# vNIDS: Towards Elastic Security with Safe and Efficient Virtualization of Network Intrusion Detection Systems

Hongda Li
Clemson University
hongdal@clemson.edu

Hongxin Hu
Clemson University
hongxih@clemson.edu

Guofei Gu
Texas A&M University
guofei@cse.tamu.edu

Gail-Joon Ahn
Arizona State University
gahn@asu.edu

Fuqiang Zhang
Clemson University
fuqianz@clemson.edu

## ABSTRACT

Traditional Network Intrusion Detection Systems (NIDSes) are generally implemented on vendor proprietary appliances or *middleboxes* with poor versatility and flexibility. Emerging Network Function Virtualization (NFV) and Software-Defined Networking (SDN) technologies can virtualize NIDSes and elastically scale them to deal with attack traffic variations. However, such an elasticity feature must not come at the cost of decreased detection effectiveness and expensive provisioning. In this paper, we propose an innovative NIDS architecture, vNIDS, to enable safe and efficient virtualization of NIDSes. vNIDS addresses two key challenges with respect to *effective intrusion detection* and *non-monolithic NIDS provisioning* in virtualizing NIDSes. The former challenge is addressed by detection state sharing while minimizing the sharing overhead in virtualized environments. In particular, *static program analysis* is employed to determine which detection states need to be shared. vNIDS addresses the latter challenge by provisioning virtual NIDSes as microservices and employing *program slicing* to partition the detection logic programs so that they can be executed by each microservice separately. We implement a prototype of vNIDS to demonstrate the feasibility of our approach. Our evaluation results show that vNIDS could offer both effective intrusion detection and efficient provisioning for NIDS virtualization.

## CCS CONCEPTS

• **Security and privacy → Intrusion/anomaly detection and malware mitigation**; **Intrusion detection systems**; *Systems security*; *Virtualization and security*;

## KEYWORDS

Network Intrusion Detection Systems; Network Function Virtualization; Software-Defined Networking


*CCS '18, October 15–19, 2018, Toronto, ON, Canada*

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243862

**ACM Reference Format:**
Hongda Li, Hongxin Hu, Guofei Gu, Gail-Joon Ahn, and Fuqiang Zhang. 2018. vNIDS: Towards Elastic Security with Safe and Efficient Virtualization of Network Intrusion Detection Systems . In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243862

## 1 INTRODUCTION

Network Intrusion Detection System (NIDS) is a critical network security function that is designed to monitor the traffic in a network to detect malicious activities or security policy violations. Recently, lots of networks have reached the throughput of 100 Gbps [83]. To keep up with the pace of the soaring throughput of networks, multi-thread approaches [35, 77] have been proposed to build NIDSes to meet the high throughput requirement for detecting attacks. In addition, some NIDSes, such as Bro [67], can be deployed as NIDS clusters [81] that spread detection tasks across multiple nodes. However, despite their usefulness in addressing scalability issues for NIDSes, both multi-thread and cluster solutions remain limited in *flexibility* regarding the processing capacity and placement location. In particular, they are still inflexible to detect attacks when a significant workload spike happens. For example, a massive attack like DDoS could bring the network traffic volume up to 500 GBps [42], which requires the NIDSes scaling accordingly to process the peak traffic load. Moreover, these approaches are also inflexible to protect current prevailing *virtualized* environments, because the perimeters of networks in virtualized environments become blur and fluid, where applications may migrate from one physical machine to another within a data center or even across multiple data centers for the purpose of flexible resource management and optimization [31]. Therefore, improving the design of current NIDSes to make them well suited to provide flexible network intrusion detection is urgent and relevant.

Network Function Virtualization (NFV) [2] and Software-Defined Networking (SDN) [59] are two emerging networking paradigms that are able to facilitate the design of *elastic security* approaches [36, 42, 85] to securing networks. Elastic security features not only scalable but also flexible network security functions. NFV implements network security functions as software instances that can be created or destroyed rapidly to handle attack traffic volume variations on demand. SDN, recognized as complementary technology to NFV [45, 71], can dynamically redistribute the traffic to support

flexible placement of network security functions. Given these benefits, there is a body of pioneer work devoting to developing elastic security solutions leveraging NFV and SDN – Bohatei [42] for flexible and elastic DDoS defense, PSI [85] for precise security instrumentation for enterprise networks, and VFW Controller [36] for elasticity control of virtual firewalls. However, rare effort is made to improve the design of current NIDSes to make full use of those two advanced paradigms.

In this work, our goal is to make a step towards elastic security through NIDS virtualization that overcomes the inflexibility of current NIDS architectures. The virtualization of NIDSes must be *safe* and *efficient*. The *safe* virtualization requires that virtualized NIDSes do not miss any potential attacks that can be detected by traditional NIDSes. The *efficient* virtualization requires that virtualized NIDSes are provisioned optimally and consume minimum resources. However, as attack postures have been shifting from per-host and byte-level exploits to multi-stage and sophisticated penetrations (e.g., Advanced Persistent Threats [78]) for the past decades, detection strategies also have been evolving from conceptually simple per-flow signature matching (e.g., Snort [20] and Suricata [21]) to more complex multi-flow information and stateful protocol analysis (e.g., Bro [22] and Libnids [5]). The complexity of multi-flow information and stateful protocol analysis makes *safe* and *efficient* virtualization of NIDSes extremely challenging.

- **Effective Intrusion Detection:** NFV enables flexible provisioning an NIDS as multiple instances that could run at different locations, and SDN can dynamically distribute network traffic to each instance. Some detection logics in NIDSes must take account of multiple network flows to identify malicious activities. For example, a scanner detector [17] must maintain a counter to count the number of flows generated by the same host to determine whether a host is a scanner. However, each NIDS instance may receive only a part of flows originated from the same scanner and maintain a counter individually. To ensure intrusion detection effectiveness, NIDS instances must share detection states with each other. Traditional NIDSes are usually provisioned with a fixed number of instances at a fixed location. This static setup mitigates the difficulty of detection state sharing between NIDS instances. In contrast, virtualized NIDSes pursue to place their instances flexibly at different locations and with different numbers. This flexibility feature makes sharing detection states among virtualized NIDS instances extremely costly and difficult. Therefore, ensuring the effectiveness of intrusion detection for virtualized NIDSes becomes more challenging than traditional NIDSes.
- **Non-monolithic NIDS Provisioning:** Existing virtualization solutions enabling elastic security [36, 85] consider a virtualized network function as a *monolithic* piece of software running in a virtual machine or container. However, simply provisioning all the components of virtualized NIDSes within a virtual machine or container is not efficient. First, since a monolithically virtualized NIDS requires more resources than any one of its components, the virtual machine that runs a monolithically virtualized NIDS should be allocated with more resources than the one that runs only some components. Second, monolithically virtualized NIDSes lack the ability to scale each component individually, thus have to scale out the entire instance even though only one component is overloaded, resulting in over-provisioning of other components. Third, it is difficult to customize a virtualized NIDS if it is provisioned as a monolith. However, NIDS customization is critical in terms of resource efficiency for advanced network attack defense [42, 85, 86]. Due to the complexity of modern detection logics, decoupling NIDSes as non-monolithic pieces that are suitable to be deployed in virtualized environments remains challenging.

In this paper, we propose a novel NIDS architecture, vNIDS, which enables safe and efficient virtualization of NIDSes. To address the *effective intrusion detection* challenge, we classify detection states of virtualized NIDSes into *local* and *global* detection states to minimize the number of detection states shared between instances. The *local* detection states are only accessed within a single instance, while the *global* detection states can be shared among multiple instances. In particular, we provide a guideline to classify detection states, and employ static program analysis and NIDS domain-specific knowledge to automatically achieve detection state classification. To address the *non-monolithic NIDS provisioning* challenge, virtualized NIDSes are provisioned as microservices [38]. We decompose NIDSes into three microservices, *header-based detection*, *protocol parse*, and *payload-based detection*. Each microservice can be instantiated independently, provisioned at different locations with a different number of instances, and configured to execute different detection programs. Multiple microservices can be chained up in different ways as microservice chains that provide flexible detection capabilities. In particular, we design an algorithm that leverages program slicing technique to automatically partition a detection logic so that the detection logic can be transparently executed by microservices. We design and implement a prototype of vNIDS, which consists of an Effective Intrusion Detection module, a Non-monolithic NIDS Provisioning module, and three microservices. Evaluations of our virtualized NIDS conducted on CloudLab [10] show that our virtualized NIDS are more flexible than traditional NIDSes regarding processing capacity and placement location. To the best of our knowledge, our work presents the first solution to enable safe and efficient virtualization of NIDSes.

The rest of the paper is organized as follows. § 2 presents the motivation of this work. § 3 gives an overview of vNIDS architecture. § 4 discusses the state management approaches to ensure the detection effectiveness as well as minimizing the performance overhead. § 5 presents how to decouple monolithic NIDS into microservices to achieve the efficient provisioning. Implementation and evaluation of vNIDS are presented in § 6 and § 7, respectively. Discussion and related work are addressed in § 8 and § 9, respectively. Finally, we conclude in § 10.

## 2 MOTIVATION

NFV and SDN introduce significant flexibility to the deployment of virtualized NIDSes. However, at the same time, such a flexibility feature can potentially compromise the detection effectiveness

of virtualized NIDSes and also requires an efficient way to provision the virtualized NIDSes. We articulate the challenges of NIDS virtualization in detail as follows.

## 2.1 Effective Intrusion Detection

Virtualized NIDSes have the ability to instantiate multiple instances running on different virtual machines and dynamically scale by destroying or creating instances. Each instance takes a part of traffic and maintains its own detection states. If the traffic is delivered to an instance that lacks the required detection states, the virtualized NIDS may miss some attacks. For example, a scanner detector usually maintains a counter to count how many flows are generated by each host. If a flow is delivered to an instance that does not maintain its counter, this flow may be overlooked. For detectors that organize detection states based on the IP address, port number, or protocol type, it tempts to distribute the traffic based on those fields to ensure detection effectiveness. The authors of [35] propose a concurrency model for NIDSes that ensures detection effectiveness by carefully distributing the network traffic. However, for detectors that maintain detection states based on complex data structures or extra information that is not available to the traffic distributor (usually a network forwarding device), it is impractical to ensure the detection effectiveness only relying on traffic distribution.

To see the problem, let us consider a cookie hijacking attack [29]. The goal of the attacker is to capture an HTTP session cookie of a victim and reuse that session. There are off-the-shelf tools, such as Firesheep [1], implementing this attack as a Firefox extension, which makes this attack accessible to the masses. Fortunately, there is a detector [6] having been developed upon Bro to handle this attack. To detect the cookie hijacking attack, the NIDS must determine whether a cookie is reused by different users. Basically, the detector tracks the IP address for every cookie usage. If a cookie is used by different IP addresses, it means the cookie has been reused by different users. To reduce the false positive rate due to dynamic IP allocation, the detector also makes use of MAC addresses. As a result, only when both source IP and MAC addresses associated to the same cookie appear inconsistency, will this cookie be considered to be reused [1]. Based on this detection logic, flows carrying the same cookie should always be delivered to the same instance to ensure the detection effectiveness. However, the cookie information is not directly available to the network forwarding devices. Moreover, the complex data structure queried jointly by cookie, IP, and MAC addresses turns correct traffic distribution into even a harder problem. A more detailed demonstration of the difficulty of correctly distributing traffic to achieve an effective detection for this example is given in Appendix A. As an alternative, sharing detection states among NIDS instances is often used by multi-thread and clustered NIDSes [77, 81] to ensure detection effectiveness.

Traditionally, NIDSes are deployed at fixed locations and with fixed numbers of instances. This invariant configuration significantly mitigates the overhead introduced by detection state sharing. Being located within the same physical machine, NIDS instances can easily exchange information through memory sharing; or being clustered in several racks connected via high-speed cables, NIDS instances can share detection states cheaply. In contrast, virtualized NIDSes feature flexibility in terms of the placement location and number of instances. The exact location and number of instances for virtualized NIDSes are unpredictable. Thus, the overhead to enable detection state sharing among virtualized NIDS instances is significant. This makes it challenging in practice to share detection states among virtualized NIDS instances. Therefore, for virtualized NIDSes, it is especially critical to minimize the number of detection states that need to be shared among instances. To facilitate the development of virtualized NIDSes, systematic analysis and classification of detection states for NIDSes to minimize the detection state sharing is appealing.

However, rare literature from the security research community provides a systematic approach to automatically analyze and classify the detection states to achieve effective intrusion detection and minimize the detection states sharing. NIDS Cluster [81] enables detection state sharing among NIDS nodes of a cluster. However, it fails to provide a guideline to analyze and classify the detection states and relies on users to determine *manually* which detection states need to be shared. The multi-thread NIDS [77] ensures detection effectiveness by enabling inter-thread communication. But still, it fails to give analyses to detection states to minimize the communication. In the networking research community, there are two major research directions on NFV state management. One direction focuses on the state migration problem during instance live migration, such as Split/Merge [70] and OpenNF [46]. These proposals maintain states in each instance separately during runtime. As we discussed, without detection state sharing, it is impractical to achieve effective intrusion detection. The other direction focuses on state sharing between instances, such as StatelessNF [51] and S6 [84]. The former shares all states between instances via a shared data store, and the latter proposes to only share partial states to reduce performance overhead. However, both works fail to present a systematic approach to analyzing and classifying states.

## 2.2 Non-monolithic NIDS Provisioning

Existing elastic security solutions [36, 42, 85] consider a virtualized network function as a monolith directly running in a virtual machine or container. However, provisioning virtualized NIDSes as a monolithic piece of software running as a whole has significant limitations to protect the virtualized environments. In this section, we discuss a few facts to highlight the limitations of provisioning virtualized NIDSes as monoliths and then identify two challenges to enable non-monolithic provisioning.

**Inefficient Scaling:** Dynamic scaling is an important feature of virtualized NIDSes. However, if not provisioned carefully, this dynamic can turn into the weakness of virtualized NIDSes. We show two cases where virtualized NIDSes are prone to inefficient scaling if provisioned as monoliths.

- *Case1:* During scaling, virtualized NIDS instances must be suspended and the traffic should be buffered [46]. The adversaries can maliciously change the traffic volume to trigger the virtualized NIDS scale-in/-out frequently. This can potentially cause inefficient scaling. If the virtualized NIDS is provisioned as a

[1]We notice that this detector has some limitations that can be exploited by attackers to bypass it. However, improving the robustness of existing detectors is not the focus of this paper.

monolith, it takes more time to launch a new instance and move states between instances, which worsens the situation.

- *Case2:* In practice, it is not likely that all the components of a virtualized NIDS are overloaded at the same time. The bottleneck component usually gets overloaded first. If the virtualized NIDS is provisioned as a monolith, the whole virtualized NIDS must scale-out when only a single component is overloaded. This scaling strategy results in over-provisioning to other components. Moreover, by overwhelming only a single component, adversaries can force the entire system to scale-out, which may exhaust the resources in the environment.

**Inefficient Resource Allocation:** Monolithic NIDSes require more resources than any components of NIDSes to run. For example, an NIDS consists of 4 components and each component requires 1 CPU core to run. If the NIDS is provisioned with a single virtual machine, one should assign 4 CPU cores to run the virtual machine. In this case, if the physical machine only has 3 CPU cores available, there is no way for the NIDS to make use of these cores. In contrast, if NIDSes are decomposed and each component is provisioned with a virtual machine separately, three components of the NIDS will be able to make use of the 3 CPU cores. Unlike traditional NIDSes that are deployed statically, virtualized NIDSes may allocate and deallocate resources quite frequently to achieve better flexibility in capacity and location. Therefore, monolithic provisioning has a greater negative impact on virtualized NIDSes than traditional ones.

**Difficulty of Customization:** Customization of virtualized NIDSes is important in many circumstances. For example, as found in [39], Bro's resource usage almost linearly scales with the number of its analyzers. Therefore, some analyzers in Bro could be turned off to save resources when they are not necessary to run at specific time. Other scenarios that require customization include applications to the networks that require context-ware detections and defenses [42, 85, 86]. Moreover, as the emergence of edge clouds (e.g., Cloudlet [75], fog cloud [28], mobile edge cloud [66], CORD [68], etc.), virtualized environments get increasingly diverse. Those heterogeneous edged, virtualized environments are not as powerful as traditional clouds, so virtualized NIDSes need to be significantly customized to protect those environments. However, customization cannot be accomplished easily and efficiently with monolithic NIDSes. Because it requires to rebuild and re-deploy the whole system if any single component of the system has been modified.

**Challenges for Non-monolithic Provisioning:** Existing approaches have discussed partitioning a single application into small pieces that can be deployed separately in other areas, such as web applications [33] and mobile applications [34]. The networking community has also contributed literature that proposes to deploy network functions in pieces [30, 52]. However, no existing work has presented how to deploy *security-specific* functions, such as NIDSes, in a non-monolithic way in virtualized environments. Two challenges remain to enable non-monolithic provisioning for virtualized NIDSes.

- **Ch1:** *How can we decompose monolithic NIDSes into several separately deployable and smaller pieces?* We should decompose the NIDSes in a way that they can still perform efficiently with respect to intrusion detection.

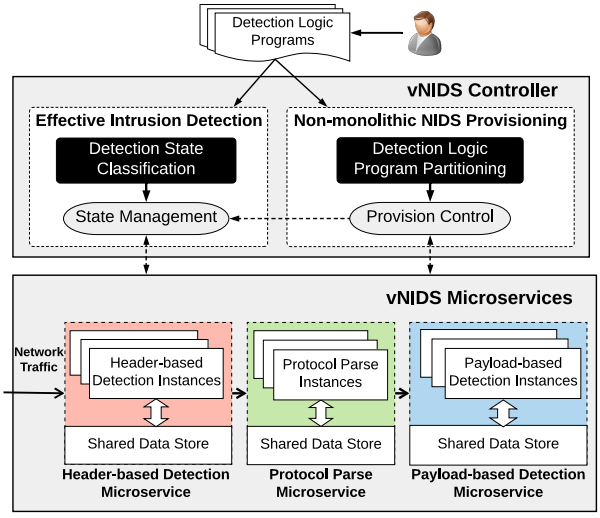


**Figure 1: High-level view of vNIDS architecture.**

- **Ch2:** *How can we enforce the detection policies with NIDSes provisioned in a non-monolithic way?* How to deploy NIDSes should be transparent to the users. That is, from users' perspective, they should be able to implement detection logics as they are using monolithic NIDSes.

## 3 OVERVIEW

In this section, we outline the key ideas of vNIDS architecture and then provide a high-level view of the vNIDS architecture.

### 3.1 Key Ideas

To ensure effective intrusion detection (**§ 4**), the traffic is distributed to different instances only based on its header information. Then, we employ static program analysis to classify the detection states of virtualized NIDSes into *global* and *local* detection states. The *global* detection states are shared by all instances to ensure that every instance has sufficient detection states for intrusion detection. The *local* detection states are maintained by each instance respectively to mitigate the performance overhead. The challenge for detection states classification is how to minimize the performance overhead without compromising the intrusion effectiveness. Our detection state classification addresses the challenge by identifying as many *local* detection states as possible. Only those detection states that cannot be guaranteed to be accessed by a single instance are classified as *global* detection states.

To achieve non-monolithic provisioning (**§ 5**), we in particular address two challenges, **Ch1** and **Ch2**. We address **Ch1** by identifying two partition points that decompose NIDSes with three microservices: *header-based detection*, *protocol parse*, and *payload-based detection*. Users implement their detection logic as Detection Logic Programs (DLPs), which are executed by the NIDS Engine (NE). NE is a piece of software that acquires traffic from the network (e.g., reading the packets, checking the checksum, reassembling IP fragments, etc.) and executes various DLPs. Each microservice acts as an NE executing specific DLPs. We address **Ch2** by

designing an algorithm to automatically partition a DLP into two small DLPs that can be executed by the *header-based detection* and *payload-based detection* microservices, respectively.

## 3.2 High-level View of vNIDS

Our design of vNIDS is demonstrated in Figure 1. vNIDS consists of *vNIDS controller* and *vNIDS microservice instances*. The users interact with vNIDS through the controller. The microservice instances are responsible for processing the traffic.

**vNIDS controller:** There are two major modules in the vNIDS controller: the Effective Intrusion Detection module (EIDM) and Non-monolithic NIDS Provisioning module (NNPM). EIDM implements a detection state classification engine, which is run off-line to classify the detection states of DLPs into *global* and *local* detection states. The detection state taxonomy is sent to the State Management component (SMC). The SMC interacts with the shared data store and is responsible for managing the detection states of instances. If the virtualized NIDSes scale-in/-out, the SMC should also address the state migration. NNPM implements the DLP partitioning algorithm, which is run off-line to partition a DLP into header-based and payload-based DLPs. The Provision Control component (PCC) in the NNPM takes charge of microservice instance provisioning, installs the DLPs into the instances, and demands the SDN controller to update the flow rules in network forwarding devices accordingly. Each time the PCC creates or destroys instances, it should inform the SMC to update the shared data store.

**vNIDS microservices:** There are three microservices, which are *header-based detection*, *protocol parse*, and *payload-based detection*. Different microservices can be chained up as microservice chains. Each microservice in a microservice chain only executes a part of the detection logic of the chain. The results of a microservice can be attached to the traffic and passed to the next microservice. This design enables more fine-grained customization of virtualized NIDSes, since the detection logic is composable and each microservice can be provisioned independently. In addition, different microservices in a microservice chain can be instantiated with different numbers of instances. Instances instantiated from the same microservice share their *global* states through the shared data store.

Provisioning virtualized NIDSes with microservices is more efficient from the following perspectives. First, microservices are smaller than monolithic NIDSes and can be provisioned separately. Therefore, microservices can scale faster and independently. Second, since microservices require less resource to instantiate, they can make better use of fractional resource than monolithic NIDSes. Third, a microservice can be modified and updated without interfering other microservices. In addition, different microservices can be instantiated at different locations (for resource optimization [31]) and provisioned on demand based on different security contexts [85].

# 4 EFFECTIVE INTRUSION DETECTION

In this section, we demonstrate how to classify the detection states based on the concept of *detection state scope* to achieve effective intrusion detection while minimizing the number of detection states that need to be shared. We first introduce the scope of detection states. Then, we present how to infer the scope of detection states by analyzing the *control-flow* of the Detection Logic Program (DLP) and NIDS Engine (NE), and classify the detection states based on their scopes. Finally, we address the effective intrusion detection challenge in NIDS virtualization with minimum detection state sharing.

## 4.1 Detection State Scope Analysis

We define the scope of a detection state by its lifetime. Detection states can only influence detection results when the states are "alive". That is, detection states cannot make any difference to the detection results before they are created or after they are destroyed. Therefore, by estimating the lifetime of the value of a variable, we can determine the scope of a detection state stored by that variable.

To understand the lifetime of the value of a variable, let's start with the discussion of the design pattern of NIDSes. Generally, NIDSes first acquire a packet from the network, then process the packet according to various detection logic, and finally destroy the packet. NIDSes perform detections by iterating the *acquire-process-destroy* procedure. We call the *acquire-process-destroy* procedure as an *iteration* of packet processing. The acquisition and destroy of packets are the basic functionalities of most network functions. Therefore, implementations of those functionalities are relatively stable and not likely to be changed. We call these functionalities as NE. The processing of packets varies for different detection purposes. These functionalities are defined by the NIDS users through DLPs. Some NIDSes, such as Libnids [5], provide the users with callback functions to implement and register their DLPs. Modern NIDSes, such as Bro [22], provide more user-friendly languages, such as Bro scripting language [8], for the users to write the DLPs. Those DLPs are then translated into low-level languages, and compiled with the NE. The recent work [35] has demonstrated the feasibility to represent DLPs of Bro through an intermediate assembly-like language and conduct program analysis based on that. The lifetime of the value of a variable is implied by when the memory location of that variable is freed. It can be freed for every iteration of packet processing, or freed after multiple iterations of packet processing.

Based on how a detection state is used, the scope of the detection state stored by a variable can be defined in three levels.

- *Per-packet* scope. We call detection states with this scope *per-packet* detection states. The *per-packet* detection states are only utilized within a single iteration of packet processing, thus are created and destroyed within a single iteration of packet processing. For example, a variable used to compute the checksum of a packet is considered as a *per-packet* detection state.
- *Per-flow* scope. We call detection states with this scope *per-flow* detection states. The *per-flow* detection states are utilized by multiple packets from the same flow. Therefore, these states persist beyond a single iteration of packet processing. However, these states must be created after a flow initiates and destroyed before a flow terminates. Otherwise, there are memory leaks [2], since other flows can never access theses detection states. For example, a variable used to count the bytes of a flow is considered as a *per-flow* detection state.

[2] An allocated memory location that will never be freed causes a memory leak.

- *Multi-flow* scope. We call detection states with this scope *multi-flow* detection states. The lifetime of *multi-flow* detection states persists beyond the duration of a single flow. Therefore, they are created before a flow initiates or destroyed after a flow terminates. For example, a variable used to count how many IP addresses have been scanned by a scanner is considered as a *multi-flow* detection state.

## 4.2 Detection State Classification

Since we notice that the lifetime of the value of a variable determines the scope of a detection state, our analysis mainly focuses on the *destroy* statements of variables. Variables in the heap are allocated by *malloc* and destroyed by *free*, while variables in the stack are allocated by *call* (invoking a procedure) and destroyed by *return* (returning from a procedure).

According to the detection state scope analysis presented in § 4.1, we can infer the scope of a detection state by checking whether the detection state is "always" destroyed before a flow terminates. If the detection state is not "always" destroyed before a flow terminates, it means this detection state must be destroyed sometime later triggered by other flows. In other words, this detection state is possibly utilized by multiple flows. Therefore, this detection state has the *multi-flow* scope. In contrast, detection states that are "always" destroyed before a flow terminates should be utilized by a single flow only, otherwise, other flows may end up with referring to a destroyed state causing a runtime error. Those states have the *per-flow* scope. Furthermore, if a detection state is "always" destroyed before the end of the iteration of the packet processing, this detection state has the *per-packet* scope.

Based on the above principle, our major task is to figure out the *execution sequence* of the destroy statement (which terminates the lifetime of a variable) of each variable, and then we can infer the scope of a detection state by checking when the variable of that detection state is destroyed. In particular, we compare whether the destroy statement of a variable is "always" executed before the destroy statement of the variable storing a packet or the destroy statement of the variable storing a flow record. We find that IP reassembly is an essential functionality for NIDS, and we can also identify the variable that stores a packet (it is a C structure in BSD and Linux systems) or stores a flow record (it is a pointer to a doubly linked list in BSD and Linux systems) from the IP reassembly implementation.

In order to define the execution sequence of two statements in a program, we introduce two techniques: 1) Control Flow Graph (CFG) [44] analysis; and 2) dominator finding [55]. The node in a CFG of a program represents a block of statements in the program and the directed edge represents a jump between blocks. The CFG of a program depicts the execution sequence of each statement. The concept of dominator is proposed by the authors of [57]. This concept is originally used in the graph theory for program optimization. There are algorithms to efficiently compute the dominators of a given node in a flowgraph [55]. We use the dominators of a statement $P$ in the CFG to imply the statements that are "always" executed before $P$, because by definition, the dominator of a statement $P$ in the CFG is a statement that appears on every path in the CFG from the entry statement of the program to the target statement $P$. This key insight provides us with the opportunity to determine whether the destroy statement of a variable is "always" executed before the destroy statement of another variable.

We explain in detail the major four steps of our algorithm that infers the scope of detection states as follows:

*Step1: computing the CFG of the DLP and NE.* Every time the NIDS receives a packet, it starts from a packet receive function call (e.g, *pcap_next* or *recv*) and ends at the destroy statement of the packet. We, therefore, consider a packet receive function call as the entry statement for the CFG.

*Step2: identifying the packet and flow record destroy statements.* We manually check the code of IP/TCP reassembly implementation, which is available as standard libraries in many operating systems. For any specific NE, once identified, this information can be reused by all DLPs.

*Step3: computing dominators of the packet and flow record destroy statements.* Computing dominators of a given node in a CFG is a mature technique [55] in the graph theory domain.

*Step4: determining the scope of each detection state.* There are three cases of the destroy statement of a variable. Different cases indicate different scopes. (1) If the destroy statement of a variable is a dominator of the packet destroy statement, this variable is a *per-packet* detection state. (2) If the destroy statement of a variable is a dominator of the flow record destroy statement, this variable is a *per-flow* detection state. (3) If the destroy statement of a variable is neither a dominator of the packet destroy statement nor a dominator of the flow record destroy statement, this variable is a *multi-flow* detection state.

After we infer the scope of each detection state by analyzing the CFG of the DLP and NE, we define the *per-packet* and *per-flow* detection states as *local* detection states and define *multi-flow* detection states as *global* detection states. We justify this definition in the next section. An example of using our approach to classify the detection states for a scanner DLP [17] is presented in Appendix B.

## 4.3 Ensuring Effective Intrusion Detection

To ensure effective intrusion detection, we must ensure that all the traffic under process can access to all the detection states relevant to the traffic. We achieve this goal by 1) distributing the traffic of the same flow to the same instance, and 2) enabling *only global* detection states shared by multiple instances.

Since advanced network forwarding devices now have programmability thanks to SDN, dynamically redistributing traffic is feasible. When virtualized NIDSes dynamically scale (by creating or destroying instances), network traffic can be redistributed among the instances at *per-flow* granularity. That is, by leveraging SDN, we can dynamically update the forwarding rules in the network forwarding device to always deliver the traffic of the same flow to the same instance.

Actually, we are trying to make a trade-off between the complexity of distribution algorithm (i.e., how to distribute the traffic to ensure detection effectiveness) and the overhead of detection state sharing (i.e., enabling information exchange among instances). On the one hand, we utilize the flexible *per-flow* forwarding capability of SDN to simplify the traffic distribution for virtualized NIDSes.

On the other hand, the intrusion detection effectiveness is guaranteed by sharing *only multi-flow* detection states.

We can enable detection state sharing among instances by maintaining *global* detection states in the shared data store, such as RAMCloud [62], FaRM [37], and Algo-logic [4]. The authors of StatelessNF [51] have shown the feasibility of using RAMCloud as the shared data store for network functions. By classifying the detection states into *local* and *global* detection states in our approach, we only put *global* detection states on the shared data store to minimize the performance overhead. We have quantified the performance overhead in our evaluation presented in § 7.3.

## 5 NON-MONOLITHIC NIDS PROVISIONING

In this section, we first address challenge **Ch1** by decomposing virtualized NIDSes into three *microservices* based on the logic structure of NIDSes and the types of detection logic (DL). Then, we address challenge **Ch2** by designing an algorithm to partition the DLPs such that they can be enforced by non-monolithic virtualized NIDSes.

### 5.1 Decomposing NIDSes as Microservices

**Logic structure of NIDSes:** Typically, the logic structure of NIDSes consists of three major components: the network protocol stack (NPS), application protocol parser (APP), and the detection logic. NIDSes firstly acquire and validate the network traffic through the NPS. This component is responsible for reading the packet from the driver, checking the checksum, reassembling the IP fragments, etc. It is implemented in the NE. Then, the outputs of the NPS are passed to the APPs. The APPs parse the payload of the traffic and extract the semantics of the conversation between two endpoints of the network. The outputs of the APPs are then processed based on various DLs.

For the purpose of reducing the development cost while maximizing the extensibility, the NPS and APPs are usually reused by different DLs. NIDS users are allowed to add their new APPs, while NPS is not likely to be changed by the users. For example, the Bro network security monitor [22] has integrated the IP/TCP network protocol stack and a number of predefined application protocol parsers[3]. Then, users only need to program their detection logic using a scripting language [8] provided by Bro. Users can also add their own APPs through specific program languages, such as BinPAC [65]. Another example is Libnids [5], which has implemented a library for developing the NIDSes. The IP/TCP NPS has already been included in the library, but the users need to implement their own APPs and DLs through the C interface stubs. Therefore, for generality, we consider APPs as a part of NE and only DL is implemented by the DLPs.

**Types of DL:** We observed that there are three types of DLs based on what information the DL needs to conduct the detection task.

- Type-I: These DLs only need header information to complete their detection tasks. For example, the *Flow Monitor* implements the DL that only monitors the statistics of the flows, such as byte-per-flow and packet-per-flow statistics. Another example is the DL of *HTTP Monitor*, which first filters out the HTTP traffic by the port numbers and then computes the statistics of traffic generated by each Web server.
- Type-II: These DLs only need payload information to complete their detection tasks. For example, all the signature-matching DLs only need to check the payload of the packets.
- Type-III: Theses DLs need both header and payload information to complete their detection tasks. For example, the Cookie Hijacking DLP first looks at the cookie, which is payload information. Then, it checks the IP address, which is header information.

[3]In the Bro community they are called as *protocol analyzers*.

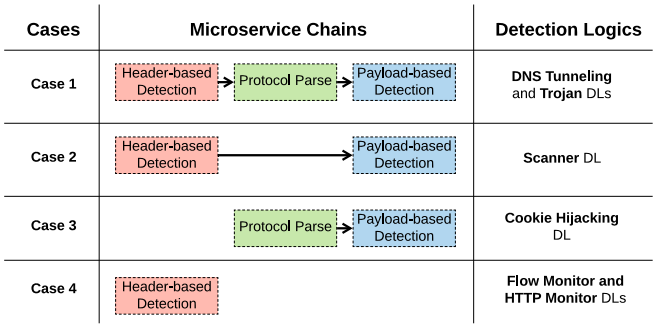


**Figure 2: Different DLs can be realized by different microservice chains.**

**Decomposing monolithic NIDSes:** Based on the logic structure of NIDSes and the types of DL, we decompose NIDSes as three microservices: *header-based detection*, *protocol parse*, and *payload-based detection*. The logic structure of NIDSes indicates three major processing of the packets. The first processing is executed by the NPS, which is the basic component of all network functions. We reuse this processing in all the microservices. The second processing is executed by APPs, which parse the traffic according to various application protocol specifications. We define this processing as a dedicated microservice called *protocol parse*, which is devoted to parsing the traffic according to various application protocol specifications. The third processing, which is specified by users, implements various DLs. Based on the types of DLs, we can actually classify all the DLPs into header-based DLPs, which implement Type-I DLs, and payload-based DLPs, which implement both Type-II and Type-III DLs. We define the *header-based detection* microservice devoted to executing the header-based DLPs, and the *payload-based detection* microservice devoted to executing the payload-based DLPs.

**Microservice chaining:** Different microservices can be chained as microservice chains. A microservice chain realizes a complete detection task, for example, detecting whether a specific host is a scanner, or whether a cookie has been reused. Each microservice in the chain takes over a part of the detection task. Intermediate results generated by the previous microservice can be encapsulated in the traffic (using some encapsulation techniques, such as FlowTags [43], NSH [69], Geneve [47], etc.) and passed to the next microservice in the chain.

We have studied six different types of DLs, which can be summarized into 4 cases when composed as microservice chains. Their DLPs are listed in Table 1 in § 6. Different DLs can be realized by different microservice chains. Figure 2 depicts that the six DLs are

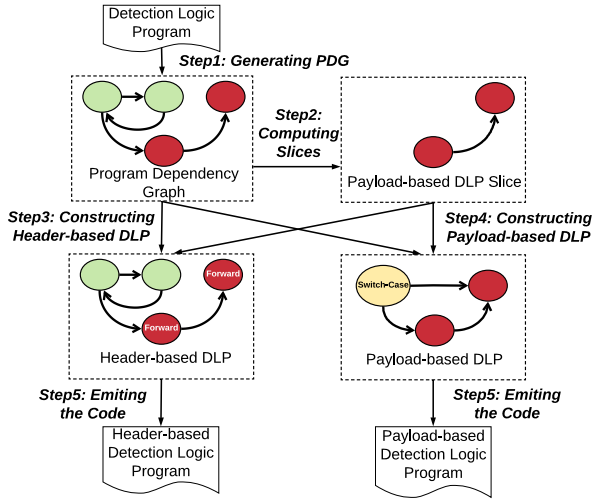


**Figure 3: Major steps of the DLP partitioning: (1) generating PDG of the DLP; (2) computing payload-based DLP slice; (3) constructing header-based DLP; (4) constructing payload-based DLP; and (5) emitting the code.**

summarized into four cases. Each case can be realized by one microservice chain. For example, the Cookie Hijacking DL (case 3) is realized by *protocol parse* and *payload-based detection* microservices, while Scanner DL (case 2) is realized by *header-based detection* and *payload-based* detection microservices.

Though microservice chains can be composed flexibly, there are still some constraints of how the microservice chains can be composed. First, the *header-based detection* microservice can be located before the *protocol parse* and *payload-based detection* microservices, because *header-based* microservice does not need payload information. Second, the *protocol parse* microservice should be located before the *payload-based detection* microservice, since the latter needs the payload information gathered by the *protocol parse* microservice.

## 5.2 Detection Logic Program Partitioning

As outlined in § 5.1, NIDSes can be decomposed into three microservices. However, it remains the challenge to partition the user-defined DLPs into header-based and payload-based DLPs that can be executed separately by the *header-based detection* and *payload-based detection* microservices. In this section, we present an algorithm that can automatically partition the user-defined DLPs by utilizing the program slicing technology [49, 72, 82].

Program slicing is a program analysis technique that can determine: *1)* which statements are influenced by a given variable at a given point (forward slicing); and *2)* which statements can influence the value of a variable at a given point (backward slicing). The inputs of the program slicing are the Program Dependency Graph (PDG), variables of interest, and the point where the variable is considered. The outputs of the program slicing are the statements of interest. PDG is a directed graph whose vertices represent statements and edges represent dependencies between statements. There are some program slicing tools available. In this work, we

**Algorithm 1:** Identifying statements of header-based DLP and variables to be tracked by header-based DLP

**Input:** $F$: a set of header fields;
$prog$: origin detection program
**Output:** header-based DLP statements and variables to be tracked

```
1  PI = {} /* a set storing inputs containing payload information */
2  foreach input in Inputs(prog) do
3      if input ∉ F then
4          PI = PI ∪ {input}
5  Slice = {} /* a set storing statements in resulted slices */
6  foreach input in PI do
7      Slice = Slice ∪ ForwardSlicing(prog, entry, input)
8  S = Statements(prog) − Slice
9  V = {} /* a set storing variables that should be tracked by
       header-based DLP. */
10 foreach v in Variables(Slice) do
11     if v ∈ Variable(S) then
12         V = V ∪ {v}
13 return S and V
```

implement our DLPs in C and utilize Frama-C [12] to conduct program slicing on the DLPs. We achieve DLP partitioning through five major steps as shown in Figure 3.

*Step1: Generating PDG.* We conduct static program analysis to generate the program dependency graph of the inputted DLP. The PDG of a program involves two dependency information between the statements of a program–the control dependency and the data dependency. The control dependency, usually represented by control flow graphs, reflects the execution conditions and sequences between statements. The data dependency reflects the declaration and reference relationships between variables. There are tools that can be used to generate the PDG of a program. For example, Frama-C [12] can generate the PDG of C program and a commercial product CodeSurfer [11] can generate the PDG of C++ program.

*Step2: Computing slices.* After we get the PDG of the DLP, we utilize program slicing tools to isolate the statements of payload-based DL from other statements. The key insight is that payload-based DL relies on payload information thus is influenced by inputs containing payload information. If an input is a direct reference of some header fields, this input only provides header information, otherwise, it also contains payload information. Based on the above insight, we compute a forward program slice from the input of the DLP for the inputs that contain payload information. we call the resulting statements as the payload-based DLP slice. All the statements in the payload-based DLP slice are affected by the payload information. Therefore, we should let the *payload-based detection* microservice execute these statements.

*Step3: Constructing header-based DLP.* Since we have isolated the statements that are influenced by payload information, the remaining statements are independent to the payload information. These remaining statements can be executed by the *header-based detection* microservice. Algorithm 1 depicts the algorithm to identify the statements of the header-based DLP. Lines 1-4 identify the inputs containing payload information. Lines 5-7 compute the payload-based DLP slice. Line 8 computes the statements of the header-based DLP.

Then, one question remains: how can we deal with the packet if the processing of the packet reaches a point where the statements in the payload-based DLP slice must be executed? Note that those statements in the payload-based DLP slice cannot be executed in the *header-based detection* microservice, since payload information is not available in that microservice. We solve the problem by introducing the forward statement. We replace all the statements in the payload-based DLP slice with the forward statement. The forward statement is actually an interface to the NE, similar to the "system call" in the operating systems. Once the packet processing reaches the forward statement, it causes a trap into the NE. Then, the NE handles the packet by encapsulating it with some metadata and sends it to the network. The metadata includes two pieces of information: *1)* the points where the execution is interrupted; and *2)* the intermediate results associated with the processing of this packet. We can get information *1)* by labeling the forward statement (e.g., using its line number). Note that there may be multiple serial forward statements, and we only label the first one. We can get information *2)* by tracking which variables have been updated by the current packet processing. Note that this information should not be too much, since we only need to track the variables that are also used by the payload-based DLP. Lines 9-12 of Algorithm 1 compute which variables should be tracked.

*Step4: Constructing payload-based DLP.* As we have already computed the payload-based DLP slice. We can use that slice as a starting point to construct the payload-based DLP. There are two major concerns with the construction: *1)* how can we get access to the intermediate results produced by the header-based DLP? and *2)* how can we retrieve the execution from the previous breakpoints? For concern *1)*, we add an initialization statement at the beginning of the payload-based DLP slice for every variable that is also used by the header-based DLP. Recall that we have a contract that the forward statement in the header-based DLP will encapsulate intermediate results of those variables used by both DLPs. As a result, those variables are initialized at the beginning of the payload-based DLP slice. For concern *2)*, we add a switch-case statement right after all the initialization statements. The switch-case statement checks the label that is set by the forward statement in the header-based DLP. According to the label, the switch-case statement jumps to the right points to start execution.

*Step5: Emitting the code.* This step is a reverse of *Step1*. Once we have constructed the PDG of header-based and payload-based DLPs, there is little challenge to emit the code for both DLPs. Many program analysis tools also support the reversing procedure to emit the code from a computed PDG.

We provide an example of partitioning the DNS Tunneling DLP in Appendix C.

## 6 IMPLEMENTATION

We have implemented a prototype of vNIDS based on Xen [25] hypervisor. The vNIDS controller is implemented in the *Dom0* of Xen, which has the ability to monitor and manage the virtual machines provisioned in Xen. We utilized Floodlight [18], an open-source SDN controller, and Open vSwitch [16] to construct our SDN environment. In particular, the vNIDS controller interacts with Floodlight to achieve per-flow traffic distribution. The three microservices are implemented based on the Click modular router software [54]. Click provides rich networking processing elements, which can be leveraged to construct a wide range of network functions.

| Detection Programs | Descriptions |
|---|---|
| DNS Tunneling DLP [7] | Detects potential DNS tunnels based on the DNS request frequency, packet length and domain name characteristics. |
| Cookie Hijacking DLP [6] | Detects whether an HTTP cookie has been reused by different users. |
| Trojan DLP [35] | Combines SSH, HTTP and FTP traffic to detect whether a host has been infected by Trojans. |
| Scanner DLP [17] | Detects whether a host has generated a large amount of small flows to different hosts in a short period of time. And searches signatures in the flows generated by scanners. |
| Flow Monitor DLP | Monitors the bytes, packets, and average byte per packet information of each flow. |
| HTTP Monitor DLP | Monitors the data being transferred through HTTP traffic and sort the hosts based on the volume of HTTP traffic. |

**Table 1: Detection programs that we have written for our virtualized NIDS.**

**vNIDS controller:** We have implemented the DLP partitioning algorithms and detection state classification engine based on Frama-C [12], a static program analysis tool framework for C. The DLP partitioning algorithms and detection state classification engine are run off-line. Each time the user specifies new DLPs, the DLP partitioning algorithms will be run to partition the DLP into two header-based and payload-based DLPs that can be installed into the *header-based detection* and *payload-based detection* microservices, respectively. The detection state classification engine takes the two small DLPs and outputs DLPs with classified detection sates. We have developed a programming interface on top of XL [26] to support provision control and state management.

**vNIDS microservice instances:** We have developed three new elements for Click to enable the three microservices. The major functionalities of the three microservice include *1)* handling the messages from the controller, *2)* integrating RAMCloud [75] to support detection state sharing between instances, and *3)* executing DLPs or protocol parsers (we have implemented HTTP, SSH, DNS, and FTP protocol parsers for our virtualized NIDS). We have written six DLPs as Click elements for various detection purposes. Table 1 shows the descriptions of the six DLPs that we have written for our virtualized NIDS. In particular, *DNS Tunneling DLP*, *Cookie Hijacking DLP*, *Trojan DLP*, and *Scanner DLP* are representatives of the DLPs handling traffic of different application protocols. *Flow Monitor DLP* and *HTTP Monitor DLP* are commonly used traffic monitors that collect traffic statistics and conduct some detection tasks based on the statistics.

## 7 EVALUATION

In this section, we evaluate our approach with the following major goals:

- Demonstrating the intrusion detection effectiveness of vNIDS. We run our virtualized NIDS and compare its detection results

with those generated by *Bro* NIDS based on multiple real-world traffic traces (Figure 4).
- Evaluating the performance overhead of detection state sharing among instances in different scenarios: 1) without detection state sharing; 2) sharing all detection states; and 3) only sharing *global* detection states. The results are shown in Figure 5. The statistics of *global* states, *local* states, and forward statements are shown in Table 2.
- Evaluating how fast each microservice can execute and scale compared with monolithically provisioned NIDSes. We compare the processing capacity and launch time of each microservice with those of the monolithically virtualized NIDS (Figure 6).
- Demonstrating the flexibility of vNIDS regarding placement location. In particular, we quantify the communication overhead between virtualized NIDS instances across different data centers that are geographically distributed (Figure 8).
- Demonstrating the flexibility of vNIDS regarding processing capacity. We compare vNIDS with Bro Cluster with respect to processing capacity and resource usage when the network traffic volume is changed (Figure 9).

## 7.1 Data Collection and Attack Trace Generation

We conducted our experiments on CloudLab [10], an open platform that provides various resources for the experimenters to build clouds by running cloud software stacks, such as CloudStack and OpenStack. In the experiments, we built a cloud environment based on Xen. The physical servers we employed have two Intel E5-2630 v3 8-core CPUs at 2.4 GHz and with two 10Gb network interface cards.

Our dataset consists of the background traffic and the labeled attack traffic. The background traffic is from three different sources: *1)* the CAIDA UCSD anonymized Internet trace [3], which is a representative of Internet traffic; *2)* the LBNL/ICSI enterprise trace [14], which is a typical traffic trace collected from an enterprise network; and *2)* the campus network trace that is collected from our campus network gateway. The labeled attack traffic is generated by conducting penetration tests in an isolated environment. We generated four attack traces for our DLPs.

- The DNS.trace contains DNS tunneling traffic generated by 20 different hosts. Therefore, we counted the number of "malicious activity" in this trace as 20. We used scapy [19], a powerful interactive packet manipulation program, to generate the traffic.
- The Cookie.trace contains HTTP traffic with 100 different cookies that have been reused by multiple hosts. Therefore, we counted the number of "malicious activity" in this trace as 100. We used Firesheep [1], an extension of FireFox, to generate the traffic.
- The Trojan.trace contains 20 victim hosts intruded by our penetration. The penetration follows the description in [35]. Basically, an attacker connects to the victim through SSH; then, the attacker downloads a ZIP toolkit from a website; finally, the attacker escalates the privilege and uploads a ZIP data file to a remote FTP server. Therefore, we counted the number of "malicious activity" in this trace as 20.
- The Scanner.trace contains the traffic from 20 hosts that try to scan other hosts in the network. Therefore we counted the number of "malicious activity" in this trace as 20. We used Nmap [15] to generate the scanning traffic.

We merged the above four traces as the Attack.trace. Then, we merged the Attack.trace with three different real-world background traffic as CAIDA+Attack.trace, LBNL+Attack.trace, and Campus+Attack.trace. The merged traces are then replayed to the NIDS. We observed the output logs to determine how many attacks have been detected by the NIDS.

## 7.2 Detection Effectiveness of vNIDS

In this experiment, we evaluated the detection effectiveness of vNIDS. Our evaluation is based on the DNS Tunneling, Cookie Hijacking, Trojan, and Scanner DLPs.

For each dataset, we compared the detection results in four scenarios:

- We used the outputs of Bro NIDS as our baseline for the experiment results (*Bro*).
- We implemented a version of the four DLPs that share all the detection states, and used them in our virtualized NIDS. Each DLP was run with multiple instances (*Share All*).
- We ran the detection state classification component in the vNIDS controller to generate another version of the four detection programs that enable state sharing between instances. Each DLP is run with multiple instances (*vNIDS*).
- We implemented a version of the four DLPs without state sharing between instances, and directly ran each DLP with multiple instances (*No Sharing*).

In our experiments, we first ran the *Bro* NIDS over the three datasets and took the outputs of Bro as the baseline for the detection results. We didn't find DNS, Cookie, or Trojan malicious activities in any of the three real-world traces. The malicious activities detected by these DLPs are all from our Attack.trace. We found malicious activities for Scanner DLP in all three real-world traces. The malicious activities detected by the Scanner DLP are from real-world traces and our Attack.trace.

Then, we ran the *Share All* NIDS over the three datasets. The traffic is distributed to two instances randomly at per-flow granularity. The number of the malicious activities reported by the NIDS is the same as *Bro* NIDS. This implies that, by sharing all detection states, our virtualized NIDS can ensure the effectiveness.

Next, we ran the *vNIDS* over the three datasets. The traffic is distributed to two instances randomly at per-flow granularity. We also observed the same number of malicious activities reported by the NIDS. This implies that vNIDS can also ensure the detection effectiveness by only sharing the *global* detection states.

Finally, to demonstrate the limitations of NIDSes without sharing detection states, we ran the *No Sharing* NIDS. The traffic is distributed to two instances randomly at per-flow granularity. At this time, we observed a degradation of detection rate for all the three datasets. The comparison of detection rates of NIDSes in four scenarios is shown in Figure 4. Figures 4-a, 4-b, and 4-c are based on the three traces, CAIDA+Attack.trace, LBNL+Attack.trace and Campus+Attack.trace, respectively.

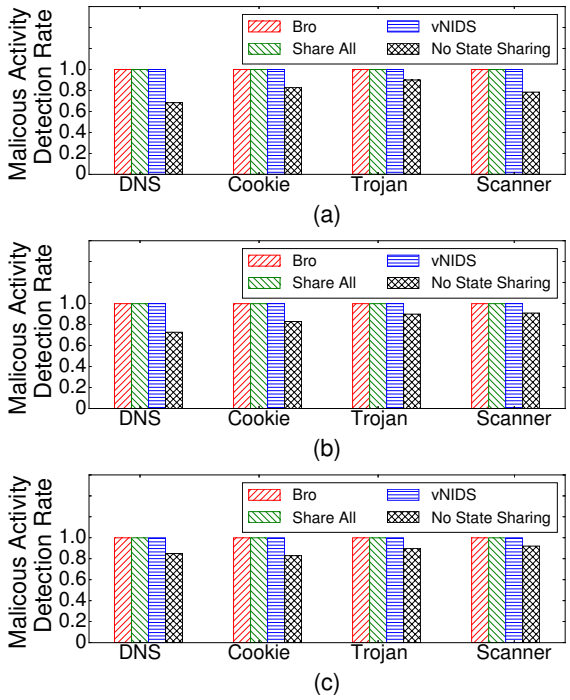


**Figure 4: Detection rate of known malicious activities for Bro, vNIDS, Sharing all states, and No sharing solutions. The experiments are based on three different real-world traffic with generated attack traces: (a) CAIDA+Attack.trace; (b) LBNL+Attack.trace; and (c) Campus+Attack.trace.**

We further examined the reasons for detection rate degradation in this scenario. We found that the detection rate degradation is caused by distributing traffic relevant to the same malicious activity to different instances. For example, for the Scanner DLP, we totally observed 58 hosts that are conducting scanning through the Bro NIDS in the CAIDA+Attack.trace. The *No Sharing* NIDS only reports 45 hosts. We manually checked the flow rules used to distribute the traffic and examined the flows generated from the 13 hosts that are not detected. We found that those 13 hosts are not reported because their flows are almost evenly distributed to the two instances. For each instance, the scanning speed of those hosts is below the threshold.

As a summary, this experiment demonstrates that vNIDS can address the effective intrusion detection challenge for virtualized NIDSes and achieve equivalent outputs as traditional NIDSes.

## 7.3 Detection State Sharing Overhead

In this experiment, we evaluated the overhead of detection state sharing in three scenarios described in § 7.2: *Share All*, *vNIDS*, and *No Sharing*.

We replayed the same traffic trace to the virtualized NIDS in three scenarios and observed the processing time of each packet. We chose the processing time as the evaluation metric because this time reflects how fast a packet can be processed. If a packet takes too much time to be processed, the NIDS instance cannot keep up with the packet transmission rate thus may drop packets.

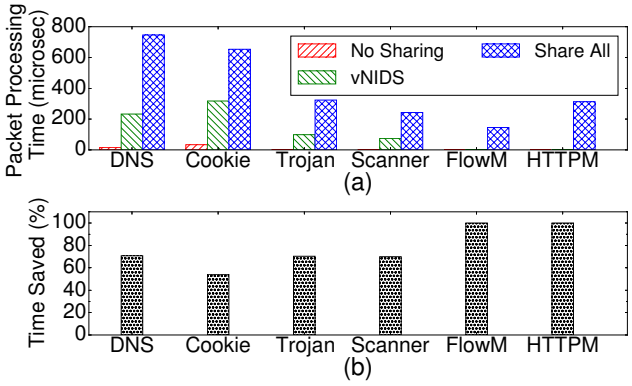


**Figure 5: (a) Packet processing time of each detection program in three scenarios. (b) Packet processing time reduced by vNIDS compared with sharing all detection states.**

| Detection Programs | # Global State | # Local State | #"Forward" |
|---|---|---|---|
| DNS Tunneling DLP | 23 | 146 | 1 |
| Cookie Hijacking DLP | 28 | 174 | 1 |
| Trojan DLP | 32 | 187 | 2 |
| Scanner DLP | 28 | 121 | 1 |
| Flow Monitor DLP | 24 | 99 | 0 |
| HTTP Monitor DLP | 29 | 168 | 0 |

**Table 2: Statistics of *global* state, *local* state and *forward* statement in our DLPs.**

Figure 5 (a) presents the packet processing time of each DLP. On the one hand, the results of this experiment demonstrate that sharing all detection states introduced non-trivial performance overhead to the virtualized NIDSes. On the other hand, for DNS Tunneling, Cookie Hijacking, Trojan, and Scanner DLPs, we observed that vNIDS saves 70.71%, 54.22%, 70.34%, and 69.96% of the processing time compared with the approach that shared all detection states, shown in Figure 5 (b). Especially, vNIDS saves 92.23% and 99.55% packet processing time for the Flow Monitor and HTTP Monitor DLPs, respectively, because their *global* states are accessed much less frequently. vNIDS keeps all of their detection states locally to each instance to significantly reduce the processing time of each packet. Those results demonstrate that sharing all detection states introduce non-trivial performance overhead and vNIDS enables only *global* detection states to be shared to minimize performance overhead significantly.

To further quantify the overhead introduced by vNIDS, we evaluated the number of *global* states, *local* states, and forward statements in our DLPs. Table 2 lists the numbers we observed. On average, the number of global detection states is only 15.4% of the number of all detection states. In addition, the number of forward statements is very small for each DLP, because most of the forward statements are never executed since the very first forward statement interrupts the execution, avoiding the followed forward statements being executed.

## 7.4 Microservice Efficiency

In this experiment, we evaluated the efficiency of virtualized NIDSes provisioned as microservices. First, we evaluated how fast each

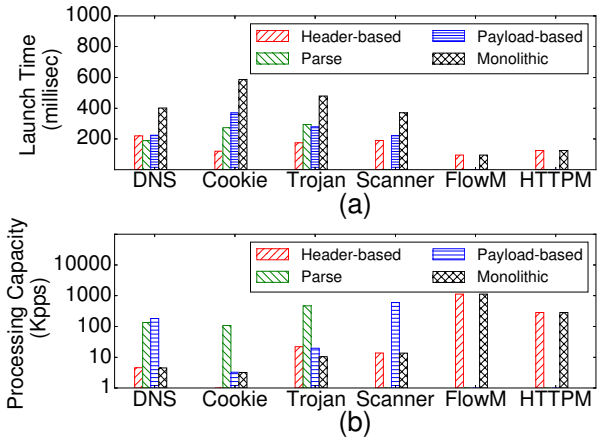


**Figure 6: Launch time (a) and processing capacity (b) of *header-based detection, protocol parse, payload-based detection*, and Monolithic NIDS instances.**

microservice instance can launch with various detection programs installed and compared them with the launch time of the monolith NIDS instance. Then, we compared the processing capacity of each microservice instance with that of monolithic NIDS instance. The launch time is used to estimate how fast the microservice instance can scale out by launching new instances. The processing capacity reflects the number of CPU cores required to achieve a specific performance requirement. The greater the processing capacity, the fewer CPU cores are required to achieve a specific performance requirement, therefore, the instance required less resource to run.

We implemented our microservices based on Click elements. For the monolithic NIDS, we included all the elements of three microservices and instantiated them within a single instance. Notice that there are four cases (Figure 2) with respect to composing microservice chains. In our experiment results, if a microservice is not included in microservice chain, we set the processing time of that microservice as 0.

Based on the evaluation results shown in Figure 6 (a), we observed that, for all the detection programs, the launch time of each microservice instance is less than the launch time of the monolithic NIDS instance, though the sum of the launch times of all microservice instances is greater than the launch time of monolithic NIDS instance. This is because each microservice instance includes fewer Click elements than the monolithic NIDS instance but some elements have been executed by multiple times. The results of this experiment indicate that microservice instances can launch faster than monolithic NIDS instances thus can scale faster.

We also compared the processing capacity of each microservice instance with that of the monolithic NIDS instance. Figure 6 (b) shows the comparison results. We observed that all the microservice instances achieve greater processing capacity than the monolithic NIDS instance. That means, every microservice instance requires less resource than the monolithic NIDS instance to achieve equal processing capacity. Therefore each microservice instance can be provisioned with less resource than the monolithic NIDS instance.

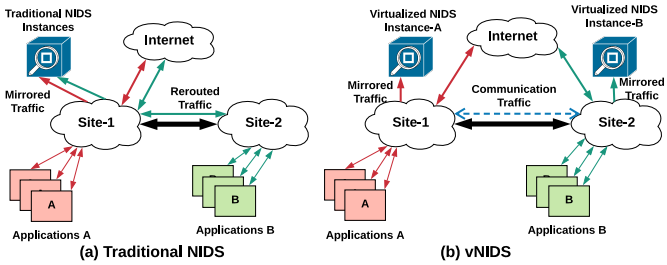


**Figure 7: (a) Traditional NIDS needs to rerouting traffic of Applications B back to *Site-1* for processing. (b) vNIDS can provision its instances flexibly at *Site-2*, if Applications B have been migrated to *Site-2*.**

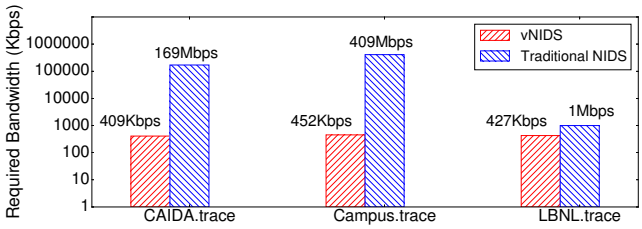


**Figure 8: Required bandwidth between *Site-1* and *Site-2* for vNIDS and traditional NIDS respectively.**

## 7.5 Flexibility of Placement Location

In this experiment, we evaluated the flexibility of vNIDS regarding placement location. The experimental environment is built on CloudLab involving two data centers (*Site-1*) and (*Site-2*) that are geographically distributed. In the experiment, we wanted to demonstrate that under the situation where some applications (Applications B) have been migrated from *Site-1* to *Site-2*, virtualized NIDSes have advantages over traditional NIDSes due to their flexibility of placement location. Figure 7 demonstrates the different behaviors of two types of NIDSes.

Originally, all applications are run at *Site-1* and the NIDS is provisioned at *Site-1* to protect those applications. If Applications B are migrated to *Site-2* due to resource management or optimization purpose, as traditional NIDSes are difficult to migrate, in this case, the network traffic relevant to Applications B must be rerouted from *Site-2* back to *Site-1* for processing. In contrast, vNIDS can flexibly provision new instances at *Site-2* to process the traffic of Applications B. The only concern is the communication traffic, the volume of which we quantified in this experiment, between the virtualized NIDS instances located at different sites.

During the experiment, we divided each of our datasets into two parts, representing the traffic of Applications A and Applications B, respectively. Then, we replayed the two parts of each dataset at two sites, respectively. In the traditional NIDS case, traffic replayed at *Site-2* is rerouted to *Site-1* and we evaluated the bandwidth required by the rerouting. In the vNIDS case, traffic is replayed at *Site-1* and *Site-2* separately and is processed by instances at corresponding sites. We evaluated the bandwidth required by the communication between instances in the two sites.

The required bandwidth between *Site-1* and *Site-2* for traditional NIDS and vNIDS respectively is shown in Figure 8. We tested the three real-world datasets (without synthetic attack traces injected) and found that the bandwidth required by vNIDS is much less than traditional NIDS. Especially, for Campus.trace, traditional NIDS requires *1000x* more bandwidth than vNIDS. This is because vNIDS only needs to exchange *global* detection states between the two sites, while traditional NIDS needs to reroute all the network traffic between the two sites. Another important observation is that for all three real-world datasets, the communication bandwidth between virtualized NIDS instances is surprisingly low – only hundreds of Kbps – for the three datasets. This is because the number of *global* detection states is relatively small.

## 7.6 Flexibility of Processing Capacity

In this experiment, we evaluated the flexibility of vNIDS regarding processing capacity and compared it with Bro Cluster. We only used Campus.trace, which has the highest traffic rate, for this experiment. We ramped the traffic sending rate gradually from 0 up to 10Gbps in 100 seconds and observed the throughput of Bro Cluster and vNIDS. We deployed 5 instances for Bro Cluster in the experiment because each Bro Cluster instance, estimated against our dataset, can roughly handle 1Gbps traffic. In contrast, vNIDS scales according to the traffic volume. Once existing virtualized NIDS instances are about to overload, we created a new instance and redirected some traffic to the new instance.

Figure 9 depicts our observation of this experiment. Figure 9-a indicates that the throughput of Bro Cluster and vNIDS steadily increases as the traffic volume grows. Once the traffic volume reaches the maximum processing capacity of Bro Cluster, the throughput of Bro Cluster is limited to less than 6Gbps. For vNIDS, as it can quickly launch new instances, its throughput grows consistently with the traffic volume. Figure 9-b depicts the number of provisioned instances for Bro Cluster and vNIDS over time. Bro Cluster provisions a static number of instances. Virtualized NIDS, in contrast, provisions its instances according to the traffic volume. As the traffic volume grows, vNIDS provisions an increasing number of instances to handle the traffic. We saw that when the traffic volume is under 5Gbps, the Bro Cluster is *over-provisioning*, while the traffic volume exceeds 5Gbps, the Bro Cluster becomes *under-provisioning* (overloaded in Figure 9-a).

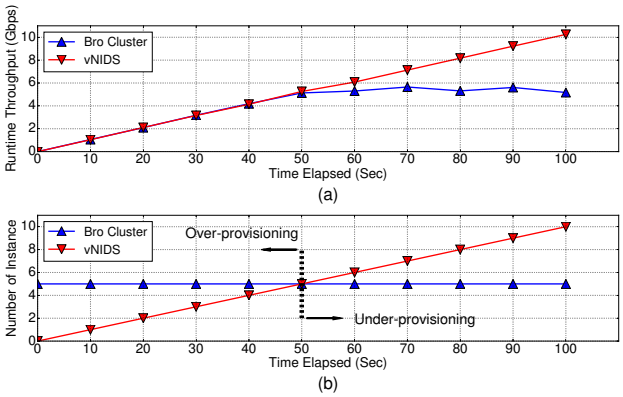


**Figure 9: (a) Runtime throughput of vNIDS and Bro Cluster. (b) Number of instances of vNIDS and Bro Cluster.**

# 8 DISCUSSION

**Service chain acceleration:** Running virtualized NIDS instances in virtual machines or containers may incur extra overhead. Also, provisioning NIDSes as microservices requires communication between microservices. We did not optimize our virtualized NIDS prototype regarding those two aspects in this work. However, acceleration of network function service chain has been widely investigated by the NFV research community [50, 56, 58, 64, 79, 87, 88]. 1) ClickNP [56] and NetBricks [64] are devoted to accelerating the execution of a single instance. 2) ClickOS [58] and NetVM [50, 87] focus on accelerating the packet delivery between instances. 3) NFP [79] and ParaBox [88] accelerate the execution of the whole service chain by parallelizing the execution of different services. We can adopt those techniques to our prototype to further mitigate the extra overhead introduced by NIDS virtualization and improve the performance of virtualized NIDSes.

**Applicability of vNIDS architecture:** In this work, we present vNIDS architecture for NIDS virtualization based on NIDSes that support multi-flow information and stateful protocol analysis, such as Bro [22]. However, due to the complexity of mature NIDSes – e.g., it is reported that the recent version of Bro includes 97K lines of code, 1798 classes and 3034 procedures [53] and has been evolving for nearly 20 years – we choose to implement a prototype of virtualized NIDS to demonstrate our idea. This is however not the limitation of our approach. Especially, the *program analysis* and *program slicing* techniques we employed in this work can also be applied to existing mature NIDSes. For example, recent work [35] has demonstrated the possibility to utilize program slicing upon Bro by introducing an intermediate assembly-like representation of its detection logic programs. Signature-based NIDSes, such as Snort [20] and Suricata [21], can also benefit from our approach. Relying on classic per-flow signature matching, though signature-based NIDSes do not need to share detection states among instances to achieve effective detection, they can still apply our approach to enable non-monolithic provisioning when virtualized. Worth noting, detection logic partitioning is not a challenge for signature-based NIDSes since signatures usually explicitly specify the header-based and payload-based detection logics [73]. In addition, as a recent trend, artificial neural network (ANN) is used to build NIDSes [60]. Our work gains a foothold in NIDS virtualization and can be further extended to support ANN-based NIDS virtualization in the future.

**Usage of virtualized NIDSes:** We envision that virtualized NIDSes with more flexibility than traditional NIDSes can be used to protect both traditional networks [76, 85] and virtualized environments [9, 24]. Besides, as a new trend, edge computing, in which computing and storage resources are placed at the Internet's edge in close to IoT devices, has drawn an increasing number of research attentions from the security community [48, 61, 63, 74]. The major challenges are the limited resource and diversity of edge

computing environments. Virtualized NIDSes can be easily customized through microservice composition and can provision different components flexibly at different locations and with different number of instances. These flexibilities of virtualized NIDSes are well suited to address the limited resource and diversity of edge computing environments. For example, we can provision only *header-based detection* microservice at the edge computing environment (easily customized). If the traffic needs further analysis on the payload, we can then provision the *payload-based detection* microservice on demand (flexible capacity) in the same edge computing environment or even in a remote cloud (flexible location), depending on the trade-offs between performance and resource usage.

**Elastic security vision:** To this end, there are some efforts having been made to develop elastic security approaches, such as DDoS defense systems [42], security architecture for enterprise networks [85], virtual firewall controller [36], and NIDS virtualization in this work. However, these efforts are disjoint points at different security areas. In our work, the idea of provisioning network security functions as microservices has been proved to be a feasible way to achieve more efficiency (than monolithically provisioning) for elastic security approaches (§ 7.4). Though our approach is specific to NIDS virtualization, it can be generalized to develop other network security functions leveraging microservices. We call this paradigm as *network security as microservices*. As a result, we envision an open platform that can facilitate the design of elastic security approaches with the support of diverse virtual security functions developed with microservices.

## 9 RELATED WORK

Some recent research efforts have used NFV and SDN techniques to address the inflexibility and inelasticity limitations of traditional network defense mechanisms [36, 42, 85]. In Bohatei [42], the authors address the limitations of traditional DDoS defense systems with respect to flexibility and elasticity. Bohatei utilizes NFV to dynamically launch virtual machines at different locations to handle DDoS attack traffic and leverages SDN to redistribute the traffic that optimizes the bandwidth consumption. The DDoS defense system is specific for DDoS attacks, while our work is for NIDSes that aim to defensing various attacks. In addition, we have considered the cases where different instances need to exchange information to ensure detection effectiveness, which is overlooked by Bohatei. The authors of PSI [85] propose a security architecture for enterprise networks leveraging NFV and SDN to provide better isolation between security functions, support context-aware security policies, and agilely change its security policies. However, the design of each specific network security function remains unsolved in that work. Our work focuses on improving the specific design of NIDSes for virtualization purpose. VFW Controller [36] is devoted to addressing the problems of firewall virtualization. In particular, VFW Controller analyzes the relationship between firewall rules and OpenFlow rules to safely select the firewall rules that need to migrate between virtual firewall instances during virtual firewall elastic scaling. Our work, in contrast, addresses the challenges in NIDS virtualization, which are fundamentally different from the challenges of virtual firewalls.

Existing work of NIDSes is devoted to developing *scalable* architectures for NIDSes [35, 77, 81]. However, *flexible* architectures with respect to capacity and location for NIDSes are missing. The authors of [35] propose a concurrency model for stateful deep packet inspection to enable NIDSes to run with multiple threads in parallel for better scalability. Their approach attempts to avoid inter-thread communication to reduce the performance overhead induced by detection state sharing. However, such an approach proposes to maintain detection states in each instance separately, without any sharing. As we have demonstrated in § 2.1, for some complex detection logics, it is impractical to achieve effective intrusion detection only relying on traffic distribution. NIDS Cluster [81] and multi-core architecture for NIDS [77] enable information exchange between NIDS nodes or threads to achieve effective intrusion detection. However, both of them fail to give any analysis or classification to the detection states. In particular, NIDS Cluster relies on users to manually determine which detection states need to be shared. Our work is fundamentally different from existing NIDS research efforts in the point of view that our work aims to develop a more *flexible* architecture for NIDSes and addresses the effective intrusion detection and non-monolithic NIDS provisioning problems, which cannot be well addressed by existing NIDS architectures for *virtualized* environments.

There is another body of work from the network research community, which addresses state management problems of virtualized network functions [46, 51, 53, 70, 84]. Split/Merge [70] and OpenNF [46] are limited to addressing the state *migration* problem during network function elastic scaling. Both methods maintain the state of each instance separately at *runtime*, therefore cannot be used to address the effective intrusion detection problem of NIDS virtualization. StateAlyzer [53] classifies the states of network functions to reduce the overhead during state migration. However, such classification is based on per-packet processing procedure and cannot identify per-flow states. Our work ensures the detection effectiveness for virtualized NIDSes at *runtime*, and is able to identify per-flow detection states to minimize detection state sharing. StatelessNF [51] proposes to share all states among network function instances. This approach incurs significantly performance overhead when applied by virtualized NIDSes (evaluated in § 7.3). The authors of [84] use the concept of *local* and *global* states for network functions. However, they fail to provide a systematic approach that can automatically distinguish between the two types of states in a program. Our work can distinguish *local* and *global* detection states automatically in a detection logic program through program analysis.

## 10 CONCLUSION

In this paper, we have proposed vNIDS, a new NIDS architecture, to achieve safe and efficient virtualization of NIDSes. We have carefully designed solutions in vNIDS to address two key challenges including *effective intrusion detection* and *non-monolithic NIDS provisioning* in NIDS virtualization. We have implemented a prototype of vNIDS and evaluated it with various detection logic programs based on real-world traffic traces. Our evaluation results have demonstrated the safety and efficiency of vNIDS for NIDS virtualization.

## ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation (NSF) under Grant No. 1642143, 1723663, 1700499, 1617985, 1642129, 1700544, 1740791, and 1642031. 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.

## REFERENCES

[1] 2010. Firesheep. http://codebutler.com/firesheep. (2010).
[2] 2012. Network Function Virtualisation - Introductory White Paper. https://portal.etsi.org/nfv/nfv_white_paper.pdf. (2012).
[3] 2016. The CAIDA UCSD Anonymized Internet Traces 2016-0406. http://www.caida.org/data/passive/passive_2016_dataset.xml. (2016).
[4] 2018. Algo-logic systems. http://algo-logic.com/. (2018).
[5] 2018. An implementation of an E-component of Network Intrusion Detection System. http://libnids.sourceforge.net/. (2018).
[6] 2018. Bro Script for Detecting Cookie Hijacking. http://matthias.vallentin.net/blog/2010/10/taming-the-sheep-detecting-sidejacking-with-bro/. (2018).
[7] 2018. Bro Script for Detecting DNS Tunneling. https://github.com/hhzzk/dns-tunnels. (2018).
[8] 2018. Bro Scripting Language. https://www.bro.org/sphinx/scripting/. (2018).
[9] 2018. Check Point virtual appliance for AWS. https://aws.amazon.com/marketplace/pp/B00CWNBJOY. (2018).
[10] 2018. CloudLab. http://www.cloudlab.us/. (2018).
[11] 2018. CodeSurfer. https://www.grammatech.com/products/codesurfer. (2018).
[12] 2018. Frama-c Software Analyzers. https://frama-c.com/. (2018).
[13] 2018. GNU Bison. https://www.gnu.org/software/bison/. (2018).
[14] 2018. LBNL/ICSI Enterprise Tracing Project. http://www.icir.org/enterprise-tracing/. (2018).
[15] 2018. Nmap Security Scanner. https://nmap.org/. (2018).
[16] 2018. Open vSwitch. https://www.openvswitch.org/. (2018).
[17] 2018. Port Scanner. https://en.wikipedia.org/wiki/Port_scanner. (2018).
[18] 2018. Project Floodlight. http://www.projectfloodlight.org/projects/. (2018).
[19] 2018. Scapy Project. http://www.secdev.org/projects/scapy/. (2018).
[20] 2018. Snort Network Intrusion Detection & Prevention System. https://snort.org/. (2018).
[21] 2018. Suricata IDS. https://suricata-ids.org/. (2018).
[22] 2018. The Bro Network Security Monitor. https://www.bro.org/. (2018).
[23] 2018. The Fast Lexical Analyzer - scanner generator for lexing in C and C++. https://github.com/westes/flex. (2018).
[24] 2018. VMware vCloud. https://www.vmware.com/products/vcloud-suite.html. (2018).
[25] 2018. Xen Project. https://www.xenproject.org/. (2018).
[26] 2018. Xen Toolstack. https://wiki.xen.org/wiki/XL. (2018).
[27] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. 2011. EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis.. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS 2011)*.
[28] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 13–16.
[29] Kevin Borders, Jonathan Springer, and Matthew Burnside. 2012. Chimera: a declarative language for streaming network traffic analysis. In *Proceedings of the 21st USENIX conference on Security symposium (USENIX Security 2012)*. USENIX Association, 19–19.
[30] Anat Bremler-Barr, Yotam Harchol, and David Hay. 2016. OpenBox: a software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 511–524.
[31] P Busschbach. 2013. Network functions virtualisation-challenges and solutions. *Alcatel-Lucent Corp., France, Strategic White Paper* (2013).
[32] Patrick Butler, Kui Xu, and Danfeng Yao. 2011. Quantitatively analyzing stealthy communication channels. In *Applied Cryptography and Network Security*. Springer, 238–254.
[33] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure web applications via automatic partitioning. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 31–44.
[34] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*. ACM, 301–314.
[35] Lorenzo De Carli, Robin Sommer, and Somesh Jha. 2014. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS 2014)*. ACM, 1378–1390.
[36] Juan Deng, Hongda Li, Hongxin Hu, Kuang-Ching Wang, Gail-Joon Ahn, Ziming Zhao, and Wonkyu Han. 2017. On the Safety and Efficiency of Virtual Firewall Elasticity Control. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS 2017)*.
[37] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI 2014)*. 401–414.
[38] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*. Springer, 195–216.
[39] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. 2008. Predicting the resource consumption of network intrusion detection systems. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*. Springer, 135–154.
[40] Wendy Ellens, Piotr Żuraniewski, Anna Sperotto, Harm Schotanus, Michel Mandjes, and Erik Meeuwissen. 2013. Flow-based detection of DNS tunnels. In *IFIP International Conference on Autonomous Infrastructure, Management and Security*. Springer, 124–135.
[41] Greg Farnham and A Atlasis. 2013. Detecting DNS tunneling. *SANS Institute InfoSec Reading Room* 9 (2013), 1–32.
[42] Seyed K Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. 2015. Bohatei: flexible and elastic DDoS defense. In *Proceedings of the 24th USENIX Conference on Security Symposium (USENIX Security 2015)*. USENIX Association, 817–832.
[43] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. 2014. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*, Vol. 543. 546.
[44] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.
[45] Open Networking Foundation. 2014. OpenFlow-enabled SDN and network functions virtualisation. https://www.opennetworking.org/images/stories/downloads/sdn-resources/solution-briefs/sb-sdn-nvf-solution.pdf. (2014).
[46] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling innovation in network function control. In *Proceedings of the 2014 ACM SIGCOMM Conference*, Vol. 44. ACM, 163–174.
[47] J Gross, T SRIDHAR, P GARG, C WRIGHT, and I GANGA. 2016. Geneve: Generic network virtualization encapsulation. IETF draft. (2016).
[48] Ibbad Hafeez, Aaron Yi Ding, and Sasu Tarkoma. 2017. Securing Edge Networks with Securebox. *arXiv preprint arXiv:1712.07740* (2017).
[49] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 1 (1990), 26–60.
[50] Jinho Hwang, K K_ Ramakrishnan, and Timothy Wood. 2015. NetVM: high performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management* 12, 1 (2015), 34–47.
[51] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless network functions: breaking the tight coupling of state and processing. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI 2017)*. USENIX Association, 97–112.
[52] Murad Kablan, Blake Caldwell, Richard Han, Hani Jamjoom, and Eric Keller. 2015. Stateless network functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox 2015)*. ACM, 49–54.
[53] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. 2016. Paving the way for NFV: simplifying middlebox modifications using StateAlyzr. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI 2016)*. USENIX Association, 239–253.
[54] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.
[55] Thomas Lengauer and Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1, 1 (1979), 121–141.
[56] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 1–14.
[57] Edward S Lowry and Cleburne W Medlock. 1969. Object code optimization. *Commun. ACM* 12, 1 (1969), 13–22.
[58] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI 2014)*. USENIX Association, 459–473.

[59] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.
[60] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. 2018. Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS 2018)*.
[61] Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jorg Ott. 2018. Consolidate IoT edge computing with lightweight virtualization. *IEEE Network* 32, 1 (2018), 102–111.
[62] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. 2010. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 92–105.
[63] Jianli Pan and Zhicheng Yang. 2018. Cybersecurity Challenges and Opportunities in the New Edge Computing+ IoT World. In *Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 29–32.
[64] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: taking the V out of NFV. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI 2016)*. USENIX Association, 203–216.
[65] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. 2006. binpac: A yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*. ACM, 289–300.
[66] Milan Patel, B Naughton, C Chan, N Sprecher, S Abeta, A Neal, et al. 2014. Mobile-edge computing introductory technical white paper. *White Paper, Mobile-edge Computing (MEC) industry initiative* (2014).
[67] Vern Paxson. 1999. Bro: a system for detecting network intruders in real-time. *Computer networks* 31, 23 (1999), 2435–2463.
[68] Larry Peterson, Ali Al-Shabibi, Tom Anshutz, Scott Baker, Andy Bavier, Saurav Das, Jonathan Hart, Guru Palukar, and William Snow. 2016. Central office re-architected as a data center. *IEEE Communications Magazine* 54, 10 (2016), 96–101.
[69] Paul Quinn and Uri Elzur. 2016. Network service header. *Internet Engineering Task Force, Internet-Draft draft-ietf-sfc-nsh-10* (2016).
[70] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/merge: system support for elastic execution in virtual middleboxes. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI 2013)*. USENIX Association, 227–240.
[71] S. K. N. Rao. 2014. SDN and its Use-Cases-NV and NFV. http://www.nectechnologies.in/en_TI/pdf/NTI_whitepaper_SDN_NFV.pdf. (2014).
[72] Thomas Reps and Genevieve Rosay. 1995. Precise interprocedural chopping. In *ACM SIGSOFT Software Engineering Notes*, Vol. 20. ACM, 41–52.
[73] Martin Roesch and Chris Green. 2016. Snort Users Manual 2.9. 8.2. (2016).
[74] Rodrigo Roman, Javier Lopez, and Masahiro Mambo. 2018. Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges. *Future Generation Computer Systems* 78 (2018), 680–698.
[75] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 8, 4 (2009).
[76] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making middleboxes someone else's problem: network processing as a cloud service. In *Proceedings of the 2012 ACM SIGCOMM Conference*, Vol. 42. ACM, 13–24.
[77] Robin Sommer, Vern Paxson, and Nicholas Weaver. 2009. An architecture for exploiting multi-core processors to parallelize network intrusion prevention. *Concurrency and Computation: Practice and Experience* 21, 10 (2009), 1255–1279.
[78] Aditya K Sood and Richard J Enbody. 2013. Targeted cyberattacks: a superset of advanced persistent threats. *IEEE security & privacy* 11, 1 (2013), 54–61.
[79] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV. In *Proceedings of the 2017 ACM SIGCOMM Conference*. ACM, 43–56.
[80] Matthias Vallentin. 2011. Taming the Sheep: Detecting Sidejacking with Bro. http://matthias.vallentin.net/blog/2010/10/taming-the-sheep-detecting-sidejacking-with-bro/. (2011).
[81] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. 2007. The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID 2007)*. Springer, 107–126.
[82] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.
[83] G. Wellbrock and T. J. Xia. 2014. How will optical transport deal with future network traffic growth?. In *2014 The European Conference on Optical Communication (ECOC)*. 1–3.
[84] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2018)*. USENIX Association, 299–312.
[85] Tianlong Yu, Seyed K Fayaz, Michael Collins, Vyas Sekar, and Srinivasan Seshan. 2017. PSI: Precise Security Instrumentation for Enterprise Networks. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS 2017)*.
[86] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. 2015. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets 2015)*. ACM, 5.
[87] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, KK Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A platform for high performance network service chains. In *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization (HotMiddlebox 2016)*. ACM, 26–31.
[88] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. 2017. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In *Proceedings of the Symposium on SDN Research*. ACM, 143–149.

# Appendix A COOKIE HIJACKING DETECTION

To detect the cookie hijacking attack, the NIDS monitors whether a cookie is used by different users. Basically, the NIDS tracks the IP address for every cookie usage. If a cookie is used by different IP addresses, it means the cookie has been reused by different users. However, the detection results are not often true if only based on IP addresses, because in many networks the IP address of a host is dynamically allocated and can be changed from time to time. In order to improve the detection accuracy, the NIDS makes use of extra information, such as MAC addresses [4]. The NIDS collects the MAC address information by monitoring the traffic of a DHCP server and maintains the IP-to-MAC bindings in a table. Every time the NIDS observes a cookie being reused by different IP addresses, it further checks whether those two IP addresses are assigned to different MAC addresses. If so, a cookie hijacking attack is reported. The detector [80] implementing the aforementioned detection logics should employ at least two tables: the Cookies table maintaining the context (e.g., the IP and MAC addresses of the original user of the cookie) of each cookie that has been observed by the NIDS, and the IP2MAC table maintaining the bindings between IP and MAC addresses.

The detection process is demonstrated in Figure 10. For each HTTP flow, the detector extracts the cookie and retrieves the context of the cookie from the Cookies table (1). If there is no context available, a new context record of the cookie is added. This operation requires access to the IP2MAC table to obtain the MAC address (2). If the context exists, but the IP address in the context record is not consistent to the IP address of the flow (3), the detector needs to access the IP2MAC table to determine whether the two IP addresses are from the same user by looking at the MAC addresses (4). In the example shown in Figure 10, *Flow1* is from the original user using cookie *C1*. If *Flow2*, which is actually from an attacker, is delivered to another instance that does not process *Flow1* thus lacks the context of *C1*, *Flow2* is falsely considered as a legitimate user of the cookie *C1*. However, without deep analysis of traffic, it is impossible to know which two flows contain the same cookie thus should be delivered to the same instance.

[4] It is still possible for attackers to bypass this detection. However, augmenting the robustness of existing detectors is not the focus of this paper.

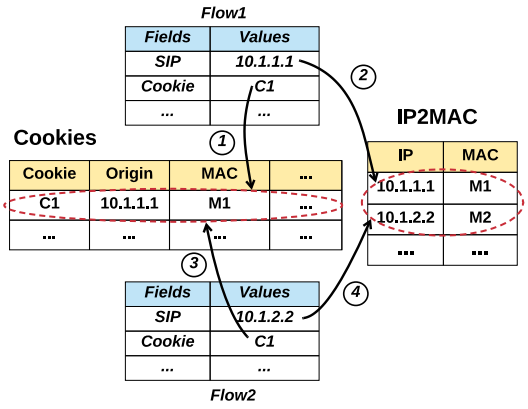


**Figure 10: The process of cookie hijacking detection and the detection states maintained by the detector. *Flow1* is from the original user of the cookie *C1*. *Flow2* is from an attacker reusing the cookie *C1*.**

In addition, it is impossible to predict which two detection states must be maintained by the same instance. For example, in Figure 10, if *Flow1* and *Flow2* do not appear, the two records indexed by *10.1.1.1* and *10.1.2.2* in the IP2MAC table can be maintained by different instances. However, if *Flow1* and *Flow2* come, these two flows must be delivered to the same instance that maintains the context of the cookie (*C1*). As described above, *Flow1* and *Flow2* also need to check the two records indexed by *10.1.1.1* and *10.1.2.2* in the IP2MAC table. Therefore, those two records must also be maintained in the same instance. This example demonstrates that the incoming traffic could dynamically correlate two detection states, thus it is impossible to predict beforehand which detection state is maintained by which instance.

## Appendix B DETECTION STATE CLASSIFICATION EXAMPLE

A DLP detecting scanners in the network consists of two tasks:

- Counting the number of flows generated by each host. If a host generates too many flows in a short time, it's potentially a scanner.
- Monitoring the number of packets for each flow. If a flow contains many packets, this flow is not likely for scanning purpose.

The DLP should have two tables: a Hosts table maintaining the number of flows generated by each host, and a Flowcounts table maintaining the number of packets of each flow.

Figure 11 shows the CFG [5] of the DLP combined with the NE. Each node in the CFG represents a block of statements and each directed edge represents a jump between blocks. The DLP considers a host as a scanner if it generates too many flows in a short period of time. But not all the flows are taken into consideration. Only those flows that contain a few packets will be considered. The Hosts[sip] stores the number of flows that have been generated by the host with sip as IP address. The Flowcounts[p.sip, p.dip, p.sp, p.dp] stores the number of packets contained in the flow identified by source IP, destination IP, source port, and destination port, respectively. For simplicity, let's denote "p.sip, p.dip, p.sp, p.dp" as "id".

[5]For simplicity, we prune nodes and edges irrelevant to our analysis.

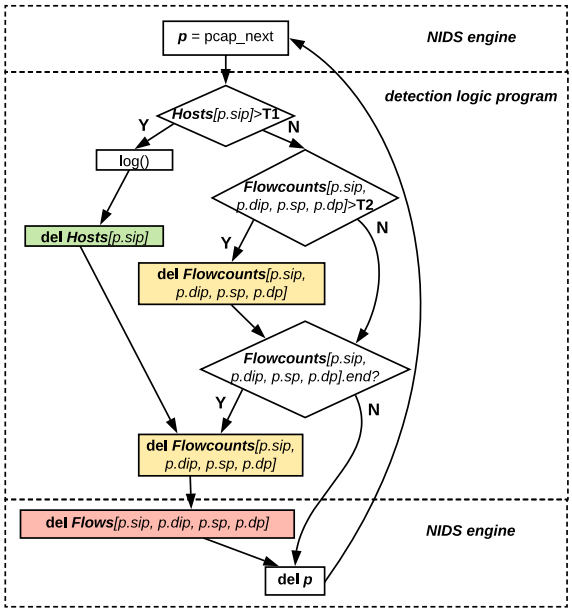


**Figure 11: The CFG of a scanner detecting DLP and NE.**

If a new flow is observed, Hosts[p.sip] increments. If a packet belonging to an existing flow comes, Flowcounts[id] is updated. If Flowcounts[id] exceeds a threshold (T2), Hosts[sip] decrements because this flow is not for scanning purpose. If Hosts[sip] exceeds a threshold (T1), the IP address sip is logged. Finally, if the flow terminates (determined by expiration or *FIN* flag), the flow record maintained by NE is destroyed.

Intuitively, Hosts[p.sip] is a *multi-flow* detection state, which is accessed by all flows originating from sip, while Flowcounts[p.sip, p.dip, p.sp, p.dp] is a *per-flow* detection state that can only be accessed by the flow generated from sip at port sp and received by dip at port dp. We manually identify the destroy statement of the flow record ("del Flows[p.sip, p.dip, p.sp, p.dp]"). The flow record is defined and maintained in the NE.

We observe that in the CFG, "del Flowcounts[p.sip, p.dip, p.sp, p.dp]" is a dominator of "del Flows[p.sip, p.dip, p.sp, p.dp]", while "del Hosts[p.sip]" is not. A statement $P$ is a dominator of statement $Q$ if every path in the graph from a entry statement ("p=pcap_next") to $Q$ always passes through $P$ [57]. This means the value of Flowcounts[p.sip, p.dip, p.sp, p.dp] variable is always destroyed before the flow record is destroyed, while the value of Hosts[p.sip] is not necessarily destroyed before the flow record is destroyed. Based on this fact, we, therefore, know that flows[p.sip, p.dip, p.sp, p.dp] can only be accessed by a specific flow (generated from sip at port sp and received by dip at port dp), while Hosts[p.sip] can be accessed by flows other than that specific flow because Hosts[sip] is possibly still "alive" after that specific flow terminates. Finally, we can infer that Flowcounts[p.sip, p.dip, p.sp, p.dp] is a *per-flow* detection state, while Hosts[p.sip] is a *multi-flow* detection state.

## Appendix C DLP PARTITIONING EXAMPLE

DNS protocol is usually considered as a commonly used protocol to interpret the domain names on the Internet thus is rarely prohibited by firewalls. In a DNS Tunneling attack, the attacker tries to bypass the firewall by encapsulating the malicious content in DNS traffic. The DNS Tunneling DLP tries to detect DNS Tunneling attacks. We use this DLP as a use case because the DNS Tunneling detection techniques have been well studied in lots of literature [27, 32, 40, 41], and there is also a DLP implemented by Bro scripting language [7]. We have implemented a C version of DNS Tunneling DLP based on existing Bro DNS Tunnels detection scripts. The DLP involves both header-based DL and payload-based DL. Thus, it would be a good use case to demonstrate our approach for partitioning the DLP.

The code shown in Code 1 is an implementation of the DNS Tunneling DLP in C language. The Tunneling_Detection is a C interface stub that will be called back by the NE when a DNS packet is received. The parameters sip and length are header fields of incoming packets, while the query is the payload information indicating the domain name being queried by the DNS packet. The basic idea of the DLP is as follows: The hosts is a pointer of a linked list. Each element in the list is a structure recording which IP address this element is associated with and the number of DNS packets being counted by the DLP. Each time a DNS packet is received, the DLP increments the count accordingly (lines 26 and 27). The DLP first checks whether a host has generated too many DNS packets in a period of time (line 28). If this is true, this host is reported as an attacker (lines 29-31). Otherwise, the DLP checks the length of the DNS packet (line 33). If the packet's size exceeds a threshold, the DLP further checks the payload of the packet (lines 34-41); otherwise, this is a normal packet.

*Step1 & Step2:* We use Frama-C to compute the forward slice of query from the beginning of the DLP (line 25). Note that Frama-C has already computed the PDG internally for its program slicing. The resulting statements of the payload-based DLP slice are lines 35, 36, 39, 40, 41, and 45.

*Step3:* To construct the header-based DLP, we replace lines 35, 36, 39, 40, 41, and 45 with forward statements. Then, we identify the variables that should be tracked. According to Algorithm 1, variable length and i have been used by both header-based and payload-based DLPs. Therefore, those two variables should be tracked. We find that length has never been updated (i.e., never appear on the left-hand-side). As a result, only i is encapsulated into the packet and sent to the payload-based DLP.

*Step4:* To construct the payload-based DLP, we first add the initialization statements for the variables storing intermediate results. In this case, we only add a statement to assign the value to variable i. Then, right after the initialization statements, we need the switch-case statement to retrieve the breakpoints. In this case, the switch-case statement checks the label and maps each label to lines 35, 36, 39, 40, 41, and 45, respectively.

*Step5:* Once we have PDGs of both header-based and payload-based DLPs, we can use tools to reverse the PDGs into the source code. In our case, instead of generating PDGs, Frama-C computes the PDG internally and utilizes the PDG to compute the forward program slice. Finally, Frama-C emits the source code of the resulting slice directly. We write a parser using Flex [23] and Bison [13] to do the construction in *Step3* and *Step4*. In practice, most modern compilers can make optimization of the source code by removing unused variables and unreachable statements.

```
1  int T1 = 100; /*frequency threshold*/
2  int T2 = 53; /*length threshold*/
3  int T3 = 0.3; /*numeric threshold*/
4  struct host {
5     unit ip;
6     int count;
7     struct host* next;
8  };
9  struct host* hosts;
10 struct host* find_host(uint ip) {
11    struct host* h = hosts;
12    while (h) {
13       if (h->ip == ip) return h;
14       h = h->next;
15    }
16    h = malloc(sizeof(struct host));
17    h->ip = ip;
18    h->count = 0;
19    h->next = hosts->next;
20    hosts->next = h;
21    return h;
22 }
23
24 int Tunneling_Detection(uint sip,int length,char* query){
25    int num_count = 0, i = 0;
26    struct host* h = find_host(sip);
27    h->count += 1;
28    if (h->count > T1) {
29       h->count = 0;
30       printf("DNS Tunnel Detected!\n");
31       return 1;
32    } else {
33       if (length > T2) {
34          for (i = 0; i < length; i++) {
35             if (query[i]>='0' && query[i]<='9'){
36                num_count += 1;
37             }
38          }
39          if (num_count > length * T3) {
40             printf("DNS Tunnel Detected!\n");
41             return 2;
42          }
43       }
44    }
45    return 0;
46 }
```

**Code 1: Simplified DNS Tunneling Detection Program**