES_REPLY
IRMA / SDN controller setup phase
Get list of switches
Setup phase may be repeated periodically to account for new or removed switches
Reply with list of switches
IRMA SDN action modifier (for pushing or recalling response actions)
Modify response action {push, recall}
OFPT_FLOW_MOD
Response action pushes/recalls for each switch in the network
OFPET_FLOW_MOD_FAILED*
Acknowledgment
Switches only acknowledge when flow modifications fail
NORTHBOUND API (Controller-dependent)
SOUTHBOUND API (OpenFlow specification)

Figure 4: IRMA control protocol diagram.

local buffers from which alert processing can occur on a per-zone level. *Local audit databases* maintain a copy of the alert data to allow for later retrieval of pertinent events at a detailed level, including full headers, full packet captures, timestamps, location of monitor, and which signature match (monitor rule) instantiated the alert.

System-wide decision making and responses at the higher level of the alert data hierarchy only require summarized alert information. We use a *global system state database* to maintain the current state of the IRMA network appliance, including a statistical summary of all alerts seen across the network and those whose response actions have or have not been sent to the SDN controller. Figure 5 shows an illustration of summarized alert data stored in this database.

Intermediate message passing among components occurs through first-come, first-served queues with multiple servers (processes), as shown in Figure 2.

3.4 Control protocol

Figure 4 shows the protocols between the IRMA network appliance, the SDN controller, and the SDN switches. These components communicate with each other during 3 important phases: the initial setup between the switches and the SDN controller when the switches come online, the setup between IRMA and the SDN controller to query the controller for a list of switches, and IRMA's SDN action modifier communication with the SDN controller when response actions are either pushed or recalled. The IRMA network appliance uses its knowledge of switches in order to push or recall appropriate response actions out to every switch.

Algorithm 1 Basic response engine algorithm.

```
procedure ResponseEngine($\mathcal{F}_i, A_i, S$)
    $T_i \leftarrow$ current time
    if $\mathcal{F}_i \in \mathcal{F}$ then                      ▷ Flow known to system
        $C_i \leftarrow C_i + 1$                      ▷ Increment $s_i$'s counter
    else                                  ▷ Flow unknown to system
        $C_i \leftarrow 1$                                ▷ Initiate counter
        $E_i \leftarrow$ false                       ▷ Do not enable action yet
        $s_i \leftarrow (\mathcal{F}_i, A_i, C_i, T_i, E_i)$                  ▷ Create entry
        $S \leftarrow S \cup \{s_i\}$                       ▷ Add entry to state
    if $C_i \geq C_{Thr}$ and $E_i$ = false then
        SDNActionModifier($\mathcal{F}_i, A_i$, push)
        $E_i \leftarrow$ true                    ▷ Enable response action
```

Algorithm 2 Basic response action recaller algorithm.

```
procedure ResponseActionRecaller($S$)
    for each $s_i$ in $S_E$ do
        if current time $- T_i \geq T_{Thr}$ then
            SDNActionModifier($\mathcal{F}_i, A_i$, recall)
            $S \leftarrow S \setminus \{s_i\}$        ▷ Remove entry from state
```

Table 1: Notation for global system state

	Term	Definition / Values
S	global system state	$S = \{s_1, s_2, \ldots, s_n\}$
s_i	global system state entry	$s_i = (\mathcal{F}_i, A_i, C_i, T_i, E_i)$
$\mathcal{F}_i$	flow match attributes	$\mathcal{F}_i = (IP^i_{src}, IP^i_{dst}, TCP^i_{src}, TCP^i_{dst})$
A_i	response action	$A_i \in \{\text{block, redirect, isolate}, \ldots\}$
C_i	count	$C_i \in \{1, 2, 3, \ldots\}$
T_i	last seen time	$T_i \in \{\textit{valid date and time}\}$
E_i	response action sent to SDN controller ("enabled")	$E_i \in \{\text{true, false}\}$
S_E	entries whose response action has been enabled	$S_E = \{s_i \in S \mid E_i = \text{true}\}$
S_{NE}	entries whose response action has not been enabled	$S_{NE} = \{s_i \in S \mid E_i = \text{false}\}$
C_{Thr}	count threshold	$C_{Thr} \in \{1, 2, 3, \ldots\}$
T_{Thr}	time threshold	$T_{Thr} \in [0, \infty)$

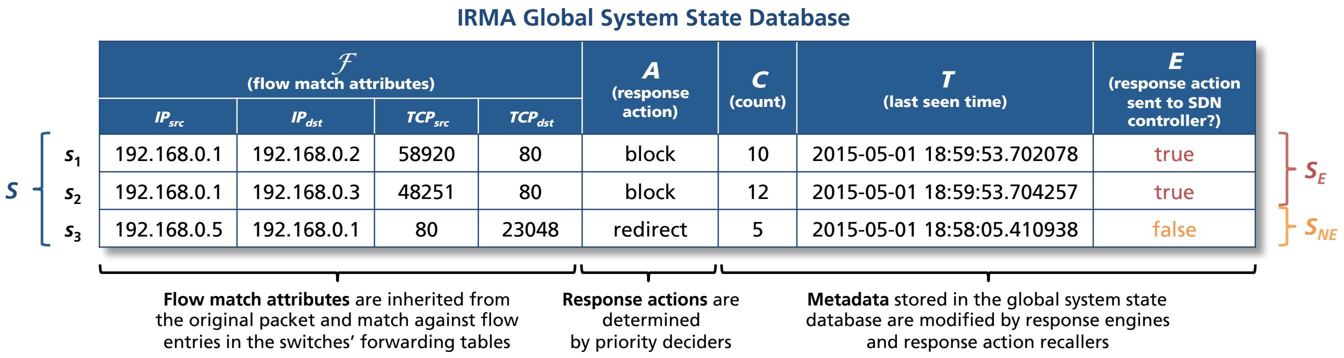
4. IRMA RESPONSE ALGORITHMS

Algorithms 1 and 2 show the algorithms implemented[5] for the response engine and response action recaller functions, respectively. Table 1 defines the notation used throughout the algorithms and this section.

4.1 Procedures

In Algorithm 1, the response engine uses flow match attributes $\mathcal{F}_i$ and a response action A_i to update the global system state S and push response action(s) as necessary. If

[5] The algorithms are just two of many potential methods that can determine when to add and remove responses. We implement basic thresholding algorithms here and note that functional modularity allows for the algorithms to be replaced without requiring modification of any other component of the IRMA architecture.



IRMA Global System State Database

	$\mathcal{F}$ (flow match attributes)				A (response action)	C (count)	T (last seen time)	E (response action sent to SDN controller?)
	IP_{src}	IP_{dst}	TCP_{src}	TCP_{dst}				
s_1	192.168.0.1	192.168.0.2	58920	80	block	10	2015-05-01 18:59:53.702078	true
s_2	192.168.0.1	192.168.0.3	48251	80	block	12	2015-05-01 18:59:53.704257	true
s_3	192.168.0.5	192.168.0.1	80	23048	redirect	5	2015-05-01 18:58:05.410938	false

Flow match attributes are inherited from the original packet and match against flow entries in the switches' forwarding tables

Response actions are determined by priority deciders

Metadata stored in the global system state database are modified by response engines and response action recallers

Figure 5: Examples of entries in the global system state database. (See Table 1 for notation definitions.)

the flow match attributes happen to be already defined in the ith entry s_i in the system state (line 3), then the corresponding entry's count C_i is incremented. Otherwise, a new entry is formed (lines 6–8) and added into the system state (line 9). If the entry has a count that equals or exceeds the count threshold C_{Thr} and the corresponding response action has not yet been sent to the SDN controller (line 10), then the flow match attributes and response action are sent to the SDN action modifier. That action is signified through setting the entry's response-action-enabled flag E_i to true.

In Algorithm 2, the response action recaller uses the global system state S to periodically recall response actions as necessary. If the time since the entry was last updated T_i is greater than the timing threshold T_{Thr}, the response action is recalled, and the entry in the global system state is removed. Algorithm 2 has two benefits: 1) hosts or host processes that have not recently caused malicious activity will eventually have their flows' corresponding response actions recalled, and 2) ongoing attacks will continue to reset T_i, ensuring that the response action remains enabled.

4.2 Time and space complexity

Time and space complexity of both algorithms interfacing with the global system state database depends on implementation. A PostgreSQL implementation uses b-trees by default for indexing, yielding O(log n) worst-case time complexity for searches, inserts, and deletes, and O(n) amount of storage for n number of entries in state S.

5. IRMA VALIDATION

5.1 Implementation

We implemented the IRMA appliance using Python and PostgreSQL. Python allows for creation of multiple processes that communicate through queues. PostgreSQL provides ACID compliance, asynchronous communication, and event-driven triggers. We use the Snort and Suricata IDSs to demonstrate heterogeneous and multiple monitors. Snort and Suricata both write to a common Unified2 logging format, so we used the Barnyard2 interpreter to convert alerts in the Unified2 format to entries in a PostgreSQL database. To implement the SDN network, we used the Floodlight OpenFlow controller and the Open vSwitch (OVS) software switch.

5.2 Experimental setup

Simulation of enterprise scalability: We simulated the zoning scale of an enterprise network to test the hypothesis that partitioning of the network will improve data collection, processing scalability, and throughput. Using a simulator, we set up and varied the number of local staging databases to simulate the zone-partitioning scheme. We generated and queued a fixed number of 25,000 raw alerts systemwide and then released them instantaneously to be processed in order to derive a steady-state throughput. We ran IRMA on a single, multi-core machine while varying the number of zones, alert distillers (AD), priority deciders (PD), and response engines (RE). For metrics, we collected the systemwide steady-state throughput (alerts processed per second).

Experimentation with SDN network: On a smaller scale, we ran IRMA within an SDN network on GENI [2] to test for modularity of system components and to validate the monitoring and response mechanisms in an SDN context. We used 2 OVS switches, 2 types of monitors (Snort and Suricata) connected to the switches' mirrored ports, 1 controller, and 2 end hosts. We set a rule that generates alerts in the monitors upon ICMP ping requests, and we configured IRMA to block the host's traffic in the switch data plane upon receiving these ping alerts. We used 2 zones (2 alert listeners) and one of each component (1 AD, 1 PD, and 1 RE). For metrics, we collected the latency of AL, AD, PD, and RE components as well as the total latency of all 4 components.

5.3 Results and discussion

Figure 6 shows the simulation results of the steady-state throughput when we attempted to process the queued 25,000 alerts. The number of zones[6] varied from 1 (no partitioning) to 32 (high partitioning), and the number of concurrent pro-

[6] For a realistic scale in determining the number of zones, we used the enterprise network of the University of Illinois' College of Engineering as a model, whose IT service manages 100 switches, 17 departments, 20 buildings, and 7,000 networked systems for approximately 13,000 faculty, staff, and students.

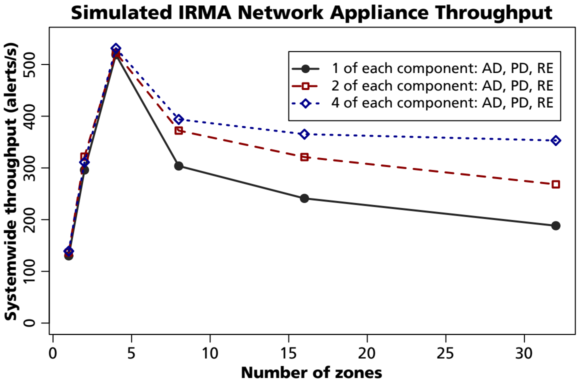


Figure 6: Simulation results of systemwide throughput using IRMA network appliance.

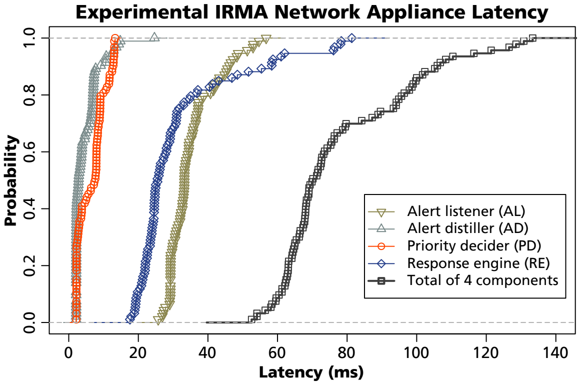


Figure 7: Experimental results showing empirical CDF of latency using IRMA network appliance.

cesses per component varied from 1 (no parallelization) to 4 (high parallelization).

We note a significant performance improvement when the network was partitioned from a single zone to 4 zones: the throughput with 4 partitions is approximately quadruple compared to the throughput with no partitioning. However, we note that the throughput decreases beyond 4 zones. We hypothesize that this is a limitation of implementing the simulation within a single multi-core machine with only 4 cores, *not* a limitation in the architecture design itself, because increasing the number of concurrent processes still increases the throughput *regardless* of the number of zones.

Figure 7 shows the experimental results of the small-scale SDN network on GENI. We repeated the ICMP ping detection experiment 100 times and used 93 of the experiments in our analysis after removing incomplete data and outliers. The mean latency from when new alerts were found to when they were instantiated as SDN responses was 78.206 ms.

Both the alert listener and the response engine showed higher latencies than the alert distiller and priority decider, likely because the alert listener and response engine use slower data stores (i.e., database storage on disk) and more sophisticated logic. The alert listener includes at least 1 round-trip time between the local staging database and the IRMA appliance in its latency, since it receives the raw alert data separately from and after the alert's corresponding asynchronous notification.

6. CONCLUSION

We presented IRMA, a network appliance for SDN-based enterprise networks that provides for dynamic, flexible, and automatic reconfigurability and response mechanisms as a result of monitor-based alerts. Our results from simulation and experimentation demonstrate the efficacy of the appliance, and we hope for its eventual adoption in real enterprise networks.

7. ACKNOWLEDGMENTS

The authors would like to thank Michael Chan at the University of Illinois at Urbana-Champaign's Engineering IT Shared Services for providing network scale statistics, Jenny Applequist for her editorial assistance, and the GENI Project Office (NSF award CNS-0714770). This research was partially supported under a Roy J. Carver Fellowship.

8. REFERENCES

[1] Z. Abaid, M. Rezvani, and S. Jha. MalwareMonitor: An SDN-based framework for securing large networks. In *Proceedings of the 2014 CoNEXT Student Workshop*, pages 40–42, New York, NY, USA, 2014. ACM.

[2] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, 2014. Special issue on Future Internet Testbeds - Part I.

[3] R. Hand, M. Ton, and E. Keller. Active security. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 17:1–17:7, New York, NY, USA, 2013. ACM.

[4] S. Kumar. Survey of current network intrusion detection techniques. Technical report, Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis, Missouri, December 2007.

[5] J. Sherry and S. Ratnasamy. A survey of enterprise middlebox deployments. Technical Report UCB/EECS-2012-24, EECS Department, University of California, Berkeley, February 2012.

[6] T. Xing, D. Huang, L. Xu, C.-J. Chung, and P. Khatkar. SnortFlow: A OpenFlow-based intrusion prevention system in cloud environment. In *Proceedings of the 2013 Second GENI Research and Educational Experiment Workshop (GREE)*, pages 89–92, Mar. 2013.

[7] T. Xing, Z. Xiong, D. Huang, and D. Medhi. SDNIPS: Enabling software-defined networking based intrusion prevention system in clouds. In *Proceedings of the 2014 10th International Conference on Network and Service Management (CNSM)*, pages 308–311, Nov. 2014.