2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)

An Ontology-Driven Approach to Automating the Process of Integrating Security Software Systems

Chadni Islam*
CREST Centre
Australia
School of Computer Science
University of Adelaide
Adelaide, Australia
chadni.islam@adelaide.edu.au

Muhammad Ali Babar
CREST Centre
Australia
School of Computer Science
University of Adelaide
Adelaide, Australia
ali.babar@adelaide.edu.au

Surya Nepal
Data61
CSIRO
Sydney, Australia
surya.nepal@data61.csiro.au

***Abstract*— A wide variety of security software systems need to be integrated into a Security Orchestration Platform (SecOrP) to streamline the processes of defending against and responding to cybersecurity attacks. Lack of interpretability and interoperability among security systems are considered the key challenges to fully leverage the potential of the collective capabilities of different security systems. The processes of integrating security systems are repetitive, time-consuming and error-prone; these processes are carried out manually by human experts or using ad-hoc methods. To help automate security systems integration processes, we propose an Ontology-driven approach for Security OrchestrAtion Platform (OnSOAP). The developed solution enables interpretability, and interoperability among security systems, which may exist in operational silos. We demonstrate OnSOAP's support for automated integration of security systems to execute the incident response process with three security systems (*Splunk, Limacharlie,* and *Snort)* for a Distributed Denial of Service (DDoS) attack. The evaluation results show that OnSOAP enables SecOrP to interpret the input and output of different security systems, produce error-free integration details, and make security systems interoperable with each other to automate and accelerate an incident response process.**

Keywords*— Security orchestration, incident response process, automated integration process, ontology, security system

I. Introduction

Organizations use multi-vendors heterogeneous security software systems [1] to protect their Information and Communication Technology (ICT) Infrastructures and Business Applications (BA) [1-4]. Most of the security systems, such as Intrusion Detection System (IDS), Security Information and Event Management (SIEM), and Endpoint Detection and Response (EDR) operate independently in silos to carry out their respective functions that generate security-related data. Then a human expert needs to manually [2] integrate and analyze the security events data generated from security-centric processes (i.e., *capture traffic data*, *detect an incident*, and *analyze the logs*) of different software to perform an incident response [1, 5-7]. Organizations use orchestration platforms to integrate and orchestrate security systems to automate the activities performed by these systems [2, 3, 8, 9]. A Security Orchestration Platform (SecOrP) aims to minimize the dependency on human experts for a streamlined incident response process [9, 10]. However, most of the existing SecOrP cannot automatically adapt to organizational systems' operational processes such as the installation of new software, deployment of new servers, and rolling out new access control policies [2, 3, 8]. Security staff needs to manually integrate different security systems with a SecOrP and map their activities into an incident response process [5, 6, 11].

Human-centric intervention is required in the existing SecOrP because security systems are not designed to interoperate with each other [12]; for example, an IDS cannot automatically send the output alert to a SIEM. The messages generated by a security system are also not semantically interpretable [7, 13]; for example, an IDS generates an alert in its proprietary format, and such an alert contains different features of an attack. A SIEM cannot ingest and interpret the meaning of those alerts unless the alerts' definitions are explicitly defined for a particular type of SIEM [7, 14]. Before deploying a SecOrP, an organization assesses the existing security system to identify the configuration details, such as dependency among different activities, process flow within a security system, input and output data format and runtime environment [5, 7]. Based on such an assessment, a SecOrP is designed, and different security systems are integrated using plugins or APIs [2, 5, 6, 15].

Human intervention can be minimized by providing SecOrP a formal specification of the security data format, configuration, and structural specifications of security systems to automate the process of integrating different security systems. A SecOrP can use such formal specification to continuously integrate and invoke security systems based on the activities in the incident response process. Ontologies can be used to provide the required formal specifications to support semantic integration [16-19]. In the context of SecOrP, a semantic integration means one security system can understand the semantics of the input and output of another security system. A SecOrP can semantically interpret the activities when the formalization incorporates semantic integration of heterogeneous security systems.

Several studies have developed ontologies to formalize heterogeneous threat intelligence information for cybersecurity system [16-20], including ontologies to help stakeholders to deal with semantic conflict that arises while integrating multiple security systems [21]. These approaches focus on providing an effective way for information sharing and exchanging among cybersecurity communities, and stakeholders. None of these approaches provides support for sharing information between different security systems and SecOrP. Security systems (e.g., IDS, SIEM, or EDR) are software-intensive systems that can be integrated and

* Also with Data61, CSIRO, Australia

[1] Security Software Systems, or Security Systems, are software intensive systems for detecting, preventing, and recovering from cyber security attacks. In this paper, we write security software systems as security systems.

978-1-7281-3393-5/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSSP.2019.00017

interoperate at the software level based on data integration. Hence, ontological approaches can be leveraged to automate the process of integrating and interoperating heterogeneous security systems in a SecOrP.

Our Solution: We have developed an **On**tology-driven approach for **S**ecurity **O**rchestr**A**tion **P**latform, ***OnSOAP*** to automate the process of integration of security systems. At the core of *OnSOAP* is an ontological model that involves both high-level and fine-grained classes for different security systems, their capabilities, and the activities of an incident response process. The developed ontology provides a formal specification of the core concepts of a SecOrP, i.e., security system, their capabilities, and the activities of the incident response processes. Based on the ontology, *OnSOAP* automatically annotates a set of security systems, their capabilities and an incident response process at a much finer-grained scale. We have also designed a set of *queries*, *rules,* and *constraints* for the orchestration process that enables *OnSOAP* to extract the required information from the ontology and invoke the required functionalities of a security system. *OnSOAP* ensures *error-free* and *automated integration* of different security systems in a SecOrP.

We have developed and evaluated *OnSOAP* with a robust incident response application. As a use case scenario, we have investigated the incident response plan for a *Distributed Denial of Service attack* (*DDoS*) with three different security systems. This can be extended for any other use case scenario and security systems. By composing a set of simple rules and defining structured queries for the orchestration process, we have shown that the developed *OnSOAP* can automatically select and invoke an appropriate set of functionalities from the available security systems. We have also shown that the developed *OnSOAP* can provide automation support to the process of integrating and interoperating security systems.

Contributions: Followings are the main contributions:

- Design and implementation of an ontology-driven approach, *OnSOAP,* that supports the process of automated integration and interpretation of a variety of security systems (Sec III, IV, V, VI).
- Demonstration of the use of *OnSOAP* to automate the execution of an incident response process for a DDoS attack with three security systems (Sec VII).
- Evaluation of *OnSOAP's* ability to automatically generate accurate configuration details for enabling security systems to *interoperate* and remove *operational silos* for a *DDoS* attack (Sec VIII).

II. Preliminaries

A. Challenges in Automated Integration

Given the diverse nature of security systems, the integration process of SecOrP has several challenges [22]. It is not possible to know all the requirements of an organization at the design and installation phases of a SecOrP [17]. A SecOrP needs to control the flow of the activities performed by different systems. A security staff modifies the activities based on a system's availability and preferences. For example, a security staff may change an activity in the incident response process from analyzing alert log to correlating alert log after installation of a new IDS in a network router. However, there is a lack of a systematic way of automating the process of integrating security systems [3, 21].

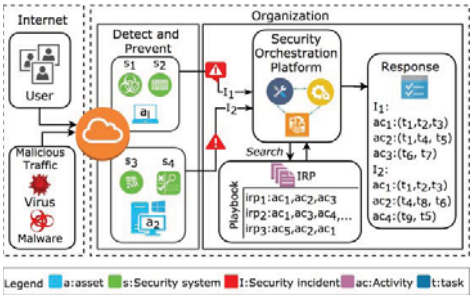


Fig. 1. An example of execution of an incident response process in a security orchestration platform.

A security expert adds or removes security systems or changes their configuration and deployment strategies; for example, a security staff may change the preferred format of alerts log file based on the SIEM system being used. A security staff manually maps the newly integrated security system's functionalities with the activities of an incident response process and vice versa. **Fig. 1** depicts a scenario of an organization that has a SecOrP. However, a SecOrP shown in **Fig. 1** cannot automatically perform the mentioned changes due to the heterogeneity and isolation nature of most security systems, which lacks interoperability and interpretation of the generated messages.

B. Problem Formulation

Consider a scenario where an application of an organizations' website is being overused with a superfluous request. Detecting the scenario as a malicious one, an IDS generates alerts that consider the behaviors as a *DDoS* attack. Upon gathering such alerts, a SecOrP orchestrates and automates the incident response process to prevent and recover from a *DDoS* attack. For an illustration purpose, we assume the actions automated by a SecOrP are *detecting incidents*, *collecting alerts* and *system logs*, *identifying the affected systems*, and *generating an incident report*. Let us assume that an organization has a set of security systems S, that are integrated into a SecOrP, where $S = \{s_1, s_2, s_3, \dots, s_i, \dots, s_n\}$. An example of s_i can be *Snort*, *Bro*, *Splunk*, or *Limacharlie*. Each security system s_i performs a set of activities, $AC = \{ac_1, ac_2, \dots, ac_j \dots, ac_m\}$ to protect against potential security attacks.

Definition: 2.1. (Activity). An Activity is an action performed by a security system or a human expert to detect, prevent, remediate, recover or response security incidents. Examples of the activities include *detecting security incidents*, *investigating threats,* and *analyzing threats behavior*. A SecOrP has a playbook that has a list of Incident Response Plan (IRP), $IRP = \{irp_1, irp_2, \dots, irp_k, \dots, irp_p\}$, where, $irp_k = \{ ac_1, ac_2 \dots, ac_j\}$ is a sequence of activities that need to execute in response to a security incident. Playbook contains the rules associated with the execution of an IRP.

Definition: 2.2. (Security Incident). A security incident is an unwanted or unexpected event/events that have a significant probability of compromising the security of an organization's assets. The examples of security incidents include security threats, breaches, attacks and so forth. For example, for a security incident, $I\,(DDoS\ Attack)$, the response process includes activities *DetectIncident*, *CollectLog*, *AnalyzeLog*, *GenerateReport*, and so on.

Fig. 1 shows a scenario of auto-execution of an incident response plan irp_k to respond to a security incident. In the scenario, a security system s_i detects suspicious traffic *on* asset a_l and generates an alert that is considered as a security incident *I* (i.e., an alert "*DDoS attack*").

Definition: 2.3 (Asset): An asset is any system, data, resources, hardware, or software that an organization wants to protect. The examples of assets include databases, servers, hosts, applications, and websites.

A SecOrP gathers the incident details and searches for an appropriate IRP in the playbook. For example, for the incident, *I*, shown in **Fig. 1**, a SecOrP finds the best match IRP irp_1, which has a list of activities $irp_1 = \{ac_1, ac_2, ac_3\}$. The SecOrP then searches for a security system that can execute those activities and finds the artifacts required to automate the execution of the activities. Then SecOrP performs a set of tasks, *T* for each activity, where $T = \{t_1, t_2, \ldots, t_q, \ldots, t_r\}$.

Definition: 2.4 (Artifact): We consider the alerts and logs generated by different security systems as the *artifacts* of a SecOrP, which deals with different structured, semi-structured and unstructured data that come in various formats and languages from different security systems.

Definition: 2.5 (Task). A task is an action that a SecOrP performs to automate the execution of the activities in an IRP. For example, the execution of an activity *DetectMaliciousTraffic* requires a SecOrP to perform three tasks: *searchDetectionSystem* (t_1): looking for an available security system that purports to detect intrusion, *selectDetectionSystem* (t_2): if multiple systems are available, selecting one, and *invokeDetectionSystem* (t_3): invoking a security system to run in detection mode.

We assume that a SecOrP selects a security system s_i *(Snort*), which can scan the asset a_l (*endpoint*) to detect incidents. A SecOrP invokes s_i to run in intrusion detection mode. These tasks are performed by a SecOrP to automate the execution of the activity ac_j (*DetectIncident*). Finally, s_i scans a_l to detect suspicious traffic. We refer to this as the execution of the activity ac_j that is performed by a security system s_i. We consider the combined activities performed by security systems and the tasks performed by SecOrP to automate the execution of activities as an automated integration process.

C. Motivation

Our research aims at automating the process of integrating different security systems into a SecOrP. The integration process is a combination of interpretation, selection, formulation, and invocation. A SecOrP performs different types of tasks to manage different aspects of threat defense and incident response. For example, a SecOrP connects different activities of different security systems to *remove operational silos*. We refer to this type of tasks as an *integration* task. A SecOrP also orchestrates the flow of data and activities to make security systems *interoperable* and enables a machine to machine automation by providing machine *interpretability semantics*. We consider these types of tasks as *orchestration*, and *automation* of security activities. The process of automated integration of security systems for the execution of an IRP depends on the combination of these three tasks.

Auto-Integration Process = task (Integration, Orchestration, Automation) = Interpretation + Selection + Formulation + Invocation + Execution

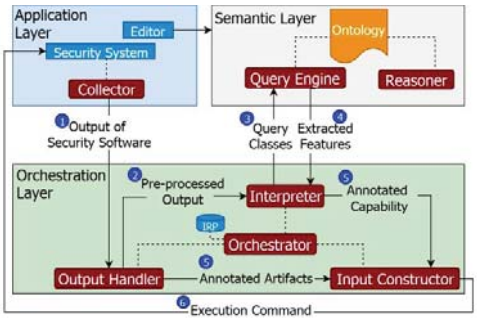


Fig. 2. A high-level overview of *OnSOAP*

In this work, we only focus on developing suitable support for automating the process of integrating security systems in a SecOrP for a seamless incident response without a human effort on performing activities.

III. The Proposed Solution

Fig. 2 provides an overview of the developed *OnSOAP*. It comprises three layers: Application layer, Semantic Layer, and Orchestration Layer. It is composed of core concepts of SecOrP's and provides (i) a unified capability of SecOrP, which can be achieved through a number of security systems, (ii) the semantics of different security systems' generated data, and (iii) the rules to avoid conflict and invalid integration of security systems. It can make different security systems interoperable so that an output from one can be used as input to another without a human effort.

Application Layer (Sec V): The application layer provides the fine-grained information about security systems, running processes and the current state of an organization. In this layer, a collector collects the artifacts generated by different security systems and assets. These artifacts are forwarded to the lower layers, which then extract the features (e.g., alerts types) from a log. Fig. 2 shows that the raw data are passed through the orchestration layer that applies pre-processing rules to map the raw events onto the classes of the developed ontology (e.g., maps the alerts log to the IDS that generated it).

Semantic Layer (***Sec IV***): The semantic layer provides the semantic details about the input, output, and activities performed by security systems to the orchestration layer. It supports the integration process of security systems of *OnSOAP* through which (i) security systems' capabilities (functional and non-functional features) are captured, (ii) the capabilities required by IRPs are identified, (iii) the related artifacts and data maintained among different security systems are identified, and (iv) the configuration details of security systems are retrieved. It also stores abstract knowledge about the tasks performed by SecOrP.

Orchestration Layer (Sec VI): The orchestration layer is responsible for invoking appropriate tasks for automating the process of integration of several security systems based on the activities of an IRP. The output produced by different security systems and the annotated output of the semantic layer are passed to the orchestration layer. The orchestrator of the orchestration layer controls the integration process.

IV. Semantic Layer

Interpretation of security systems capabilities requires formalization of various concepts (i.e., activities, inputs and outputs). The semantic layer uses an ontology to represent the

domain knowledge of SecOrP through a set of concepts and their relationships, as described in the following section. It leverages the ontologies' capability to represent multi-sourced data [17, 20] taking into account the semantic integration process among heterogeneous data produced by different security software systems.

A. Ontological Model

For building the ontological model for *OnSOAP*, we have cataloged the existing security systems based on their key features, different data types and runtime environments. Our ontology engineering focuses on leveraging the existing and widely adopted ontologies [21]. We used the ontological model to formalize the semantics of some of the key security systems' capabilities (e.g., *intrusion detection* and the *command to invoke a system*), artifacts (i.e., windows log, Syslog) or context data (affected assets) and the activities of IRPs. The proposed ontology model consists of the following classes shown in Fig. 3.

Security system class (*SecuritySystem*) represents all types of security systems. These systems are designed to protect the assets based on an organization's requirements. We modeled different features of security systems under this ontology class. The *activity class* (*Activity*) formalizes the actions of the IRP. The activities are presented with respect to a system's required functional features. The *capability class* (*Capability*) is used to capture functional and non-functional features of an individual security system. This class is used to instantiate the underlying ontology model with response to a security incident to execute the selected activities. It also formalizes the types of data with which security systems deal with.

As shown in Fig. 3, the *Activity* class has a relation *requireUseOf* with *SecuritySystem* class, where the *SecuritySystem* class has a relation hasCapability with the class *Capability*. The relationship between classes is defined as the object property. Here the hasCapability is the object property of the *SecuritySystem* class. These types of relationships can also be presented as the triplet in RDF or XML, for example, (*SecuritySystem*, hasCapability, *Capability*) *w*here *SecuritySystem* is the domain of the object property and *Capability* is the range.

1) Security System

Different types of security system have been categorized under the abstract class *SecuritySystem*, which has different subclasses with some subclasses based on its functionality as shown in Fig. 3. For example, both *Bro* and *Snort* in Fig. 3 are considered as the subclass of *IDS* due to their extensive use for intrusion detection. Though *Snort* can be used for both intrusion detection and packet sniffing, most organization use *Snort* to detect intrusion which is the main functionality of *Snort*. We express the necessary conditions for a security system to execute an activity through the semantics of *Capability* class. The *SecuritySystem* class can be extended to incorporate new security system with new types of behaviors. When *OnSOAP* intends to integrate a security system, it creates an instance of a security system by instantiating the associated properties required for security systems integration. For example, an instance of *SecuritySystem, SnortInstance* represents the instance of *Snort* class, which indicates a *Snort* system is integrated into *OnSOAP*.

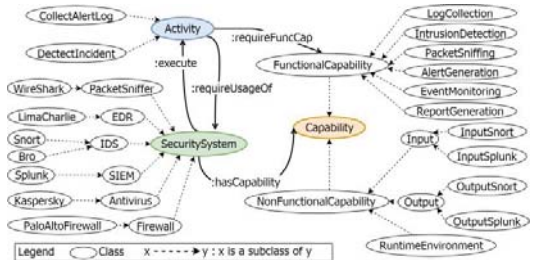


Fig. 3. Part of our ontology, the dashed arrow represents subclass and the solid line represents the relationship among classes

To categorize a security system under a particular subclass, a security system must satisfy the capability associated with that subclass. We impose different types of restrictions for creating an instance of a class that must satisfy the relations with other classes. For example, as shown in Fig. 3, the *SecuritySystem* class has a relation *execute* with the *Activity* class. An instance of a *SecuritySystem* class must execute an activity. We describe more details on the rules for imposing different constraints and restriction in section IV.B.

2) Activity

We categorize each activity of the IRP under the *Activity* class. This class is instantiated in response to an incident *I* (i.e., *DDOS* attack). An instance ac_i (i. e., *DetectMaliciousTraffic*) of an activity class ac_j, $ac_j \in AC$ represents the execution of ac_jin response to the incident *I*. *DetectMaliciousTraffic* and *CollectSnortAlertLog* are the instances of the subclass of *Activity* class. Fig. 3 shows the details about the subclasses and their relationships. Execution of each activity further depends on the availability of security systems and preferences of an organization's security requirements.

A SecOrP executes an orchestration routine to call individual security system to execute an activity. Execution of activities generate artifacts, i.e., system log and alert log. Artifacts are also required before executing the activities. For each activity, a SecOrP performs a set of task T (i.e., *select security system* and *invoke security system*) to collect and manage artifacts, automate, and track execution of the activities performed by security systems. The execution of these tasks generates further events, i.e., *system found* and *endpoint protection system running successfully* through which *OnSOAP* keeps track of the tasks and the activities being executed by a security system.

3) Capability

We define the capability of a security system under the class *Capability* with two subclasses: *FunctionalCapability* and *NonFunctionalCapability*. We consider each security system can be represented regarding their functional and non-functional capability.

Definition 4.1. (Functional Capability). The functional capability is the capability of a security system to perform activities to achieve security objectives. We denote a set $\mathbb{F}$ as the functional capability of a security system. Each security system, s_i, can have a list of functions denoted by δ_{s_i} where $\delta_{s_i} \subset \mathbb{F}$. For example, *IntrusionDetection* and *LogManagement* support the activities *DetectIncident* and *MangeLog* respectively.

Definition 4.2. (Non-Functional Capability). The non-functional capability is the ability of a security system to support the quality requirement while providing the functional features. Examples of a security system's non-functional capabilities include command syntax, input parameter format, and data type. For instance, alerts generated by *Snort* in

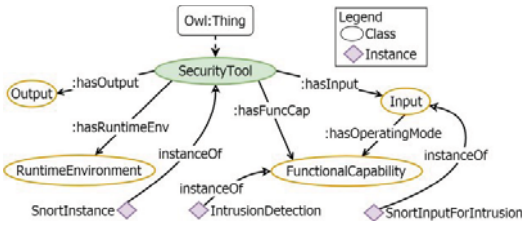


Fig. 4. The relationship between the *SecuritySystem* class and subclass of *Capability* class.

various file formats (i.e., *CSV* or *binary*) are considered as the non-functional capability.

Each security system has a different data structure, preferred configuration, generated workflow and a way to share information with security experts [2-4, 8]. We define these kinds of the knowledge required by OnSOAP to run and maintain a security system under non-functional capability. The Non-Functional capability class further has three subclasses: Input, Output, and RuntimeEnvironment. The proposed ontology model required the Input class for executing an activity. The *SnortInputForIntrusio*n has the information to run Snort on *IntrusionDetection* mode. The Input class has the configuration details for Snort in intrusion detection mode. We have designed the Output class to capture different types of outputs that are generated by security systems after the execution of an activity. Inputs and outputs of each security system vary with the activities executed and depend on the runtime environment. Whenever a new security system is installed in an organization, a security expert can populate the ontological model by defining the capabilities of the installed security system. More details on the relationship between the classes are shown in Fig. 4. Fig. 5 shows some of the instances of *SecuritySystem*, FunctionalCapability, Input and Output class.

B. Ontological Reasoning

Our proposed *OnSOAP* has a *Reasoner* that uses rule-based reasoning to derive semantic correlation among the activities, security system and capabilities. We have defined various rules within the ontology to provide inferred information and some constraints for error-free integration. These rules help *OnSOAP* to avoid ambiguity while creating an instance of classes. Based on the rule-based reasoning of the ontology, the *Reasoner* provides the inferred information.

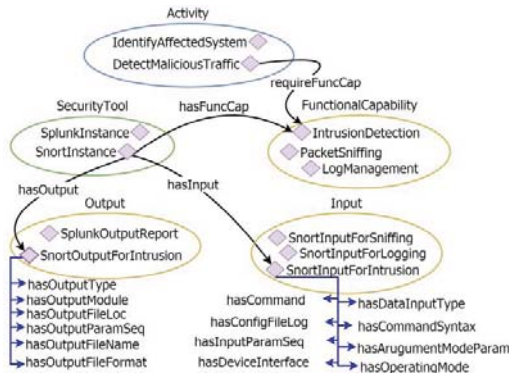


Fig. 5. Instances of classes of the ontological model and their relationship with other instances. Blue lines represent the data property of the instance of the class Input and Output.

Fig. 5 shows that *SnortInstance* hasFuncCap *IntrusionDetection*, and hasOutput *OutputForIntrusion*. *OutputForIntrusion* hasOutputType *Alert*. Based on the reasoning, the *Reasoner* infers the relation *SnortInstance* hasOutputType *Alert*. Using the *Reasoner*, we can also derive the following information: *"If Snort generates an alert in intrusion detection mode, then the execution of activity DetectIncident by Snort must generate alert while seeing malicious traffic*".

We provide the examples of some rules (Rule 1 to 9) that are defined in our ontology. The rules enable *OnSOAP* to satisfy reliable automation of the activities. These rules enable us to express the conditions about the occurrence or non-occurrence of auto execution of the activities, the creation of instances, and tracking and managing the activities of IRP. Each security system must have at least one functional capability to execute an activity (Rule 1). For example, *Snort* must have a functional capability *Intrusion Detection* that can execute an activity *detection intrusion*. Auto-execution of an activity requires at least one system with the functional capability required to execute the activity (Rule 2 & 3). The input and output of security systems need to be explicitly defined to be integrated into a SecOrP. For example, every security system must have an input command (Rule 4) so that *OnSOAP* can automatically invoke a security system to execute different types of activities. Most of the output produced by different security systems need to have an *output file location* from where a SecOrP reads the file to interpret the output (Rule 5). For example, if a *Snort* runs in *intrusion detection* mode and generates output, then the output type must be an *Alert* (Rule 6).

Rule 1: SecuritySoftware hasFuncCap min 1 FunctionalCapability
Rule 2: Activity requireUseOf exactly 1 SecuritySoftware
Rule 3: Activity requireFuncCap some FunctionalCapability
Rule 4: Input hasCommandSyntax some xsd:string
Rule 5: Output hasOutputFileLoc some xsd:string
Rule 6: OutputForIntrusion hasOutputType only Alert

We have also defined the rules for each security system class. Rule 7 to 9 are the examples of such rules, where Rule 7 and 8 are dedicated to security system SIEM and Rule 9 is for IDS. We impose the criteria to categorize a security system under a subclass of *SecuritySystem* class. For example, using Rule 7 and 8, we restrict the creation of SIEM system instance. Any instance of a SIEM system must satisfy Rule 7 and 8. *OnSOAP* executes the activities sequentially; that causes the input to execute ac_{j+1} is relying on the output of ac_j. As a result, a security system that is selected to execute ac_{j+1} must have access to the output of a security system that executes ac_j. For example, if *Splunk* needs to analyze the alert log produced by *Snort*, it must have access to the output file of *Snort*. As per our rules, if *Splunk* input type is equivalent to *Snort* output, then *Splunk* input file location must need to be the same as the *Snort* output file location. Similarly, *OnSOAP* needs to have the authorization to invoke and stop every security system that is integrated into its platform.

Rule 7	*SIEM hasFuncCap min 3 FunctionalCapability*
Rule 8	*SIEM hasFuncCap some (EventManagement or Event Monitoring or LogAnalysis or LogCollection or LogManagement)*
Rule 9	*IDS hasFuncCap only (IntrusionDetection or PacketLogging or PacketSniffing)*

TABLE I. DIFFERENT TYPES OF QUERY

Query Type	Query Details
Q_1	Query to identify the functional capability required to execute an activity ac_j.
Q_2	Query to search for a security software system s_i that has functional capability $\mathbb{F}$.
Q_3	Query to retrieve the non-functional capability required by s_i to execute functional capability F_a

C. Querying the Ontology

The semantic layer deploys a *Query engine* to extract the necessary features from the ontology. The *Query engine* is responsible for communicating with our ontology. It queries the ontology based on the requirement of the *Interpreter*. We designed a set of queries for *OnSOAP* to retrieve the necessary information from the ontology. The queries have three different structures depending on the required information as shown in TABLE I. The *Interpreter* of the orchestration layer invokes the appropriate query to select the security systems (Q_1), functional capabilities required by the activities (Q_2), and capabilities of a security system (Q_3). The query Q_3 has three different structures: query to extract the input details, query to extract the output details and query about the runtime environment. If an incident response process has an activity *DetectIncident*, and execution of that activity requires the capability *IntrusionDetection*, then the *Query engines* queries the ontology for a security system that has the capability *IntrusionDetection*. Assuming an instance of *Snort* is available, the query returns *SnortInstance*.

V. APPLICATION LAYER

The application layer employs a *collector* to collect the raw events data from organizations' environment. The *Collector* collects system log, network packets, alerts, security incidents, configuration changes, or commands from experts. Given the logs collected from different security systems, the *Collector* pre-processes the raw events data before sending to the *Output handler* in orchestration layer. It annotates the output with the context details, such as *types of log* (i.e., Syslog, server log, event logs, and message log), the *location* from where it has collected the logs (i.e., directory), the *environment* (OS, endpoint, sensor, server) and timestamp.

The application layer also provides support to integrate the knowledge in the ontology through an ontology *Editor*. The ontology *Editor* is deployed in the application layer to create, update and modify the ontology classes. At this stage, we consider a security staff will update the ontology's details. *OnSOAP* uses the reasoner in every step to check the consistency of the operation that is performed via the editor. For example, if the *editor* attempts to create an instance of *Snort* with functional capability *MonitorFile*, the reasoner considers the ontology inconsistent because based on rule 9 *Snort* has three capabilities that do not include *MonitorFile*.

VI. ORCHESTRATION LAYER

This section describes the process of automating the integration of security systems performed in orchestration layer in four stages: (i) interpretation of incident, (ii) identification of activities and functional capabilities required to respond to an incident (iii) selection of security systems, and (iv) formulation of command to invoke a security system.

A. Interpretation of the Incident

The *output handler* of orchestration layer receives the output of a security system from the *Collector*. Upon receiving the *alert* event, it sends the output (i.e., *alert log* produced by s_i *Snort)* to the *Interpreter* to interpret the incident type *I*.

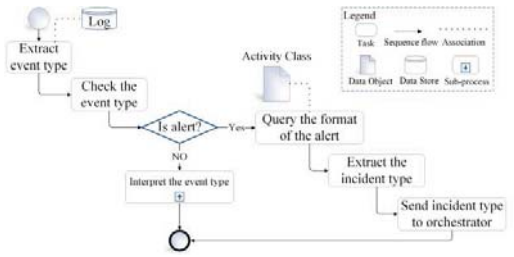
Fig. 6. Example of sub processes of integration process for (a) interpreting the incident and (b) identification of capabilities to automatically response against the incident.Fig. 6. Example of sub processes of integration process for (a) interpreting the incident and (b) identification of capabilities to automatically response against the incident.(a**)** shows an example process of interpreting the incident type from the alert log generated by a security system. To semantically interpret the incident, the interpreter first identifies a security system s_i that generates the alert. It invokes the *Query engine* to get the output format of s_i. Upon receiving the output, it semantically annotates the incident type *I* among the list of features in the alert. The *Interpreter* returns the *Output handler* the annotated alert (e.g., alert type, description, and source IP) and sends the incident *I* to the *Orchestrator* for taking the response action.

B. Identification of Capability to Response an Incident

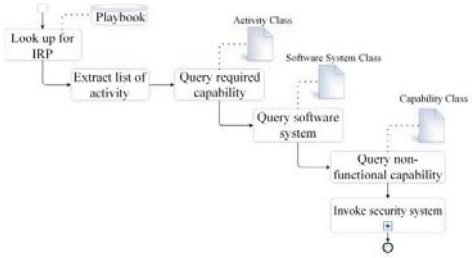
Upon receiving the incident I, the orchestrator looks for the possible IRP in the incident response playbook. Assuming $irp_k = \{ac_1, ac_2, ac_3\}$ is the IRP for incident *I*, the *Orchestrator* extracts the list of activities from irp_k and invokes the *interpreter* to identify the functional capability required to perform an incident response against an incident. For each activity, the *Interprete*r invokes the query engine to run query Q_1 (TABLE I) that returns the functional capability required to execute an activity and send to the *Orchestrator*.

C. Selection of Security System

According to the proposed scenario (II.B), the auto-execution of the IRP requires *OnSOAP* to identify a security system s_i with the functional capability F_a to execute the activity. The query Q_2 is used to identify a security system



(a) Interpreting the Incident



(b) Identification of capability to response an incident

Fig. 6. Example of sub processes of integration process for (a) interpreting the incident and (b) identification of capabilities to automatically response against the incident.

with functional capability F_a, where $F_a \in \mathbb{F}$ (**Fig. 6** (b)). Each security system has relation with the functional capability class. The *Interprete*r queries the ontology to find the list of security systems required to execute irp_k.

D. Formulation of Command to Invoke a Security System

The *Input constructor* requires the knowledge to formulate the instruction to run s_i in F_a mode. The formulation of commands requires the *Orchestrator* to invoke the interpreter to interpret the input features of a security system. The ontological model has that information as a data property in the *Input* class (IV.A.3). The *Interpreter* invokes query Q_3 of *Query engine* to extract the details of the input command. The annotated inputs are passed to the *Input constructor,* which ultimately generates the command details to invoke a particular security system (**Fig. 6** (b)).

In some cases, the execution of specific activities requires the output from previous activities. The *Input constructor* needs to generate the script to invoke a security system and sends an integration command to execute the activities sequentially. To execute a sequence of activities, the *Input constructor* needs the annotated output of the previous activities. The *Orchestrator* invokes the *Output handler* to send the annotated output to the *Input constructo*r to execute this activity. *OnSOAP* also has a set of rules to manage the interoperability among security systems. The *Orchestrator* controls the flow of the operation in these cases and invokes appropriate modules, rules, and query to automate the execution of irp_k. The orchestration layer enables direct interpretation of the input and output of security systems in order to make them interoperable.

This whole process of integration security system is automated by *OnSOAP* using the proposed ontology, a set of rules and queries.

VII. Experimental Design and Setup

In this section, we describe the experimental setup used for demonstrating and evaluating *OnSOAP*'s ability to automate the process of integrating various security systems. We have developed the required components based on **Fig. 2**.

A. Gathering Input Data for OnSOAP

We gathered a list of activities and identified the functional capabilities required to execute these activities. TABLE II shows the part of the activities gathered. The activities were extracted from the IRP for different types of attacks. We extracted the IRP from the website ServiceNow[3] for different incidents (Table III). We slightly modified the IRP to match with the capabilities of security systems we were using. We used the IRP for *DDoS* attack shown in Table III for our experiment. The purpose of this experiment is to demonstrate *OnSOAP* ability to automate the integration process that enables auto-execution of the IRP.

TABLE II. FUNCTIONAL CAPABILITY MAPPED WITH ACTIVITY

Activity		**Functional capability**	
ac_1	Detect incident	Intrusion detection	F_1
ac_2	Collect alert log	Log collection	F_2
ac_3	Identify affected system	Alert Analysis	F_3
ac_4	Generate incident report	Report generation	F_4
ac_5	Sniff network packet	Packet sniffing	F_5
ac_6	Log network packet	Packet Logging	F_6
ac_7	Isolate affected node	Node Isolation	F_7
ac_8	Kill malicious process	Process killing	F_8
ac_9	Generate report	Report Generation	F_9
ac_{10}	Monitor Event	Event monitoring	F_{10}
ac_{11}	Manage log	Log management	F_{11}
ac_{12}	Investigate alert	Alert analysis	F_3
ac_{13}	Generate alerts	Intrusion detection	F_1
ac_{14}	Scan endpoint	Intrusion detection	F_1
ac_{15}	Remove malware	Process killing	F_8

TABLE III. USE CASE SCENARIO WITH IRP

Incident type		**Incident response plan (IRP)**
I_1	DDoS attack	$ac_2, ac_{12}, ac_5, ac_3, ac_{13}, ac_{10}, ac_7, ac_9$
I_2	Malicious process	$ac_{10}, ac_{11}, ac_3, ac_9, ac_8$
I_3	Malware	$ac_{14}, ac_{12}, ac_{15}, ac_9$

B. Application Environment Setup

To set up the application environment, we chose three different types of security systems, *Snort* as IDS, *Splunk* as SIEM and *Limacharlie* as EDR that have the capabilities to execute the activities of TABLE II. Among these security systems, *Snort* and *Limacharlie* are open source security systems where *Splunk* is a commercial one. We used the free trial of *Splunk* enterprise version due to its wide range of functional capability. We mapped the functional capability of security systems with TABLE II, which gave $\delta_{\text{Splunk}} = \{F_2, F_3, F_4, F_9, F_{10}, F_{11}\}$, $\delta_{\text{Snort}} = \{F_1, F_5, F_6\}$ and $\delta_{\text{Limacharlie}} = \{F_2, F_7, F_8, F_{12}\}$. Both *Limacharlie* and *Splunk* can perform other activities that are not listed here. We used a centralized directory to collect alerts logs produced by *Snort*, the event and process log sent by a *Limacharlie* sensor to a Limacharlie cloud and gathered the reports generated by *Splunk*. We defined variables to preserve the information following from application layer to orchestration layer, for example, variable *SystemFrom* to store security systems that produced the output and variable *filePath* to store the location of the output file (Sec V). We installed *Snort* and *Limacharlie* sensor application on the local host. *Limachairlie* cloud server and *Splunk* server application were deployed on a virtual machine. We also defined the detection and response rules for *Limacharlie* and *Splunk*.

C. Development of the Ontological Model

We developed the ontology of *OnSOAP* using protégé[4], an ontology *editor*. We defined the details of the abovementioned three security systems, the capabilities of these systems and activities of incident response plan in the ontology. We used RDF/XML serialization to store the ontology. We followed the similar approaches discussed in IV.A. We used the OntOlogy Pitfall Scanner[5] to evaluate the functional and structural dimension, conciseness, and completeness of the ontology. We maintained the consistency of the ontology while developing the concepts and populating the ontology. We defined the reasoning rules discussed in IV.B and used Pellet reasoner to remove the ambiguity. Whenever any class instance does not satisfy the rules imposed on the class, the ontology becomes inconsistent. The reasoner generated error notification if it finds any inconsistency within the classes of the ontology. A violation of the restriction imposed on the classes also caused the reasoner to give error notification.

[3] https://www.servicenow.com
[4] https://protege.stanford.edu/products.php#desktop-protege
[5] http://oops.linkeddata.es/

Thus, the reasoner ensured the consistency of our developed ontological model. We developed the *Query engine* that can access the ontology through SPARQL queries. We loaded the ontology meta model into Apache Jena Fuseki server[6], an open source SPARQL server.

D. Development of the Orchestration Layer

Language. We have used both Java and Python for developing *OnSOAP*. *Splunk* has an SDK that allows interfacing with Java-based programs. It has a REST API to send a command to a security system through HTTP requests. *Limacharlie* has a Python API available on GitHub[7]. It is simply an interface to a REST API service. Whilst the API provides an easy way to interface with *Limacharlie* service, it is not tool agnostic. Because the *Limacharlie* API has been implemented in Python with *Limacharlie* package, the *Limacharlie* class in Java is not able to send commands to endpoints by itself. It needs to execute Python scripts and passes the required arguments to the appropriate command to an agent on an endpoint.

Module. We defined a task associated with each activity as discussed in II.A. We developed rules for the *Interpreter*, *Output handler* and the *Input constructor* to analyze the events log, interpret output and issuing commands to invoke security system. We developed the processes described in VI.A, VI.B, VI.C, and VI.D for automating the process of integration. For example, to interpret the output of security systems, the developed *Interpreter* extracted the information about a security system's capabilities and returned hash maps, which contain the information about the output file location and the action that generated the output.

E. Baseline Approaches

We used two baseline approaches to perform comparative analysis: manual integration process (BL1) and API based integration process with a static interpreter (BL2).

In BL1, the response process depends on a human. The security staff performed the tasks using the security system. For example, during the monitoring process, if security staff found alerts, they looked in the playbook for the response actions or used previous experience to investigate the alerts.

In BL2 solution, we developed a set of APIs between security systems to automate the sequence of activities. This process needed pre-developed APIs in both directions for each security system, i.e., API to send data from *Splunk* to *Limacharlie*, from *Snort* to *Splunk* and *Limacharlie* to *Splunk* as well. The goal of these APIs was to capture the essential expected capabilities of each security system that allowed the implementation of the interface by any that kinds of security systems. We developed a static interpreter along with output handler to automate response for *DDoS* attack of Table III.

VIII. Evaluation

We have evaluated *OnSOAP* using the following Research Questions (RQ).

A. RQ1: How Effective is OnSOAP's Process of Automating the Integration of Security Systems?

Motivation. *OnSOAP* leverages the semantic interpretation of both activities and security systems capabilities to integrate security systems through the orchestration process. However, the integration process still works if the APIs are designed and developed between security systems. Thus, we would like to investigate whether the combination of semantic integration and orchestration results in a better-automated process. The RQ1 answers how effective is *OnSOAP* in making security systems interoperable where one system can directly use the output of another system as its input. It also investigates whether or not the system interprets the output of a security system to formulate the input of another security system with different capabilities.

Approaches. We used network traffic with malicious behavior that has DDoS attacks. The Snort security system generated alerts for the security incident I_1 *DDoS* attack, which triggers the whole integration process. We compared the process of *OnSOAP* with the two baseline approaches. We monitored the actions performed by human experts for each activities for the IRP of I_1 . We also considered different number of APIs needed for the automation of the process using the BL2. Finally, we investigated the developed *OnSOAP* to execute the same list of activities.

Results. Considering all three approaches use the same IRP, for each alert the security staff first searched for the alert types and then looked for the possible list of actions and based on the list performed the actions. For the standard case, the staff used their previous experience to select security system to perform each activity. For example to perform the activities ac_2 and ac_{12} that are *collect* and *investigate alerts logs* the security staff collected the alerts generated by *Snort*, and then uploaded those alerts to *Splunk*. For this, the experts manually needed to log in to *Splunk* and upload the alert logs and then defined the rules to investigate alerts. In case the same types of alerts were seen next, security staff needed to go through the same manual process again. The expert also needed to read the reports generated by *Splunk* and then sent the commands to *Limacharlie* to isolate the affected nodes. For similar types of alerts, the manual process requires staff to repeat the same sequence of actions, that require huge amount of man-hours and also delays the response process.

For BL2, APIs were available to perform the same sequence of actions. The APIs used a shared directory where Snort stored the alerts and *Splunk* collected those alerts. A separate collector needed to be designed for each security system to automate the process and interpret which security system produced the alerts. The BL2 also required to define the APIs for each function before the execution of an IRP. For example, execution of I_1 required to design eight different APIs for each activity, even though the number of the security systems were three. If the same set of operations was needed to be performed in a different host and server, the APIs needed to be redesigned to work with that host and server.

With *OnSOAP*, once *Snort* generated the alerts, the *Interpreter* automatically identified that the incident as *DDoS* attack and triggered the incident response process. We discuss the effectiveness of *OnSOAP* regarding the challenges mentioned above. *OnSOAP* chose the security system based on their functional capabilities. It gathered the list of the security systems that can perform the activities in the IRP for incident I_1. The *input command syntax* has the command needed by the security systems to run the security software. The *commandSyntax* has the sequence of the parameters to formulate the commands. Retrieving this information, *OnSOAP* successfully generated the scripts to run *Snort*, *Splunk* and *Limacharlie* in a different mode. Thus, the *OnSOAP* successfully invoked each security system to

[6] https://jena.apache.org/documentation/fuseki2/

[7] https://github.com/refractionpoint/python-limacharlie/

perform the sequence of activities without the intervention of human experts. Our developed system executed all the activities where it generated three different types of command to run *Splunk* in three different modes. We state that the program removes the operational silos by enabling the security system to execute different activities.

Considering, the developed *OnSOAP* selected Splunk to execute the activity ac_2 , it generated the configuration commands for *Splunk* to collect *Snort* log. It queried the *Ontology* to identify the *input command* to invoke *Splunk* to gather *Snort* alert logs as well as the location of the output of the *Snort* alert log. Based on the results, *OnSOAP* generated the scripts to run *Splunk* to collect *Snort* output. Finally, it invoked Splunk to analyze logs (ac_3) which were based on the rules defined in *Splunk* to identify the affected assets or cluster malicious assets. *OnSOAP* interpreted the output of a security system to formulate the input of another security system with different capabilities. The similar process is performed when *OnSOAP* select *Splunk* to identify the malicious nodes and *Limacharlie* to isolate those nodes (ac_3). *OnSOAP* extracted the node details from *Splunk* and generated the commands details for *Limacharlie*. Through the same process, the system auto-executed a sequence of activities where the output of one system has been used by another system. The system is able to interpret the data generated by the security system and also interpret the actions performed by a security system. With BL_2, the same API that was used to send the *Snort* log to *Splunk* was not applicable here. To interpret and extract the message of *Splunk* BL_2 needed to define the rules in the interpreter and then developed the API to send the message to *Limacharlie.*

From the evaluation of *OnSOAP* in comparison to traditional approaches, we state our proposed approach for automating the process of integrating security system successfully executes the IRP. The same process can be used to automate a different number of IRP with different types of security systems where with BL2 new rules and APIs needs to be defined for different IRPs.

B. RQ2: How Efficient is OnSOAP for Practical Use?

Motivation. During the ontology development process, *OnSOAP* needs a domain expert to design the classes of security system capabilities. Developing the ontology requires a substantial understanding of security systems being used. Another time-consuming process is designing the incident response plan and orchestration process. If *OnSOAP* cannot alleviate challenges related to the manual integration process, and substantial efforts needed to build the ontology, the security experts may not be willing to use it in practice.

Approaches. We introduce new incidents (I_2 and I_3) that includes a list of activities and investigate the results of *OnSOAP*. Among these activities, some activities were executed during incident I_1, and some are new. For new activities, we compare the amount of effort requires for *OnSOAP* in terms of the information an expert needed to include in the ontology and the efforts of BL2 in terms of the number of new APIs needed to design.

Results. For all the three incidents, the security staff required a substantial understanding of security systems. Also for both BL2 and *OnSOAP,* the IRP and the automation processes needed to be defined. For BL2, the security staff needed to select security systems, developed the APIs accordingly, and then define the rules to automate the process. For *OnSOAP*, the staff only need to define the capabilities of security systems to execute those activities. For the already defined capabilities, we do not need to do any further actions. The same integration process of extracting incidents, selecting security systems, interpreting output and formulating input works for executing the IRP for I_2 and I_3. The evaluation shows that little effort or changes are required in *OnSOAP* with the change in IRP.

C. Threat to Validity

Our work is focused on security systems that are used by the SOC of an organization. Gathering the security system capabilities was challenging, as the information is not freely accessible. The developed system is limited to security tools that are widely used, freely available and open-source. Currently, the evaluation of the proposed method is carried out in a University laboratory environment, which also limits the scope of the experiment. The proposed evaluation approach does not provide any quantitative measurement, which we plan to carry out in the future work.

IX. Related Work

We have carried out a multi-vocal literature review to gather the state of the art on security orchestration [3]. Many research efforts are dedicated to using security ontologies to formalize several concepts of cyber security domain [14, 17-19]. Such concepts include security mechanisms, security objectives, attacks, alerts, threats, vulnerabilities, and countermeasures. Most studies are focused on modelling various types of attacks using ontologies [14, 23]. Where others use ontologies to detect and prevent attacks [18, 24]. None of the abovementioned ontologies can be used by SecOrP to make security systems work together as these ontologies do not have the capabilities of security systems to streamline incident response process.

Several studies [16, 18, 19] have developed ontologies to formalize heterogenous threat intelligence information for cybersecurity systems. These studies focus on providing an effective way for information sharing and exchanging among cybersecurity communities. One study [16] has developed a Unified Cyber Security (UCO) ontology by combining and mapping widely used ontologies, i.e., STIX and CybOX. UCO provides a standard semantic representation of cybersecurity systems for information integration and cyber situational awareness. Although UCO has enriched the threat vocabulary, it does not provide any support for interoperability among heterogeneous security systems. The work also lacks ontologies that SecOrP can use to interpret the activities performed by different security systems to make them work together as a requirement for auto-execution of IRP.

Recently, a set of ontologies have been developed to enable tool-as-service (TSPACE) for a cloud-based platform [21]. Ontologies are used to select, and provision tool based on stakeholder's requirements and semantically integrate the artifacts of the tools. A platform provides stakeholder a set of tools by using the ontology proposed by the authors; further stakeholders use those tools to perform the required activities. Ontologies help stakeholders to deal with semantic conflict that arises while integrating multiple tools in the same platform. The ontologies of TSPACE do not provide any support to automate the execution of activities or make the tools interoperable. Thus, the ontologies are not applicable to SecOrP. The features of the tools, needed to make them interoperable and remove the operational silos, are not captured in the ontologies of TSPACE. Unlike this generic

work, our proposed *OnSOAP* provides the features of security systems that address the issues of interoperability, and interpretability and an automated process of integrating security systems, which can execute their respective activities for working together without human interventions.

We have not found any other work that addresses the interpretability and interoperability issues, which need to be addressed to integrate, automate and orchestrate security systems' activities for seamless incident response process. Our work supports the interoperability and interpretability issues by mapping the capabilities of the security systems with the activities of the IRP and providing the orchestration process to automate the execution of the IRP. Our ontology formalizes the security system in detail by capturing the functional and non-functional features of security systems. *OnSOAP* can interpret the capability of security systems and generate the command required to invoke a tool. Hence, it is uniquely positioned to address the challenges. To the best of our knowledge, this is the first work that has developed an ontology to automate the process of integrating security systems in SecOrP. The automation is achieved by enabling interpretability, interoperability, and removing operation silos of the multivendor heterogeneous security systems.

X. Conclusion and Future Work

We propose an ontology-driven approach to automating the process of integrating different security systems in a security orchestration platform. By formalizing the concepts of security systems, we aim to support automation in the integration process of security systems that further enables *interoperability* among different security systems. We provide an ontological model that characterizes all the concepts and relationships of SecOrP required for the integration process. We assert that *OnSOAP* can *interpret* the semantics of the output shared by different security systems and formulate the input required by security systems.

Further, *OnSOAP* glues security systems to execute the incident response process automatically. We have demonstrated the viability of the proposed approach by developing and using a proof-of-concept system. The results show that *OnSOAP* can (i) interpret the output of security systems, (ii) invoke a security system to analyze the data of one security system and (iii) automate the integration process to execute an incident response plan. We assert that our approach can minimize the challenges characterized by the manual integration process and effectively automate the integration process of different security systems. The findings from developing and evaluating our approach enable us to believe that *OnSOAP* can be easily integrated with an existing SecOrP for the large-scale realization of security orchestration and automation in an organization's SOC.

Our work requires a detailed definition of the features of different security systems and incident response plan. Without suitable definitions of a security system's functional and non-functional capabilities, *OnSOAP* will not be able to perform the abovementioned tasks. Our future work aims at performing a large-scale evaluation of the proposed system. In addition, we aim at designing a probabilistic learning model to automate the integration process that can use the ontological model and the existing security systems' configurations to generate the APIs when an exact match is missing.

Acknowledgment

The work in partially supported by Data61/CSIRO, Australia.

References

[1] McAfee, "*MacAfee Orchestration Platform*," [Online]. Available on: https://www.mcafee.com/au/solutions/orchestration.aspx Accessed on: [October 10, 2017.]

[2] S. Luo and M. B. Salem, "Orchestration of software-defined security services," in *2016 IEEE International Conference on Communications Workshops, ICC 2016*, Kuala Lumpur, Malaysia, 2016.

[3] C. Islam, M. A. Babar, and S. Nepal, "A multivocal review of security orchestration," *ACM Comput. Surv., In Press,* 2019.

[4] E. Feitosa, E. Souto, and D. H. Sadok, "An orchestration approach for unwanted Internet traffic identification," *Computer Networks,* vol. 56, pp. 2805-2831, 2012.

[5] Komand, "*Security automation best practice*," [Online]. Available on: https://www.komand.com/, Accessed on: [October 21, 2017.]

[6] FireEye, "*Security Orchestrator: Simplify threat response through integration and automation*," 2017, [Online]. Available on: https://www.fireeye.com/solutions/security-orchestrator.html, Accessed on: [October 31, 2017, 2017.]

[7] Demisto, "*Security orchestration and automation*," 2017, [Online]. Available on: https://www.demisto.com/wp-content/uploads/2017/04/MH-Demisto-Security-Automation-WP.pdf Accessed on: [October 11, 2017.]

[8] T. Koyama, B. Hu, Y. Nagafuchi, E. Shioji, and K. Takahashi, "Security orchestration with a global threat intelligence platform," *NTT Technical Review,* vol. 13, 2015.

[9] F. Dario, "Security orchestration & Automation: parsign the Options," vol. 2017, ed: darkreading, 2017.

[10] O. Rochford and P. E. Proctor, "Innovation Tech Insight for Security Operations, Analytics and Reporting," Gartner 2015.

[11] Intel Security, "*Automating the Threat Defence Lifecycle*," 2017, [Online]. Available on: https://www.mcafee.com/au/solutions/orchestration.aspx, Accessed on: [October 20, 2017.]

[12] FireEye, "*Security Orchestration In Action: Integrate – Automate – Manage*," [Online]. Available on: https://www.fireeye.com/solutions/security-orchestrator.html, Accessed on, 2018.]

[13] T. Kenaza and M. Aiash, "Toward an Efficient Ontology-Based Event Correlation in SIEM," in *Procedia Computer Science*, 2016, pp. 139-146.

[14] T. Kenaza and M. Aiash, "Toward an Efficient Ontology-Based Event Correlation in SIEM," in *The 7th International Conference on Ambient Systems, Networks and Technologies*, Madrid, Spain 2016.

[15] Demisto, "Collaborative and Automated Security Operations - A comprehensive Incident Management Platform."

[16] Z. Syed, A. Padia, T. Finin, M. L. Mathews, and A. Joshi, "UCO: A Unified Cybersecurity Ontology," in *AAAI Workshop: Artificial Intelligence for Cyber Security*, 2016.

[17] A. Evesti and E. Ovaska, "Ontology-based security adaptation at run-time," in *Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference on*, 2010, pp. 204-212.

[18] E. Casey, G. Back, and S. Barnum, "Leveraging CybOX™ to standardize representation and exchange of digital forensic information," *Digital Investigation,* vol. 12, pp. S102-S110, 2015.

[19] S. Barnum, "Standardizing cyber threat intelligence information with the Structured Threat Information eXpression (STIX)," *MITRE Corporation,* vol. 11, pp. 1-22, 2012.

[20] A. Kim, J. Luo, and M. Kang, "Security ontology for annotating resources," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, 2005.

[21] M. A. Chauhan, M. A. Babar, and Q. Z. Sheng, "A Reference Architecture for provisioning of Tools as a Service: Meta-model, Ontologies and Design Elements," *Future Generation Computer Systems,* vol. 69, pp. 41-65, 4// 2017.

[22] F. Asplund and M. Törngren, "The discourse on tool integration beyond technology, a literature survey," *Journal of Systems and Software,* vol. 106, pp. 117-131, 2015.

[23] M. Bist, A. P. S. Panwar, and V. Kumar, "An agent based architecture using ontology for intrusion detection system," in *2016 2nd International Conference on Next Generation Computing Technologies (NGCT)*, 2016, pp. 579-587.

[24] A. Razzaq, Z. Anwar, H. F. Ahmad, K. Latif, and F. Munir, "Ontology for attack detection: An intelligent approach to web application security," *Computers & Security,* vol. 45, pp. 124-146, 2014.