Automatic Synthesis of Network Security Services: A First Step

Lei Xu, Yangyong Zhang, Phakpoom Chinprutthiwong, Guofei Gu
SUCCESS Lab, Texas A&M University
Email: lexu@paloaltonetworks.com, yangyongzhang.io@gmail.com, cpx0rpc@gmail.com, guofei@cse.tamu.edu

***Abstract*—In the network security life cycle, security needs are initialized by network operators and typically documented in natural languages, and later implemented and deployed in developed/acquired security appliances, typically written in a programming language by third-party developers. However, oftentimes, those security appliances/programs may not quite match the urgent and fast-evolving security needs since the whole developing/deployment procedure is very time-consuming.**

In this paper, we propose a novel framework, AUTOSEC**, to aid network operators in building up or rapid prototyping operational network security services directly from high-level service needs as automatically as possible.** AUTOSEC **helps bridge the huge gap from human intents in natural language descriptions to the deliverable network security services. More specifically,** AUTOSEC **utilizes Natural Language Processing (NLP) techniques to infer security intents from natural language descriptions, and then performs Interactive Synthesis to assist users to validate and refine parsed intents if necessary.** AUTOSEC **further leverages Software-Defined Networking (SDN) and Network Function Virtualization (NFV) techniques to automatically compose and instantiate security services in terms of refined security intents. In the evaluation, we demonstrate the early success of** AUTOSEC **with security policy descriptions collected from various data sources including research papers, appliance descriptions, real-world security standards, and human-written policies.**

I. INTRODUCTION

As a common practice in the typical network security life cycle, network security operators always start with understanding threats and defining required security needs in specifications or policies (typically written in natural languages), before they design, implement, or acquire network security service programs/appliances (typically written in a programming language and often by third-party developers) [1]. However, often times, the procedure is time-consuming and can hardly meet the urgent and fast-evolving security needs, since tedious low-level manual efforts are typically required for implementations or configurations of network appliances from various vendors [2]. For example, in order to enforce a security intent to counter against network reconnaissance attacks within an operational network (e.g., "If we detect a password guessing attack, we need to redirect the attack traffic to a honeypot."), a network operator needs non-trivial efforts to "program" password guessing detection function and enforcing the correct response, i.e., redirect the attack traffic to a honeypot for further analysis.

To ease the pain, some prior development frameworks and domain-specific languages (DSL) [3], [4], [5], [6], [7], [8] have been proposed to provide higher-level abstractions for implementing network security services. However, those approaches are far from satisfactory for several reasons. First, understanding such domain-specific programming abstractions requires a high learning curve for network operators, e.g., understanding the usage of functional components (e.g., APIs or modules). Second, the implementation/enforcement of security appliances still needs non-trivial programming efforts, such as introducing variables and handling data-flow context between functional components. Consequently, it remains to be a time-consuming and labor-intensive task for a network operator to translate their high-level network security intents into actual implementations.

Thus, a fundamental research question is still open: *Can we provide a framework to assist network operators in synthesizing operational network security services directly from high-level needs in a natural language as automatically as possible?* In order to answer this challenging research question, we divide it into three sub-questions:

- How to properly model network security services to balance its expressibility in natural language and the ability of synthesis?
- How to effectively infer security intents from free-linguistic descriptions of security needs?
- How to synthesize and instantiate concrete security services according to security intents and resource constraints?

To answer the first question, we design a novel graph-based security service intermediate representation (SSIR) between natural language specifications and actual implementations of security services. In particular, we adopt a *micro-service* architecture [9], [4], [10] that splits the entire security service into basic function blocks, called security functional blocks (SFBs), and merges those functional blocks in a Directed Acyclic Graph (DAG) according to high-level network security specifications. In this way, we essentially reduce the problem of automatic network security service synthesis into the problem of automatic synthesis of security micro-services according to natural language specifications.

To infer security intents from high-level natural language descriptions, we leverage Natural Language Processing (NLP) techniques. Particularly, we convert natural language descriptions for network security policies/specifications into a normalized form, namely *semantic vector* that preserves its semantic meaning as well as reduces the analysis overhead. Then, we can identify SFBs from the semantic vector by

measuring the semantic relatedness and construct the SSIR by connecting identified SFBs. Furthermore, considering the potential incompleteness of natural language specification, we provide interactive synthesis techniques to help users validate the generated SSIR by providing preview sentences and smart interactions.

Traditionally, network security policies are mostly enforced in specialized hardware or software appliances as middleboxes, e.g., firewall, IDS/IPS. However, we consider that such rigid practices cannot meet the goals for the customization of user-defined high-level policies. To address the problem, we propose to leverage two emerging networking paradigms, i.e., Software-Defined Networking (SDN) and Network Function Virtualization (NFV), to provide a unified service enforcement. Atop SDN, we can enforce service enforcement modules and a couple of network security preventions and responses. Moreover, NFV techniques can help us make virtualized security analysis/detection functions in an on-demand way.

In this paper, we propose a novel framework, called AUTOSEC,which can aid network operators in automatically synthesizing intended security services according to high-level security policies. AUTOSEC utilizes NLP techniques/algorithms to automatically parse user intents from natural language into an SSIR. Furthermore, considering the potential incompleteness of natural language specification, it allows human in the loop to perform Interactive Synthesis to validate/refine the parsed SSIR if necessary. Then, AUTOSEC leverages SDN/NFV techniques to automatically build up operational security services on-demand. While not perfect (limitations discussed in Section X), AUTOSEC represents the first step towards an important but so far less touched network security area, i.e., automatic programming of network security services. Finally, the experimental evaluations and user studies showcase the effectiveness and usability of AUTOSEC.

The main contributions are highlighted as follows:

- We systematically study the design/implementation of existing network security appliances and propose a graph-based intermediate representation, i.e., SSIR, to model network security intents from natural language descriptions.
- We present an effective SSIR Extractor by using NLP techniques, which can effectively infer security intents from textual descriptions.
- We design, implement and evaluate AUTOSEC, a new framework to program network security services from natural language policies/specifications.

The rest of the paper is organized as follows: Section II introduces the background and the problem statement of the paper. Section III discusses the Security Service Intermediate Representation (SSIR) of network security service. Section IV presents the overview and assumptions of AUTOSEC. Section V describes the SSIR extraction, validation and refinement of AUTOSEC. Section VI presents security service composition and enforcement of AUTOSEC. Section VII presents the implementation of AUTOSEC and Section VIII evaluates AUTOSEC. Section IX reviews related work of the paper. Section X discusses limitations and future work. Section XI concludes this paper.

II. BACKGROUND AND PROBLEM STATEMENT

A. NLP Preliminaries

Named Entity Recognition. *Named Entity Recognition (NER)* is extensively adopted to identify and classify entities from text into predefined categories, e.g., person names, locations, and organizations. NER techniques are widely recognized to be domain-specific, i.e., NER systems in one domain usually work poorly in other domains [11].

Dependency Parsing. *Dependency Parsing* resolves grammatical dependency relations (e.g., determinant, noun compound modifier) between linguistic tokens of the natural language text. For example, Stanford Dependency Parser [12], can identify a grammatical dependency relation [13] as triplets of *relation*, *governor* and *dependent*, i.e., *relation(governor, dependent)*.

B. SDN and NFV

Software-Defined Networking (SDN). SDN has been rapidly innovating the networking industry through a new paradigm of network programming, as it decouples the control plane from the forwarding appliances of the data plane (e.g., routers and switches) into a logically centralized controller. By providing holistic visibility and flexible programmability, SDN controllers greatly facilitate network management tasks, e.g., load balancing, network virtualization, and access control, in a flexible and programmable manner.

Network Function Virtualization (NFV). Another trend, termed Network Function Virtualization (NFV), is to extract network (middlebox) functionalities from dedicated hardware boxes to software applications on commodity servers. By virtualizing network functions, NFV empowers flexible scaling and on-demand deployment of network services. Moreover, NFV can also benefit composition and deployment of network security services including firewall, and DDoS detection/prevention.

C. Problem Statement

The goal of AUTOSEC is to synthesize network security services that aim to detect, prevent, and respond to malicious/abnormal activities based on metadata information from network flows, such as flow identifiers (e.g., the Five-Tuple), payload content, and statistics. Here, we present an example policy to motivate our system AUTOSEC. One widely adopted network security policy is to counter attackers with early detection of reconnaissance attempts, e.g., port scanning attacks. To realize such a security posture, the network operator may document a high-level security policy as follows: "If we detect port scanning attacks in the network, we need to redirect these attacks to a honeypot."

Existing Practice. Implementing/enforcing such a policy is non-trivial with existing network security programming frameworks. First, developers need to conceptualize the security intents into low-level components (e.g., modules or APIs)

in a specific programming language abstraction and write programs to synthesize those components with a lot of low-level programming details. For example, for a simple scanning attack detection, it takes around 160 lines of code in low-level programming details on the system in the paper [5] and takes around 180 lines of code in an optimized Bro script (the initial version takes over 600 lines of codes). In addition, network operators need to configure all relevant network devices (e.g., Cisco routers or OpenFlow switches) to properly steer the target traffic to the correctly programmed security appliances. Such practice is tedious and error-prone due to its complexity. As a result, network security policies could take a long time (days to weeks) from plan to installation.

AutoSec Solution. In contrast, upon AUTOSEC, users can input natural language specifications of the target security service to generate and deploy the service and interact with the system to refine it if necessary. More specifically, from natural language specifications, AUTOSEC aims to automatically parse security intents including security functions (e.g., scan detection function and redirect function in the example policy) and their contexts. Optionally, AUTOSEC provides a human-in-the-loop feature that allows for interactive synthesis methods. This enables humans to refine parsed intents as needed. Finally, AUTOSEC automatically instantiates the parsed security intents using SDN and NFV techniques.

To further justify the motivation of AUTOSEC, we conducted a user study to understand what real network operators think regarding the usage of natural language in deploying/configuring network security services. We recruited 34 respondents, including 14 security experts. We found that most of the security experts (10 out of 14) reported that, based on their experience, they have already used the natural language to firstly express the security need before creating a network security service. Moreover, the majority of all participants (82.4% out of 34) thought it is a preferred method to directly create a network security service from a natural language description.

III. SECURITY SERVICE INTERMEDIATE REPRESENTATION

Instead of utilizing the monolithic representation for network security services, we adopt a microservice-based model, i.e., abstract network security functions as microservices and merge those functions based on natural language descriptions. The key insight of such microservice-based model is to hide complicated low-level functional procedures (which can hardly be expressed in natural language, e.g., machine learning based algorithms) and hence provide the proper programming abstraction that can be expressed in natural language and composable in the actual implementation. In this paper, we propose SSIR to serve as an intermediate representation between natural language descriptions and actual implementations of security services.

Definition 1 (Security Service Intermediate Representation). *A security service intermediate representation (SSIR) is a directed acyclic graph (DAG) $G = (E, V)$ of a network security service in* AUTOSEC*, in which each node V denotes a security functional block (SFB) and each edge E denotes a contextual relation between two connected SFB.*

Each SSIR consists of a finite number of V and E. Each V is an SFB, as explained in detail in Section III-A. In this way, we abstract unwanted low-level implementations such as details of header classifier. Also, we chain SFBs with contextual connections (i.e., E in graph G). These edges make sure of consistent data passing among SFBs. Figure 1 shows an example of SSIR for the security policy of detecting scanning attacks. This SSIR consists of three security functional blocks, in which *Header Classifier* and *Count Classifier* are used to classify suspicious incoming flows, and *Alert* is used to respond to those suspicious ones. The edges in SSIR between security functional blocks denote their contextual connections. For example, the header classification (with a parameter of IP) decides the work flow towards the count-based inspection.

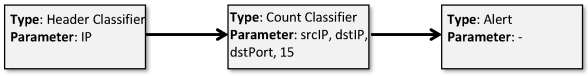

Fig. 1. An example SSIR for security policy

TABLE I
EXAMPLE SFBS OF NETWORK SECURITY SERVICE.

Type	Example Parameters
Detection	
HeaderClassifier	src_ip, dst_ip, protocol
PayloadClassifier	payload_pattern
RateClassifier	rate_num, rate_unit
CountClassifier	count_num, interval
StateClassifier	state_info
Prevention & Response	
Allow	src_ip, dst_ip, protocol
Deny	src_ip, dst_ip, protocol
Quarantine	src_ip, dst_ip, protocol
Redirect	src_ip, dst_ip, protocol
RateLimit	src_ip, rate_num, rate_unit
Mirror	src_ip, dst_ip, location
Proxy	protocol, src_ip, dst_ip
Alert	alert_info
Log	log_info

A. Security Functional Block

To compose a network security service, we define SFBs as basic elements to serve network security functions. In this paper, we define a set of SFBs (examples shown in Table I) that can cover many common security services (will be expanded over time). For instance, a firewall may use *HeaderClassifier* component to *Deny* malicious flows and *Alert* users; payload-based classification methods, e.g., *regex classifier* and *string classifier*, are widely used in intrusion detection/prevention (IDS/IPS) systems and data loss prevention (DLP) systems. We also note that, in some cases, a network operator may already have some existing/customized security functions that are not covered by our pre-defined SFBs. In such cases, we also allow the network operators to supply their existing/customized SFBs by providing their natural language descriptions. AUTOSEC will automatically learn/identify pre-defined and user-supplied SFBs in natural language security policies (see Section V-B).

As illustrated in Table I, an SFB has two interfaces: **type** and **parameter**. For a specific SFB, *type* specifies the abstracted function of the SFB, which can be further defined with *parameter*. For example, a *Deny* SFB can be specified to block a specific host with parameters of source IP address or terminate a specific TCP connection with parameters of the Five-Tuple (i.e., source/destination IP, source/destination port and tcp protocol).

B. Contextual Connections

To build an SSIR, we also need to reason about the connections between each SFB. There are different types of contexts that can be passed between security functions, such as a numeric value, a logic value and even the header of a packet. Considering the feasibility of the inferring processing based on natural language descriptions, we use a logic connection to denote that the execution of the latter SFB is dependent on logic metadata (i.e., true or false) from the former SFB in a specific parameter.

IV. System Overview and Assumptions

System Overview. Figure 2 overviews the process of AutoSec for synthesizing network security services from natural language specifications/policies. The intuitive inputs of AutoSec are natural language descriptions of network security services, e.g., specified by network security administrators. The SSIR Extractor module of AutoSec extracts internal security intents from natural language policies and converts them into an initial SSIR. Next, the SSIR Refiner module resolves the (potentially) ambiguous mapping (from natural language descriptions to SSIR) by conducting interactive synthesis to facilitate validation and refinement of the initial SSIR. Lastly, the Service Composer module merges multiple SSIRs into a concise composition to represent the whole space of policies in scope, e.g., a company, campus or cloud. the Service Composer module dynamically translates composed SSIRs into low-level configurations of network appliances to enforce actual security services. Furthermore, users can use real network test cases to validate the deployed network security services and rectify those incorrect intent in SSIR based on counter-examples.

System Assumptions. An ideal user of AutoSec is a network administrator of an operational network, who can control and configure network (SDN) controllers, network switches, and commodity servers to deploy/enforce desired security services. Also, the network administrator can overcome the topological constraints in an operational network by configuring non-OpenFlow devices (e.g., Cisco Router) to tunnel stakeholders' traffic to AutoSec. We assume network infrastructures, including network controllers, switches, and security function provision servers, are not compromised by an adversary since those resources could be well protected by the network administrators and not permitted to be accessed by normal users. Moreover, we assume administrative and control channels between network administrators, SDN controllers, and switches are well protected by SSL/TLS. In addition, we assume that there are no functional or security flaws in the implementations of these SFBs. The verification of the correctness of SFBs is out of the scope of this paper.

We assume that the input of AutoSec can be decomposed into disjoint textual blocks, where each block can be semantically mapped to a specific security function, i.e., the security descriptions can be modeled in the form of SSIRs by composing existing SFBs.

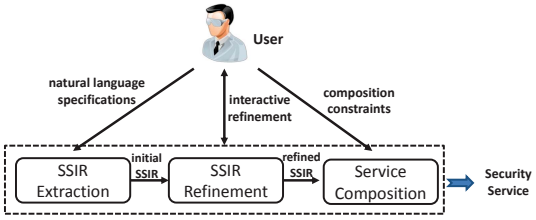

Fig. 2. System Architecture of AutoSec

V. SSIR Extraction, Validation and Refinement

A. Normalizing Natural Language Inputs

In the first step, the SSIR Extractor module transforms the natural language sentence (for network security policies or SFBs) into a normalized and context-aware form, called *semantic vector*, which reduces some irrelevant terms (e.g., stop words) and ambiguous entities (e.g., resource labels).

Definition 2 (Semantic Vector). *A semantic vector is a vector of lemmatized, network-security relevant terms that preserves the grammatical relations and semantic meaning of a natural language sentence.*

Identifying Network-Security Relevant Terms. To understand network-security relevant terms, we use a domain-specific dictionary to identify key terms from textual input, which covers abbreviations (e.g., IP, TCP, ACL) and phrases (e.g., ICMP Time Exceeded message) mentioned from Wikipedia and online glossaries of network terms [14], [15]. Moreover, we adopt regular expressions (regex) for matching domain-specific notations. For efficiency and disambiguation, we replace identified entities with their named tags, e.g., *%Protocol* for TCP.

Vectorizing and Normalizing. We adopt a tokenization to chunk processed text into a list of words and remove common stop language list in English language [16]. Moreover, we apply *lemmatization* [17] to normalize each remaining words to its *lemma*, i.e., the canonical form of a word.

B. Classifying Security Functional Blocks

Semantic Similarity Measurement. To identify SFBs from the semantic vector of the input text, we first measure if a subset of the semantic vector of the input text is semantically related to one or more semantic vectors of SFBs. We utilize WordNet [18], a widely used dictionary for ontology [19], which provide a structured network between two general concepts including nouns, verbs, adjectives, and adverbs. Semantic relatedness between two concepts can be estimated

with topological distance by using ontology model [20]. We further utilize *Align, Disambiguate and Walk (ADW)* algorithm [21] to measure semantic similarity between two pairs of terms with arbitrary size as *Similarity Score*:

$$SimScore(SV, SFB) = Argmax\{x|ADW(SV, x), x \in SFB\}$$

N-gram based Classification. With the understanding of the semantic similarity between two chunks of terms, we then locate SFBs from the semantic vector of the input text with a greedy *n-gram* based algorithm (illustrated in Algorithm 1). In particular, we first compute a set of *adjacency powerset* [22] of input semantic vector that includes all possible *n-gram* terms from the semantic vector of the input text. Next, we iteratively locate the best matching element from *adjacency powerset* to SFBs, i.e., the element with the maximum *similarity score* with the semantic vector of SFBs. Then we remove all relevant elements with the identified element from the *adjacency powerset*. The iteration terminates if the maximum *Similarity Score* of one round is lower than the threshold or the *adjacency powerset* is exhaustively explored. After identifying SFBs, we also attempt to complement their parameters by matching elements in the semantic vector, e.g., an IP address, to the parameter lists of parsed SFBs.

Algorithm 1 Classifying Security Functional Blocks

Input: T: semantic vector of input text, ST: a set of semantic vectors of SFBs.
Output: SFB: a list of SFBs correlated with input text.

$SFB \leftarrow \emptyset$, $S \leftarrow \emptyset$, $sv \leftarrow \emptyset$
Normalize ST
for i = 1 to sizeof(T) **do**
    Put i-gram of T into ST
end for
while S != $\emptyset$ **do**
    Locate the u with maximum SimScore(u, ST) in S
    if SimScore(u, ST) > THRESHOLD **then**
        Put corresponding sfb into SFB
    else
        return SFB
    end if
    remove all subset and superset of u from S
end while
return SFB

Blackboxing Completion. To handle unidentified security functions, we also provide a Blackboxing Completion approach that places a functional blackbox in the SSIR that allows users to customize their security detection/analysis algorithms with annotation in the raw input text. AUTOSEC will automatically the connections between the black box to other SFBs.

C. Resolving Contextual Connections

To construct SSIR, we further utilize grammatical dependencies to resolve explicit connections between located SFBs and type-based synthesis techniques to complement implicit connections. We convert grammatical dependencies of a sentence into a DAG representation, in which each node denotes terms in the sentence and each directed edge denotes their grammatical dependencies. Then we build up explicit grammatical relations between each semantic group (corresponding to an SFB) by amalgamating all grammatical relations. For isolated SFBs, we adopt type-based heuristics to chain them: 1) a detection SFB (e.g., *payload classifier* or *count classifier*) always points to a prevention/response SFB (e.g., *deny* or *alert*); 2) a *header classifier* SFB always precedes other detection SFBs.

D. SSIR Validation and Refinement

We consider that security policies in natural language may be implicit when inferring network security intents from users' descriptions due to two reasons: 1) the natural language specification may ignore low-level details (e.g., parameters for SFBs); 2) parsing intents from natural language descriptions (even by using state-of-the-art NLP techniques) may be incomplete. To tackle the problem, AUTOSEC conducts interactive synthesis by providing users a preview of the SSIR parsed from natural language (an example as shown in Figure 3) and initiating a smart interactive conversation to facilitate users to validate and refine SSIR.

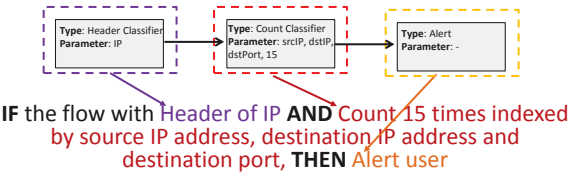

Fig. 3. Example of SSIR Preview

VI. SERVICE COMPOSITION AND ENFORCEMENT

Given the refined SSIRs derived from SSIR Refiner module of AUTOSEC, the Service Composer module further composes consolidated security services and deploys them in the operational network. Service Composer of AUTOSEC verifies all classification SFBs (as listed in Table I) for all input SSIRs. In particular, it first translates each policy into mappings from conditions to actions. The conditions cover both types of classifiers and their parameters. For example, the predicate of IDS policy (in Figure 1) may be presented as "header:tcp, dst_port(443), payload:0x18030200". Then, Service Composer locates all pairs of policies that have intersections in their predicates and resolves their conflicting actions by checking their priority (if applicable). Service Composer will raise a composition alert for any unsolved conflict to the network operators.

Context-aware Processing. According to composed security intents, AUTOSEC instantiates them into VMs/containers. In order to preserve security contexts, we adopt tag-based flow processing [23] by maintaining a global tag table for the entire network. When processing a packet, each network device may inspect its tag and adopt corresponding security actions. As illustrated in Figure 4, based on the SSIR, the control plane will pre-assign tags to share contexts between SDN switches, security NFs and the control plane. Those contexts cover each flow predicate (in *header classifier* SFBs) and flags for security response/prevention SFBs. If a packet matches

predicates specified in header classifier SFBs, the switch will put the corresponding tag into the VLAN field of the packet and route it to the port connected to security NFs indicated in the SSIR. Then the security NFs can handle the packet based on its logic and the SSIR to add tags for security actions. Lastly, the SDN switch handles the packet based on the tag information from security NFs. For user-supplied SFBs and blackbox security functions, AUTOSEC asks users to specify the location of the corresponding runnable security functions (e.g., the attachment of a specific switch port) and tag mapping information for those functions.

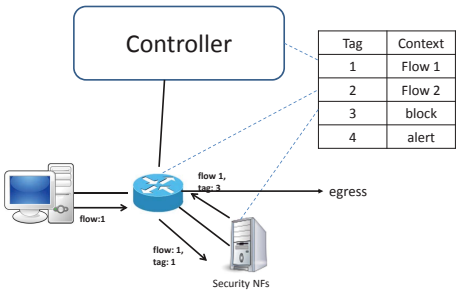

Fig. 4. Context-aware service composition

VII. IMPLEMENTATION

We use Floodlight SDN Controller [24] as the security service manager for our system. We implement the service composition module in Floodlight Controller, which automatically parses SSIRs generated from the SSIR Extractor and instructs switches and software-based network functions to enforce it. Moreover, we embed the SSIR Extractor with a conflict detection engine by using alias set rule reduction algorithm [25], [26].

We implement 14 SFBs listed in Table I upon abstracted network resources by leveraging SDN and network functions. In particular, we leverage Click Router [27] to implement SFBs for packet/flow analysis. In addition, We also implement some security prevention/response SFBs within the data plane by using various actions of OpenFlow flow rules. First, we use an *Output* OpenFlow action to implement *Allow* and *Mirror* SFBs: for *Mirror* SFB, the output port is specified in its parameter, however, for *Allow* SFB, the output port is dependent on the routing service in the controller. In contrast, we use a *Drop* OpenFlow action to realize *Deny* SFB. We parse *Quarantine* SFB into a list of flow rules to only allow communication from a specific host within a restricted network. For *Redirect* SFB, we use *Set-Field* action to modify the target addresses in the packet header. Moreover, we utilize *Set-Queue* action to implement *RateLimit* SFB. In addition, we consider that 3 security prevention/response SFBs require the involvement of the AUTOSEC control plane. For *log* and *alert* SFBs, we pass the packets to the control plane, which instantiates the corresponding service to record or alert the detected threat. Also, we implement a SYN proxy service in the control plane to enforce the *proxy* SFB.

VIII. EVALUATION

A. Experimental Setup

We evaluated a prototype of AUTOSEC in a Ubuntu Linux OS. For domain-specific terms, we implement 16 types of regular expressions for domain-specific named entities (e.g., IPv4 address, MAC address, port number, hash) and about 300 domain-specific entries in the dictionary by utilizing Stanford TokensRegex framework. We choose 0.3 as the threshold for Security Functional Block classification. To evaluate the effectiveness of security services generated from AUTOSEC, we set up an experimental topology based on Mininet [28].

Dataset. We collected security policy descriptions from a variety of data sources including research papers, descriptions of security appliances, documented real-world security standards, and human-written security policies. For research papers, we utilized network security relevant keywords, e.g., "network", "security", "defense", to filter out recent 10-year research papers from S&P, CCS, USENIX Security, NDSS, and RAID. We manually extract/verify network security-relevant policy descriptions from data sources. Moreover, we utilized the search engine to collect security policies for various security appliances including iptables, ufw, pfsense. Furthermore, we collected 18 security policies from operational security specifications including 9 security policies from a configuration standard for real-world operational networks [29] and 9 network security policies from Payment Card Industry (PCI) Data Security Standard [30]. In addition, we asked five people with network security backgrounds to individually write down security policies to realize 10 security requirements including 5 access control security requirements referring to online configuration exercise for security policy [31] and 5 more security requirements for network attack detections. In total, we collected 190 security policies. Since there is no ground truth for the security intents parsed from natural language descriptions, we asked two experts individually label SFBs and their connections. Then, we took their consent labeling results as ground truth for our data sources.

B. Evaluations on SSIR Extractor

In this part, we first evaluate the effectiveness of AUTOSEC in classifying SFBs from security policies. Then, we test the overall differences between SSIRs parsed by SSIRs and the manually labeled ones by using edit distance. Lastly, we evaluate the performance of SSIR Extractor in generating SSIRs. Note that, we excluded those sentences in security policies if they do not contain any SFB.

Effectiveness of SSIR Extractor. Columns 2-5 of Table II showcase the result for the inference of SFBs from AUTOSEC. In particular, AUTOSEC can correctly locate all SFBs for 84.2% of all our collected policies and over 77% of each individual data source. Totally, AUTOSEC falsely classified (at least one) irrelevant SFBs for 15 policies and missed (at least one) relevant SFBs for 27 policies. One cause of the misclassification of AUTOSEC stems from some (generally) semantically related terms (e.g., synonyms) that are

TABLE II
RESULTS OF AUTOSEC: #FI DENOTES THE NUMBER OF POLICIES WITH INCORRECT SFBS. #MI DENOTES THE NUMBER OF POLICIES MISSING SOME CORRECT SFBS. %C DENOTES THE PERCENTAGE OF POLICIES WITH ALL CORRECT SFBS. ED DENOTES THE EDIT DISTANCE BETWEEN PARSED AND LABELED SSIRS. P/A MEANS PASSED SECURITY SERVICES OVER ALL POLICIES.

Data Source	#FI	#MI	%C	ED	P/A
Research paper	3	3	80%	0.85	18/20
Appliance description	4	11	88.2%	0.16	102/102
Human writing	4	9	82%	0.68	49/50
Operational specification	4	4	77.8%	0.78	18/18
Overall	15	27	84.2%	0.43	187/190

not closely related in the context of network security. For example, AUTOSEC infers *Allow* SFB from a security policy, i.e., *"For SSH, Bro provides a heuristic that determines if a login succeeded or failed, based on the volume of data exchanged as well as the number of packets seen during the session."* After checking each pair of the term in the dictionary, we found word "provide" and "allow" are synonyms, which account for the misclassification. Another source of misclassification of AUTOSEC lies in the failure of handling negated contexts in security policy descriptions. For example, AUTOSEC mistakenly locates *allow* from policy *"you don't want to allow direct connection from outside"*. In addition, AUTOSEC misses SFBs due to the misunderstanding of terms with implicit meaning in their context. For example, AUTOSEC did not infer any classification SFB in policy *"Check openSSL version, and block those lower than 1.0.1g"*, because it can hardly understand the meaning of openSSL version.

We also measured the effectiveness of SSIR Extractor module by comparing differences between SSIRs parsed from AUTOSEC and manually labeled ones without consideration of parameters of each SFB. In particular, we use *edit distance* as a metric to determine the minimum transformations between two graphs. We set the cost of node addition, node removal, edge addition, and edge removal operations as 1. As a result, the average edit distance of the SSIRs generated by AUTOSEC is 0.43 compared with the labeled SSIRs. The small number showcases that AUTOSEC can effectively assist network administrators to set up an initial SSIR from natural language policies.

C. Effectiveness of Service Composition

To evaluate AUTOSEC in service composition and enforcement, we set up a test topology and tested the effect of actual service orchestration in the data plane. In particular, we input the SSIRs parsed by SSIR Extractor and refined (if needed) the parameters of SFBs in parsed SSIRs during policy refinement for testing purposes, e.g., configuring thresholds for anomaly detections or making IP addresses consistent between SSIRs and devices in the topology. Then, we launched corresponding test cases, e.g., generating packets to violate ACL policies, to observe if the SSIR was successfully enforced (e.g., via alerts in the control plane or monitoring traffic in target devices).

Column 6 of Table II lists the test result. In particular, 187 (out of 190) security services generated by AUTOSEC can successfully pass their test cases. For those three failed cases, we find that their corresponding policies include some security functions that are not provided by AUTOSEC. One of them is from human writing policies and the rest two policies are from research papers. In particular, the human writing policy includes a check of software version in victim servers. The first policy from papers is a stateful firewall policy that requires to judge if a packet is authenticated (via web or 802.1x). The second one from papers is a spam filter policy that needs to decide if there is a new MTA in the SMTP path. The result also calls for future research to embed more customized security functions in AUTOSEC framework.

TABLE III
EXAMPLES OF RED-BLUE TEAM EXERCISE

Red-team Scenarios	Blue-team Policies
Port Scanning	The script tracks the number of unique ports and destination addresses that attempts to connect to, generating alarms when they exceed, by default, 15 or 25 attempts within a 5 minute interval, respectively.[5]
Topology Inference	One of them counts the number of packets per host pair with TTLs lower than 10. The second counts the number of ICMP Time Exceeded messages relating to the same hosts. We consider a traceroute to be in progress if we see at least one low-TTL packet between a pair of hosts along with at least three matching ICMP Time Exceeded messages.
UDP Flooding	The analysis node A_UDP identifies source IPs that send an anomalously higher number of UDP packets and uses this to categorize each packet as either attack or benign. The function forward will direct the packet to the next node in the defense strategy; i.e., R_OK if benign, or R_LOG if attack. [32]
SQL Injection	We first wrote a regular expression that matches typical injection URIs (e.g., /site.php?site=5' and 1=1 and '='). We then set up two summary statistic instances to count the number of times the regular expression matches. For the first instance, the key is the source IP address while for the second we use the destination address. Once one of the instances hits a configured threshold of matching requests (50 in 5 minutes), the detector triggers an alert.
Buffer Overflow	Detect buffer overflow attack by counting the number of "0x90" in the payload of the packet.
Unauthorized Access	The policy block packets between unprivileged devices and protected servers.
Credential Sniffing	The policy sends an alert when it observes keyword "password" in the payload from PCs.
Password Guessing	Block the host if several failed HTTP login attempts from the same IP address.

D. Red-Blue Team Exercise

Furthermore, we conducted a red-blue team exercise with two participants with a network security background. In the exercise, One participant (the red team) selects some common network attack scenarios including unauthorized accesses, network reconnaissance (port scanning and topology inference), UDP flooding, user privacy/credential stealing, SQL injection attacks, buffer overflow, and so on. The other participant (the blue team) inputs the defensive security policy descriptions into AUTOSEC as shown in Table III. In each scenario, the red team launches a corresponding attack to achieve the attack goal. Here are some example attack strategies adopted by the red team: use *Nmap* to scan the vulnerability of a public server; utilize *traceroute* to infer network topology; employ compromised hosts to launch a UDP flooding attack against a public server; guess the website credentials of the user by utilizing dictionary attacks. The blue team inputs defensive policies (from existing papers, documents, or human writing) into the SSIR Extractor, refines the final SSIR, and puts the SSIR into Service Composer to detect/prevent the attack. In all cases, AUTOSEC can successfully help the blue team to compose/deploy security services to detect or prevent corresponding network threats.

We now discuss in detail the case study on scan attacks to showcase how AUTOSEC can benefit the composition of security services. The detection policy is from [5] and shown in Figure 5. As illustrated in Figure 5, a network administrator

(as the blue team) first passed the high-level policy descriptions of scan detection to AUTOSEC, which extracted an initial SSIR. Based on the initial SSIR, then AUTOSEC provides a preview sentence as "IF the flow with Header of IP AND Counter of source IP address and destination IP address is over 15 within 300 seconds, THEN Alert user", and inquires the blue team to refine the policy. Note that, during the refinement conversation, the blue team can also add new security actions, i.e., redirecting the traffic to a honeynet, into the SSIR. Then AUTOSEC enforces a service into the data plane from the refined SSIR. When the red team used *Nmap* [33] on the compromised host to launch port scanning attacks to the public server, AUTOSEC control plane was correctly alerted by such attack and the suspicious flow was redirected to a honeynet for further inspections.

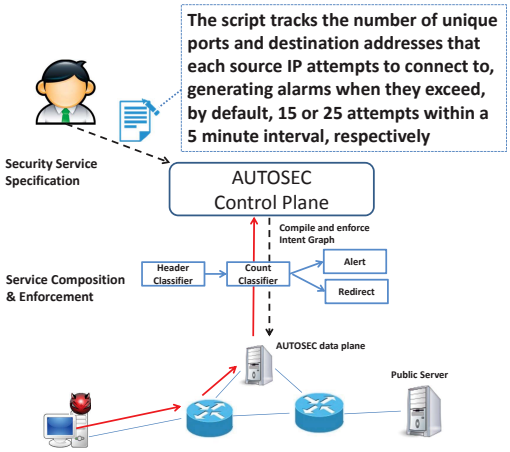

Fig. 5. Scan detection Service using AUTOSEC

E. User Study on the Usability of AUTOSEC

We designed a user study to answer the following question: When users use AUTOSEC to create a network security service, do they find AUTOSEC usable in practice (e.g., does AUTOSEC achieve the goal with no or minimal user refinement)? In this user study, we first helped the participants explore AUTOSEC with an end-to-end user experience given real-world scenarios. Then, we collected participants' responses regarding both their step-by-step user experience and overall evaluation of AUTOSEC.

Methodology. In October 2020, we carried out a user study that utilized MTurk for participant recruitment and payment, while Google Forms was utilized to implement the survey questionnaire. Our user study was designed with two key principles in mind to ensure high quality and accurate results. Firstly, we aimed to recruit participants with a foundational background in network and security to facilitate thoughtful responses to our survey questions. Secondly, we designed our survey questions to emulate real-world use cases, enabling participants to better understand the practical implications of their responses and provide more meaningful insights.

Recruitment. We recruited 34 participants from MTurk to conduct the survey. All of the participants are required to have experience related to network security applications such as a firewall. When recruiting the participants, we use MTurk's criteria filtering function to recruit respondents whose employment is Industry Software & IT Service. We also collected their detailed employment and education background, such as their job titles, work experience, and highest degree earned. Moreover, we set a requirement of the approval rate (i.e., 90%) and approval number (i.e., 100) of Human Intelligence Tasks (HIT) for the participants. We paid MTurk 10 dollars for each finished HIT.

TABLE IV
USER STUDY RESULT.

Participant Group	**Need Refine (%)**	**Refine Satisfy (%)**	**Overall Satisfy (%)**	**Helpfulness (%)**
All Participants	14.7%	80%	84.8%	73.6%
Security Experts	14.3%	100%	92.4%	85.8%
Higher Education (Bachelor or higher)	6.3%	66.7%	83.9%	71.9%

Survey. Among the 34 respondents, most of the respondents are male (67.6%). While all of them claimed to be from the IT field, more than half of them (18 out of 34) identified themselves to be software developers or similar job titles. Moreover, we consider 14 of them to be security experts. This is because their job titles are highly related to the computer security field. For example, one of them works as "Security Consultant - Web, Mobile, Network Pentesting". We also asked them if they have ever used network security tools to confirm whether they have the proper knowledge for the survey. As a result, all of them reported "yes" to this question. Another question we asked is to check the most familiar tool or markup language related to network security services. The result showed that *iptables* was the most reported one.

Result. To evaluate the usability of AUTOSEC, we first showed the respondents AUTOSEC' first step result (e.g., Figure 3 in Section V) based on the input natural language description. Two different scenarios were used in this user study. Then, we asked the respondents if they were satisfied with the result AUTOSEC generated. Based on the collected results from all respondents, most of them (67.6%) thought the yielded result was good enough and usable for them. Only 14.7% (5/34) of respondents thought that there needs further refinement to complete service creation (and the remaining 17.7% were neutral). For these 14.7% respondents who chose to further refine the result, we provided them the options for what parameters in the modules they could change. As shown in Table IV, most of these respondents (80%) were satisfied with the refining process/result provided by AUTOSEC. Moreover, we surveyed their overall satisfaction rate on AUTOSEC and if they thought AUTOSEC helped reduce the workload of network security administrators' job in creating and configuring network security services. The results indicated that most of the respondents agreed on: 1) they were satisfied with AUTOSEC user experience; 2) AUTOSEC would significantly reduce their workload in creating a network security service.

IX. Related Work

Programming Frameworks/Abstractions for Network Security. Bro [8] provides an event-driven scripting language for programming network security monitor applications. Existing SDN controllers [24], [34], [35] allow developers to write network security applications by using general-purpose programming language. FRESCO [3] presents an SDN-based security application development framework to compose network security services by using a scripting language. OpenBox [4] and Slick [36] propose a programming abstraction to ease the development and management of middleboxes, which can also enforce some security appliances, e.g., firewalls and IDS. Some recent programming language research efforts [37], [38], [6], [7], [39] offer new domain-specific programming languages for SDN switches, which also simplify the development and deployment of network security services. PSI [40] proposes a scalable enterprise network security framework to enforce context-aware, flexible security policies by using DSL or GUI. However, programming network security applications/services upon above frameworks/abstractions are still tedious since they require users to first conceptualize their high-level security intents into low-level components (e.g., modules or APIs) and write programs to synthesize those components that follow high-level security specifications. In comparison, AutoSec provides a new framework to automatically build up security services to enforce high-level network security needs described in natural languages.

NLP for Security and Privacy. We also review NLP applications in the literature on security and privacy. NLP is adopted in Android security to check consistency between applications and their required permissions [41], [42]. AutoPGG [43] presents a framework that automatically constructs descriptions of the privacy policy for Android applications. UIPicker [44] and SUPOR [45] utilize NLP to locate sensitive input from Android applications. SEISE [46] leverages NLP techniques to locate infected websites. iACE [11] showcases a novel application of NLP for automated cyber threat intelligence (i.e., IOC) extractions. FeatureSmith [47] proposes an NLP-based feature engineering for malware. Compared with them, AutoSec exhibits a novel application of NLP that bridges the gap between high-level network security policies (i.e., natural language descriptions) and well-deployed security services. Moreover, some works [48], [49] have been proposed to extract access control rules from natural language descriptions based on semantic/syntactic patterns. Different from them, AutoSec covers diverse types of network security services without the knowledge of their semantics/syntactics.

NLP for Automated Programming and Configuration. There are a couple of prior works proposed to translate natural language descriptions to different types of program languages [50], [51], [52], [53], [54], [55]. For example, NL2Bash [50] presents novel data and semantic parsing approaches to map natural language descriptions to Bash commands. SQLizer [51] proposes an end-to-end system to automatically synthesize SQL queries from natural language and NaLix [52] introduces interactive natural language interface for XML queries. In addition, many works, such as [53], [54], [55], attempt to translate natural language statements to general-purpose programming languages. GPT-3 [56] is proposed to leverage pre-trained language model with task-specific tuning, which can also automatically generate programming code. However, all of those NLP-based automated program/configuration synthesis systems are coupled with the syntax of programming/configuration languages, and they can hardly help synthesize network security applications/services as AutoSec attempts to do.

X. Limitations and Discussions

Correctness and Security of SFBs. It is clear that any incorrectly/insecurely implemented SFBs, e.g., with bugs/vulnerabilities, may incur functional or security breaches of composed security services from AutoSec. The validation/verification of the correctness and security of SFBs could be one interesting future work.

Misclassified SFBs. One limitation of AutoSec lies in possibly misclassified SFBs, which is mostly due to the limited ontology model we use. Our future work plans to build better network-security-domain-specific ontologies, which can benefit AutoSec and future researches of NLP applications in network security.

Unrecognizable SFBs and limited SFB number. As any NLP-based application, AutoSec cannot perfectly infer all intents from descriptions of complicated functions because of the ambiguous and imprecise natural of natural languages and the complexity of network security functions. Our current system supports 14 abstracted general functional blocks that can be used to compose network security services. This is certainly not a complete list. Thus, we expect to expand over time and we also allow user-supplied SFBs. Currently, to handle unrecognizable SFBs, we provide a Blackboxing Completion approach as mentioned in Section V-B. Future work is needed in this space to handle/understand/supply complex network security functions and we envision new advances in AI/NLP, e.g. Large Language Models [56], could benefit this area.

XI. Conclusion

In this paper, we propose AutoSec to automatically build up network security services from high-level user security descriptions/intents. Our evaluation shows that AutoSec is promising. While clearly our work is not perfect and there is much room to improve, we believe this is an important first step towards an important security research direction, i.e., automatic synthesis of network security services. We hope our work can stimulate more further research into this area.

Acknowledgement

This material is based upon work supported in part by NSF under Grant No. 1700544, 1816497, 2148374, and ONR Grant No. N00014-20-1-2734. Any opinions, findings, and conclusions or recommendations expressed in this material are

those of the authors and do not necessarily reflect the views of NSF and ONR.

REFERENCES

[1] "Secure network life-cycle management," http://www.learncisco.net/courses/iins/common-security-threats/secure-network-life-cycle-management.html.

[2] "Dangers of Complexity in Network Security Environments," http://wp.eurosecglobal.de/wp-content/uploads/2013/04/www.algosec.com_resources_files_Specials_Survey%20files_12_10_11_security_complexity.pdf.

[3] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "FRESCO: Modular Composable Security Services for Software-Defined Networks," in *NDSS'13*, 2013.

[4] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions," in *SIGCOMM'16*, 2016.

[5] J. Amann, S. Hall, and R. Sommer, "Count Me In: Viable Distributed Summary Statistics for Securing High-Speed Networks," in *RAID'14*, 2014.

[6] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," in *ICFP,11*, 2011.

[7] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN Programming with Pyretic," in *Technical Reprot of USENIX*, 2013.

[8] "The Bro Network Security Monitor," https://www.bro.org/.

[9] N. Dragoni, A. L. L. Saverio Giallorenzo, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, 2017.

[10] H. Li, H. Hu, G. Gu, G.-J. Ahn, and F. Zhang, "vNIDS: Towards Elastic Security with Safe and Efficient Virtualization of Network Intrusion Detection Systems," in *CCS'18*, 2018.

[11] X. Liao, K. Yuan, X. Wang, Z. Li, L. Xing, and R. Beyah, "Acing the IOC Game: Toward Automatic Discovery and Analysis of Open-Source Cyber Threat Intelligence," in *CCS'16*, 2016.

[12] "Stanford NLP toolkit," https://nlp.stanford.edu/software/.

[13] S. Schuster and C. D. Manning, "Enhanced English Universal Dependencies: An Improved Representation for Natural Language Understanding Tasks," in *LREC '16*, 2016.

[14] "Glossary of Network Security Terms," http://www.networksorcery.com/.

[15] "Glossary of Network Security Terms," https://www.sans.org/security-resources/glossary-of-terms/.

[16] "Stop Word List," http://www.ranks.nl/stopwordss.

[17] J. Plisson, N. Lavrac, and D. Mladenic, "A Rule based Approach to Word Lemmatization," in *SiKDD'04*, 2004.

[18] C. Fellbaum, "WordNet: An Electronic Lexical Database. Cambridge," in *MA: MIT Press*, 1998.

[19] M. A. Rodriguez and M. J. Egenhofer, "Determining semantic similarity among entity classes from different ontologies," in *IEEE Transactions on Knowledge and Data Engineering*, 2003.

[20] D. Lin, "An Information-Theoretic Definition of Similarity," in *ICML '98*, 1998.

[21] M. T. Pilehvar, D. Jurgens, and R. Navigli, "Align, Disambiguate and Walk: A Unified Approach for Measuring Semantic Similarity," in *ACL'13*, 2013.

[22] "Wikipedia: Power Set," https://en.wikipedia.org/wiki/Power_set.

[23] S. K. Fayaz, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags," in *NSDI'14*, 2014.

[24] "The Floodlight SDN Controller," https://github.com/floodlight/floodlight.

[25] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A Security Enforcement Kernel for OpenFlow Networks," in *HotSDN'12*, August 2012.

[26] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the Software-Defined Network Control Layer," in *NDSS'15*, 2015.

[27] "Click Router," http://read.cs.ucla.edu/click/click.

[28] "Rapid prototyping for software defined networks," http://mininet.org/.

[29] "Colorado department of education - network firewall implementation policy," https://www.cde.state.co.us/dataprivacyandsecurity/networkfirewallpolicy.

[30] "Payment card industry (pci) data security standard," https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-2-1.pdf.

[31] "CCNA blog," http://www.ccnablog.com/acls-part-ii/.

[32] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and Elastic DDoS Defense," in *USENIX Security 15*, 2015.

[33] "Nmap security scanner," https://nmap.org/.

[34] "The ONOS SDN Controller," http://onosproject.org/.

[35] "The OpenDaylight SDN Controller," https://www.opendaylight.org/.

[36] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming Slick Network Functions," in *SOSR'15*, 2015.

[37] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch," in *SIGCOMM CCR '14*, 2014.

[38] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful Network-Wide Abstractions for Packet Processing," in *SIGCOMM'16*, 2016.

[39] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," in *NSDI'15*, 2015, pp. 59–72.

[40] T. Yu, S. K. Fayaz, M. Collins, V. Sekar, and S. Seshan, "PSI: Precise Security Instrumentation for Enterprise Networks," in *NDSS'17*, 2017.

[41] R. Pandita and X. Xiao, "Whyper: Towards automating risk assessment of mobile applications," in *USENIX Security'13*, 2013.

[42] Z. Qu and V. R. X. Zhang, "Autocog: Measuring the description-to-permission fidelity in android applications," in *CCS'14*, 2014.

[43] L. Yu, T. Zhang, X. Luo, and L. Xue, "AutoPPG: Towards Automatic Generation of Privacy Policy for Android Applications," in *SPSM'15*, 2015.

[44] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "UIPicker: User-Input Privacy Identification in Mobile Applications," in *USENIX Security'15*, 2015.

[45] J. Huang, Z. Li, and X. Xiao, "SUPOR: Precise and scalable sensitive user input detection for android apps," in *USENIX Security'15*, 2015.

[46] X. Liao, K. Yuan, and e. a. X. Wang, "Seeking nonsense, looking for trouble: Efficient promotional-infection detection through semantic inconsistency search," in *S&P'16*, 2016.

[47] Z. Zhu and T. Dumitras, "FeatureSmith: Automatically Engineering Features for Malware Detection by Mining the Security Literature," in *CCS'16*, 2016.

[48] J. Slankas, X. Xiao, L. Williams, and T. Xie, "Relation extraction for inferring access control rules from natural language artifacts," in *ACSAC'14*. ACM, 2014.

[49] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie, "Automated extraction of security policies from natural-language software documents," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 12.

[50] X. V. Lin, C. Wang, L. Zettlemoyer, and M. D. Ernst, "NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system," in *LREC'18)*. Miyazaki, Japan: European Languages Resources Association (ELRA), May 2018. [Online]. Available: https://www.aclweb.org/anthology/L18-1491

[51] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "Sqlizer: query synthesis from natural language," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 63, 2017.

[52] Y. Li, H. Yang, and H. Jagadish, "Nalix: an interactive natural language interface for querying xml," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 2005, pp. 900–902.

[53] D. Price, E. Rilofff, J. Zachary, and B. Harvey, "Naturaljava: a natural language interface for programming in java," in *Proceedings of the 5th international conference on Intelligent user interfaces*. ACM, 2000, pp. 207–211.

[54] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *arXiv preprint arXiv:1603.06744*, 2016.

[55] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2017, pp. 440–450.

[56] T. B. Brown and et. al., "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.