Aloe: An Elastic Auto-Scaled and Self-stabilized Orchestration Framework for IoT Applications

Subhrendu Chattopadhyay*, Soumyajit Chatterjee†, Sukumar Nandi‡, Sandip Chakraborty§
*‡ Department of CSE, IIT Guwahati, India
†§ Department of CSE, IIT Kharagpur, India

***Abstract*—Management of networked Internet of Things (IoT) infrastructure with in-network processing capabilities is becoming increasingly difficult due to the volatility of the system with low-cost resource-constraint devices. Traditional software-defined networking (SDN) based management systems are not suitable to handle the plug and play nature of such systems. Therefore, in this paper, we propose Aloe, an elastically auto-scalable SDN orchestration framework. Instead of using service grade SDN controller applications, Aloe uses multiple lightweight controller instances to exploit the capabilities of in-network processing infrastructure. The proposed framework ensures the availability and significant reduction in flow-setup delay by deploying instances near the resource constraint IoT devices dynamically. Aloe supports fault-tolerance and can recover from network partitioning by employing self-stabilizing placement of migration capable controller instances. The performance of the proposed system is measured by using an in-house testbed along with a large scale deployment in Amazon web services (AWS) cloud platform. The experimental results from these two testbed show significant improvement in response time for standard IoT based services. This improvement of performance is due to the reduction in flow-setup time. We found that Aloe can improve flow-setup time by around $10\% - 30\%$ in comparison to one of the state of the art orchestration framework.**

***Index Terms*—Network architecture, Fault-tolerance, Programmable network, software defined network**

I. Introduction

The rapid proliferation of Internet-of-Things (IoT) has made the network architecture complicated and difficult to manage for service provisioning and ensuring security to the end-users. Simultaneously, with the advancement of edge-computing, in-network processing (also known as fog computing) and platform-as-a-service technologies, end-users consider the network as a service platform for the deployment and execution of myriads of diverse applications dynamically and seamlessly over the network. Consequently, network management is becoming increasingly difficult in today's world with this complex service-oriented platform overlay on top of the inherently distributed TCP/IP network architecture. The concept of software-defined networking (SDN) has gained popularity over the last decade to make the network management simple, cost-effective and logically centralized, where a network manager can monitor, control and deploy new network services through a central controller. Nevertheless, edge and in-network processing over an IoT platform is still challenging even with an SDN based architecture [1].

The primary requirements for supporting edge and in-network processing over a networked IoT platform are as follows: (1) The platform should be agile to support rapid deployment of applications without incurring additional overhead for in-network processing [2]. This also ensures scalability of the system [3]. (2) Many times, in-network processing requires dividing a service into multiple microservices and deploying the microservices at different network nodes for reducing the application response time with parallel computations [4]. However, such microservices may need to communicate with each other, and therefore the flow-setup delay from the in-network nodes need to be very low to ensure near real-time processing. (3) The percentage of short-lived flows are high for IoT based networks [5]. This also escalates the requirement for reducing flow-setup delay in the network. (4) Failure rates of IoT nodes are in-general high [6]. Therefore, the system should support a fault-tolerant or fault-resilient architecture to ensure liveness.

Although SDN supported edge computing and in-network processing have been widely studied in the literature for the last few years [1], [7] as a promising technology to solve many of the network management problems associated with large-scale IoT networks, they have certain limitations. First of all, the SDN controller is a single-point bottleneck. Every flow initiation requires communication between switches and the controller; therefore, the performance depends on the switch-controller delay. With a single controller bottleneck, the delay between the switch and the controller increases, which affects the flow-setup performance. As we mentioned earlier that the majority of the flows in an IoT network are short-lived flows, the impact of switch-controller delay is more severe on the performance of short-lived flows. To solve this issue, researchers have explored distributed SDN architecture with multiple controllers deployed over the network [8]. However, with a distributed SDN architecture, the question arises about how many controllers to deploy and where to deploy those controllers. Static controller deployments may not alleviate this problem, as IoT networks are mostly dynamic with a plug-and-play deployment of devices. Dynamic controller deployment requires hosting the controller software over commercially-off-the-shelf (COTS) devices and designing methodologies for

Email:subhrendu@iitg.ac.in*,soumyachat@iitkgp.ac.in†,sukumar@iitg.ac.in‡, sandipc@cse.iitkgp.ac.in§

This work is partially supported by TCS India, Microsoft India, LRN Foundation and CNeRG IIT Kharagpur.

controller coordination which is a challenging task [9]. The problem is escalated with the objective of developing a fault-tolerant or fault-resilient architecture in a network where the majority of the flows are short-lived flows.

In contrast to the existing architectures, the novelty of this paper is as follows. We integrate an SDN control plane with the in-network processing infrastructure, such that the control plane can dynamically be deployed over the COTS devices maintaining a fault-tolerant architecture. This has multiple advantages for an IoT framework with in-network processing capabilities: (a) The distributed controller approach ensures that there is no performance bottleneck near the controller. (b) The flow-setup delay is significantly minimized because of the availability of a controller near every device. (c) The fault-tolerant controller orchestration ensures the liveness of the system even in the presence of multiple simultaneous devices or network faults. To achieve these goals, we first design a distributed, robust, migration-capable and elastically scalable control plane framework with the help of docker containers [10] and state-of-the-art control plane technologies. The proposed control plane consists of a set of small controllers, called the micro-controllers, which can coordinate with each other and help in deploying new applications for in-network processing. The container platform helps in installing these micro-controllers on the COTS devices; a container with a micro-controller can be seamlessly migrated to another target device if the host device fails, yielding a fault-tolerant architecture. In addition to this, the deployment mechanism for the micro-controllers ensure elastic auto-scaling of the system; the total number of controllers can grow or sink based on the number of active devices in the IoT network. We develop a set of special purpose programming interfaces to ensure fault-tolerant elastic auto-scaling of the system along with intra-controller coordinations. Finally, we design a set of application programming interfaces (API) over this platform to ensure language-free independent deployment of applications for in-network processing. Combining all these concepts, we present Aloe[1], a distributed, robust, elastically auto-scalable, platform-independent orchestration framework for edge and in-network processing over IoT infrastructures.

We have implemented a prototype of Aloe using state-of-the-art SDN control plane technologies and deployed the system over an in-house testbed and a 68-node Amazon web services platform. The in-house testbed consists of 10 nodes (Raspberry Pi devices) with Raspbian kernel version 8.0. As mentioned, we have utilized docker containers to host the distributed control plane platform. We have tested Aloe with three popular applications for in-network IoT data processing – (a) A web server (simple python based), (b) a distributed database server (`Cassandra`), and (c) a distributed file storage platform (`Gluster`). We observe that Aloe can reduce the flow-setup delay significantly (more than three times) compared to state-of-the-art distributed control plane technologies while boosting up application performance even in the presence of multiple simultaneous faults.

[1]Aloevera: A plant known for its healing property.

II. Related Work

Traditional single controller architecture is not suitable for IoT infrastructure, where the network is dynamic and failure prone. One way to address such a problem is to deploy a distributed control plane. However, existing distributed control planes are not efficient for handling large-scale IoT systems that require in-band control. ONIX [11] and ONOS [12] are two popular distributed control plane architectures. ONIX uses a distributed hash table (DHT) data store for storing volatile link state information. On the other hand, ONOS uses NoSQL distributed database and distributed registry to ensure data consistency. Although both of them can scale easily and show a significant amount of fault-resiliency, they require high end distributed computing infrastructure for execution. Deployment of such infrastructure increases the cost of IoT deployment and leads to performance degradation of IoT services which rely on short flows due to the increase in the number of controller consultation. On the other hand, IoT devices require in-band control as most of them have limited network interfaces. Therefore, a disruption in IoT link can have a severe impact on multiple IoT nodes due to disconnection from the control plane. Similar problems can be observed in case of Kandoo [13] and Elasticon [14].

To avoid these challenges DIFANE [15], DevoFlow [16] and BLAC [17] design control plane by utilizing data plane services. DIFANE and DevoFlow use special purpose switches which can take decisions in its local neighborhood in the absence of the controller. However, this requires switch-level modifications which may not be possible for every hardware components. On the other hand, BLAC uses a controller scheduling mechanism to dynamically scale the control plane to accommodate the need the system. However, BLAC increases the flow setup time and not suitable for real-time IoT applications.

In SCL [18], the authors have developed a coordination layer which can provide consistent updates for a single image, lightweight controllers deployed in an in-band fashion. However, SCL uses two-phase commit mechanism for consistency preservation, which incurs high latency. Moreover, SCL is not suitable for IoT services as it assumes the existence of robust channels among switch and controllers, which is not suitable for low-cost and resource-constrained networked IoT systems.

III. Components of Aloe

The Aloe orchestration framework exploits the capabilities of the in-network processing architecture over an IoT based platform where devices work mostly in a plug-and-play mode. The main components of the architecture are shown in Fig. 1. It can be noted here that the proposed architecture does not bring new hardware or software platforms at its base; instead, we utilize the available COTS hardware and open-source software suites to design this entire architecture. Our objective is to design an orchestration platform that can be developed with market-available components while integrating

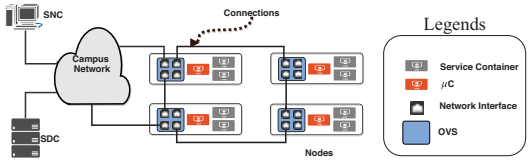


Fig. 1: Components of Infrastructure

innovations in the design such that the shortcomings of the existing systems can be mitigated. We discuss the individual components and their functionalities in this section.

A. Infrastructure Nodes

The networking equipment and devices are considered as the infrastructure nodes. Therefore, nodes are essentially embedded and resource-constraint devices like smart-gateways, smart routers, smart IoT monitoring devices, etc. These devices participate in communication and provide in-network processing platforms for lightweight services by utilizing residual resources. We consider that these nodes are either SDN-supported or can be configured with open-source software platform like *open virtual switch* (OVS) to make them SDN capable.

We use containerized platforms like docker [10] to offload services in the IoT platform for in-network processing. The containerized service deployment helps in supporting service isolation and makes the architecture failsafe by supporting live migration of containers. Further, containers reduce a programmer's overhead for service delegation and cost of deployment, as the same device can be used for in-network processing of IoT applications along with execution of custom networking services.

B. Service Deployment Controller

To identify the resource requirement and delegation of the services which require in-network processing, we use a centralized service deployment controller (SDC). The SDC periodically monitors the resource consumptions of the nodes. Once a new service is ready for deployment in the system, SDC identifies the schedules in which the services can be executed by the nodes without violating resource demands from individual services. Once the schedule is generated, the SDC is responsible for delegating the services based on the schedule. It can be noted that the load of an SDC is much less compared to the network management controller. Therefore we maintain a single instance of SDC in our system.

C. Super Network Controller

Network management in an IoT system is non-trivial due to the diversified inter-service communication requirements and the dynamic nature of the network. Aloe uses a two-layer approach. We deploy a high availability super network controller (SNC) at the first layer, which is responsible for storing persistent network information, like routing protocols, quality of service (QoS) requirements, the periodicity of statistic collection from nodes, etc. An SNC also manages an access control list (ACL) to provide necessary security to the infrastructure nodes.

D. Micro-Controllers

Although super controllers are highly available, an IoT infrastructure has a time-varying topology due to the use of the resource constraint devices and the devices being plug-and-play most of the times. Therefore, the use of a centralized controller cannot achieve fault-tolerance (failure of infrastructure nodes) and partition-tolerance (failure of network links resulting in network partitions). On the other hand, unlike SDC, SNC needs to be consulted by the nodes each time a new flow enters the system. This increases the communication overhead and flow initiation delay which also affects the performance of the services deployed in the infrastructure. Therefore, Aloe uses a second layer of network controllers named as *"micro-controllers"* (μC).

μCs are lightweight SDN controllers. A μC stores volatile link layer information of a small group of nodes placed topologically close to it. Thus a μC maintains information consistency by minimizing the delay between the governing μC and the nodes managed by it. The SNC can aggregate these statistics via `REST API` queries from the μC. Based on the changing QoS of the services, network service provisioning can be achieved in the μC via the same `REST API`. Based on the configuration of the SNC, a μC collects statistics from individual OVS modules of the nodes. Thus a μC can achieve a fine-tuned network control for the infrastructure nodes.

However, deployment of μCs in nodes might also create network partitioning issue. To avoid such an undesirable scenario, Aloe uses a novel approach where the μCs are encapsulated inside a container and deployed as a service inside the infrastructure nodes itself. Thus Aloe supports *μC as a Service* (μCaaS) which ensures fault-tolerance of the system. μC containers can be migrated to a target node quite easily with the help of the live-migration technique of a container when the host node fails. Aloe ensures that a set of μCs is always live in the system maintaining the requirements for minimized switch-controller delay. On the other hand, a μC container can be customized depending on the available capacity of the nodes and resource consumptions by the controller applications. It can be noted that this μC architecture is different from existing distributed SDN controller approaches, such as DevoFlow [16] and SCL [18], which require switch-level customization. μCs can run over the existing COTS devices without any requirement for switch-level modifications.

IV. Design of Aloe Orchestration Framework

This section discusses the Aloe orchestration framework by highlighting various functional modules of Aloe and their working principles. Finally, we develop a set of APIs for language-independent and robust deployment of applications over the Aloe framework. The various functional modules of Aloe are shown in Fig. 2; the detailed description follows.

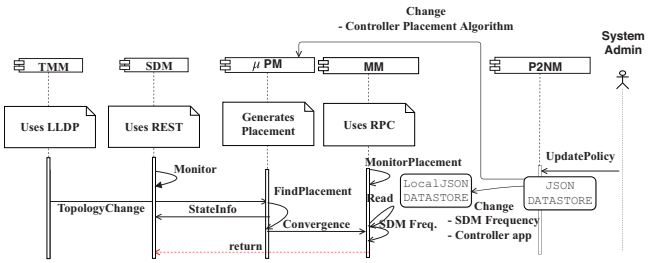


Fig. 2: Aloe function modules and their interactions

A. Aloe Functional Modules

The proposed framework consists of four node-level modules and one SNC-level module. The node-level modules run inside the infrastructure nodes and decide the topology and service parameters that need to be synchronized across various nodes. These modules collaborate with each other to take distributed decisions in a fault-tolerant way. It can be noted that in Aloe, infrastructure nodes are mutable and they can convert themselves as a μC if required. An interesting feature of Aloe is that this decision mechanism is executed in a pure distributed way, preserving the safety and liveness of the system in the presence of faults. The functionalities of various modules are as follows.

1) Topology Management Module (TMM): We design Aloe as a plug-and-play service, where an Aloe-supported IoT device can be directly deployed in an existing system for flexible auto-scaling support. The TMM initializes the Aloe framework on a newly deployed node. The tasks of the TIM are as follows – (i) identify the nodes in the neighborhood, and (ii) determine whether an Aloe service is running in that node. An Aloe service is of two types – (a) μC service, and (b) user application service. To find out the active nodes in the neighborhood, TMM uses *Link Layer Discovery Protocol* (LLDP). We assume that each Aloe service deployed in the IoT cloud uses a unique predefined port address. TIM queries about the services in the local neighborhood via issuing a `telnet open port` requests. Apart from the initialization, this module is invoked whenever a node/link failure or μC failure event is detected.

2) State Discovery Module (SDM): In case of a node or a link failure after the initialization through TMM, there is a possibility that the infrastructure nodes get disconnected from the μC. To identify such a scenario, Aloe maintains various state variables for each node as follows. (i) *Controller State* (CTLR): This state variable decides whether a node is in a `general` (does not host a μC service), `μC` (hosts a μC) or `undecided` (an intermediate state between the general state and the μC state) state. (ii) *Priority* (PRIO): This state variable is required only if the node is undecided and denotes the priority of the node for becoming a μC. The states associated to nodes are kept and managed by the nodes themselves. However, a node can access a copy of the states from its neighbor to decide its state. SDM is responsible for accumulating the state information collected from the neighbors. SDM uses `REST` for this purpose. Once a failure event occurs, TMM invokes the SDM. SDM keeps on executing periodically until the node finds at least one μC in its neighborhood. The periodicity of the execution of this module is dependent on the link delay. For implementation purpose, we consider the periodicity as the largest delay observed to fetch data from a neighbor.

3) μC Placement Module (μPM): Based on the neighbor states collected through SDM, every node independently determines whether it needs to launch a μC service. This is done through the μPM module. We consider the nodes as the vertices of a graph where the edges determined by the connectivity between two nodes, and place the μC services to the nodes that form a *maximal independent set* (MIS) on that graph. An MIS based μC placement ensures that there would be a μC at least in one-hop distance from each node, which can take care of the configurations and flow-initiations for the application services running on that node. As we have claimed earlier and will show in § VI that the μCs utilized in Aloe are significantly light-weight but efficient for performing network and service management activities. Therefore the total overhead due to MIS based μC placement is not significant. For identification of a suitable set of μC capable nodes, we develop a distributed randomized MIS algorithm given in Algorithm 1. The novelties of this algorithm are as follows. (1) **Randomized**: The algorithm selects different nodes at different rounds, ensuring that the load for μC service hosting is distributed across the network and does not get concentrated on some selected nodes. (2) **Bounded set**: The number of deployed μCs are always bounded based on the total number of nodes in the network. (3) **Self-stabilized**: The algorithm is self-stabilized and converges in linear time (the proof is omitted due to space-constraint), ensuring fault-tolerance of the system under single or multiple simultaneous faults until complete network partition occurs. Intuitively, we can see that; a node cannot retain its status as `undecided` forever. In a neighborhood, if it finds another μC then it changes to `general` (Line 24), otherwise, it competes with other nodes having `undecided` status in a series of tiebreaker rounds by choosing a random `PRIO` value (Line 25). Until one of the nodes receives a unique maximum priority in the neighborhood and gains `μC` status (Line 29), the random trials continue (Line 31). It can be shown that the expected number of such trials for a node is less than thrice the number of competing nodes (proof omitted due to space constraint). Therefore, the Algorithm 1 always converges in linear time.

4) μC Manager Module (μMM): Once a node decides its state through μPM, the μMM module initiates the μC service on the selected nodes and establishes a controller-switch relationship between the μC and the nodes with *general* state in the one-hop neighborhood. As we mentioned earlier, a μC is initiated as a containerized service over the node designated for hosting a μC by the μPM algorithm. For a node with *general* state, this process may involve changing of

Algorithm 1: μPM Controller Placement Algorithm

```
1  Function Trial():
2      PRIO← Rand();
       ;                                  // Breaks priority ties
3      return;
4  Function NeighborμC():
5      if Another μC in one-hop neighborhood then
           ;                              // If μC in neighborhood
6          return true
7      else
8          return false
9  Function UMPriority:
       /* If node has unique maximum priority     */
10     if PRIO of this node > maximum PRIO in neighborhood then
11         return true
12     else if PRIO of this node = maximum PRIO in neighborhood then
13         return false
14     else
15         return None
16 Function Main():
17     while state change detected do
18         if CTLR=general & NeighborμC()=false then
               // No μC in neighborhood
19             CTLR← undecided;
20             Trial();
               // Initialize priority
21         else if CTLR=μC & NeighborμC()=true then
               /* Two μCs are adjacent                */
22             CTLR← general;
23         else if CTLR=undecided & NeighborμC()=true then
               /* μC found in neighborhood            */
24             CTLR← general;
25         else
26             if UMPriority()=None then
                   /* Executor is not maximum         */
27                 continue;
28             else if UMPriority()=true then
                   /* Unique maximum priority.        */
29                 CTLR← μC;
30             else
                   /* Maximum but not unique priority
                      */
31                 CTLR← undecided;
                   /* Next round of trial starts      */
32                 Trial();
33     return
```

controller services from one μC to another μC, which requires the reestablishment of the controller-switch relationship. For this purpose, the SDN flow tables need to be migrated from the old μC to the newly associated μC. The flow table migration mechanism is specific to the SDN controller software used, and therefore, we discuss it in §V.

5) PushToNode Module (P2NM): Along with fault-tolerance, Aloe supports rapid deployment and runtime customization of the system. To implement this feature, we develop P2NM. Unlike the rest of the modules, P2NM is centralized and/or deployed in the SNC. It provides an interface for monitoring and changing the policy level information for the μC at runtime which is useful for system administrators. Aloe supported policy level information include (i) ACLs, (ii) controller application to be executed in the μC, (iii) routing protocols running in the μC, and (iv) SDM update frequency. Apart from the specified policies, Aloe also gives freedom to its user to customize the Aloe modules itself. This feature is achieved by developing a set of Application Programmer's Interfaces (API) as discussed next.

B. Application Programmer's Interfaces (API)

The primary objective of this orchestration framework is to deploy the *controller as a service* to the in-network processing infrastructure in the form of a μC. There are some significant differences between a user application service and a μC service, which makes the deployment of the later non-trivial. Unlike user application services, μC services should not act location transparently. A location transparent deployment of μC might allocate all the μCs in the same node, which can degrade the network performance of the infrastructure. Therefore, during the design of Aloe, we consider the extendibility of this work. Many of the implemented functionalities of this framework can be reused as API for distributed controller application development. For ease of understanding, we only provide the python sample programs here. However, all the APIs can be invoked as a `script` over the SNC using P2NM.

1) Topology Monitor: Using this python API, Aloe can detect a topology change event (`TopologyMonitor()`) and take actions accordingly. This API can also be used for general purpose routing application, as given in the following code.

```
''' Find shortest path between dpidS and dpidD'''
import networkx as nx
G=TopologyMonitor()
path_dpid_list=nx.algorithms.shortest_path(G,"delay"
    )
```

Listing 1: Topology Change Detector

2) Distributed State Inspector: We develop this API to observe the state of the nodes (`getNeighborStates()`), which helps in developing new placement algorithms for μPM. This API relies on a remote procedure call (`rpc`).

```
''' Find max priority amongst neighbor'''
import json
states=getNeighborStates()
maxPrioUndecided=max([v["Prio"] for v in states.
    values() if state["CTLR"]=="undecided"])
```

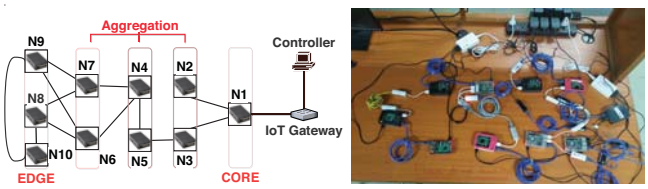
Listing 2: Distributed State Inspector

3) Find Node Services: The framework requires to identify the deployed services (`getNeighborServices()`) to enforce service level policies. We provide a python API to ease this task. The following example can be used for selective service blocking ACL.

```
'''Service blocking (ACL)'''
import json,os
services=getNeighborServices()
bport=blocking_port
if(blocking_port in service["dpid"]):
 os.system("ovs-ofctl add-flow match:src=dpid,
    tcp_port=bport action:drop")
```

Listing 3: Find Node Services

Next, we discuss the details of the Aloe implementation as a general orchestration framework.



(a) Topology (b) In Action

Fig. 3: Testbed

TABLE I: Wilcoxon Rank Sum Test (↑indicates μC in top header consumes less resources, ←indicates μC in left header consumes less resources, **X** indicates the choice is undetermined)

CPU				
μC	No μC	Ryu	Zero	ODL
No μC		←	←	←
Ryu	< 0.0001		←	←
Zero	< 0.0001	< 0.0001		←
ODL	< 0.0001	< 0.0001	< 0.0001	
Memory				
No μC		←	X	←
Ryu	< 0.0001		X	←
Zero	> 0.03	> 0.01		←
ODL	< 0.0001	< 0.0001	< 0.0001	
CPU Temperature				
No μC		X	←	←
Ryu	> 0.39		←	←
Zero	< 0.0001	< 0.0001		↑
ODL	< 0.0001	< 0.0001	< 0.0001	

V. Aloe Implementation

We have implemented Aloe as a middleware over Linux kernel with the integration of open-source technologies, like docker containers, various SDN controllers, and `REST` based communication modules. We first discuss the implementation environment that we utilized, followed by a brief description of two different implementation aspects.

A. Environmental Setup

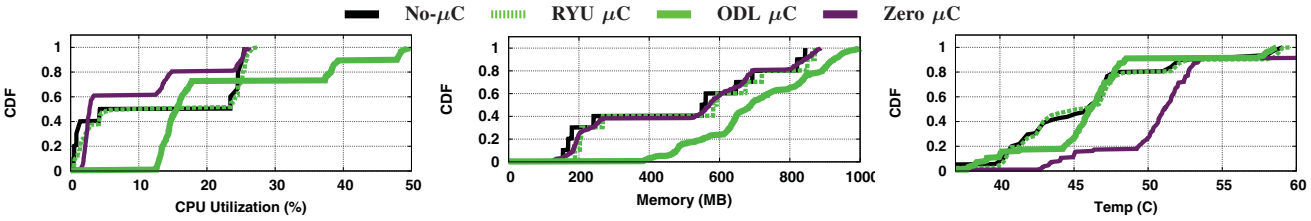
To analyze the performance of Aloe, we have deployed an in-house testbed using the topology given in Fig. 3a (Fig. 3b shows the live-snapshot of the testbed). The nodes in the testbed are Raspberry Pi version 3 Model B, which are configured with Raspbian 8.0 operating system with kernel version 4.4.50-v7+. The nodes are connected via multiple 100Mbps USB-to-Ethernet adapter-edges representing the physical Ethernet links among the nodes. We use Linux `tc` to configure each link to use 5Mbps of bandwidth and added $100ms$ of propagation delay to match real life IoT deployment specification. Further, to analyze the scalability of Aloe, we have also deployed Aloe in a large-scale 68-node testbed using Amazon Web Services (AWS). For this purpose, we consider a sub-topology from `rocketfuel` [19] topology which consist 68 nodes. The nodes in the topology are deployed using 18 AWS *nano* instances (1 vCPU and 512 MB RAM) and 50 AWS *micro* instances (1 vCPU and 1 GB RAM). The AWS nodes are configured with Ubuntu 16.10 operating system with Debian kernel version 4.4.0. To emulate edges between the nodes, we use the *Layer 2 Tunneling Protocol* (l2tp) between the AWS instances. Every infrastructure node, both in the testbed and in the AWS, are configured with *open virtual switch* (OVS).

B. Implementation Aspects

Here we discuss two important implementation aspects of Aloe – (i) flow-table consistency preservation during μC migration, and (ii) choice of controller service for μC implementation. Here, we discuss these two aspects in details.

1) Migration of μC and consistency preservation: A change in a policy level parameter requires a migration of the flow tables from the old μC instances to the new instances of the μC. Similarly, after a μPM execution, there might be a need for change in node-controller association. To implement such functionality, we have implemented a `rpc` and `REST` based API (`changeCtlr()`) which can dynamically change a switch's allegiance towards a μC. `changeCtlr()` forces the node to invoke a "`controller re-association request`" to the target μC with its previous μC address. After receiving a "`controller re-association request`", the target controller invokes migration of flow entries from the previously assigned μC. During the migration procedure, it is important to keep track of the previous state informations. To ensure the consistency, Aloe preserves snapshots of the μC flow table entries by sending `REST` queries to the μCs before the migration process starts. To make the migration process lightweight, the container instance is not transferred from one node to another node; instead, the source node container is killed, and a new container is invoked at the destination node via `rpc`. In case of a network partitioning between the previous μC and the target μC, the target μC obtains the copy of flow table from the requester node itself. In this way, the μMM preserves weak consistency in the system.

2) Choice of controller service for μC: The efficiency of Aloe is dependent on the efficiency of the choice of a controller service for the μC. Deployment of a heavy-weight controller can over-consume resources of the nodes; moreover, one μC is only responsible for managing a small set of nodes. Therefore, we target to opt for a light-weight μC for Aloe. In order to identify a suitable controller platform for μC, we compare a set of existing SDN controller services like "`Open Day light (ODL)`" [20], "`ONOS`" [12], "`ryu`" and "`Zero`" in our in-house testbed in terms of theirs resource utilization. Amongst these controllers, "`ONOS`" requires high memory consumption ($> 500MB$) which creates an instability in the docker environment. Further, we have observed that approximately 32% times, "`ONOS`" fails to execute over the testbed nodes due to unavailability of sufficient virtual memory. Therefore, we report the performance of the controllers other than "`ONOS`". The performance is reported based on three major system parameters – CPU utilization, memory utilization and CPU temperature variation. In Fig. 4a, we provide the comparison of the performance of the competing



(a) CPU Utilization (b) Memory Utilization (c) CPU Temperature Variation

Fig. 4: Resource Utilization Comparison of Controller Applications

TABLE II: Wilcoxon Rank Sum Test conclusions with p-values over response time of different applications:
X=Inconclusive, ✓=*Aloe* better, •=In band better

Services / Failure	Cassandra	HTTP	Gluster
0	X (>0.11)	X (>0.20)	•(<0.0001)
1	✓(<0.0001)	X (>0.04)	•(<0.0001)
2	✓(<0.0001)	✓(<0.0001)	X (>0.39)
3	✓(<0.0001)	✓(<0.0001)	✓(<0.01)

controllers in terms of CPU utilization. The plot suggests that approximately 30% "ODL" μCs use more than 30% of the CPU utilization. In comparison to that, around 40% "Zero" μCs use 15% of the CPU utilization. In Fig. 4b, we observe that almost 80% "ODL" μCs use more than $600MB$ of memory space. All the other controllers show lower memory utilization. Fig. 4c shows the variation in CPU core temperature while executing different types of controller services. The consolidated pair-wise comparison of the controllers are provided in Table I (upper right triangle in blue color). The notation **X** signifies that the comparison cannot be obtained. On the other hand ↑suggests that the μC listed in the top header consumes less amount of resources. We use ←to denote the higher efficiency of the μC application mentioned in the left header. To ascertain the comparison, we perform a statistical hypothesis testing using non-parametric, one-tailed Wilcoxon rank sum test ($\alpha = 0.01$) [21]. Our alternative hypothesis assumes that the mean resource consumptions and the core temperature will be higher than the one in the normal case. The left lower triangular part of Table I (given in green color) signifies the p-values obtained from the rank test. Based on our observation from Table I, we choose "Zero" as our choice of μC in testbed and AWS.

With this implementation, we evaluate the performance of Aloe, as discussed in the next section.

VI. Evaluation

We have tested the performance of Aloe with three different categories of standard applications which are common and useful for a networked IoT based system – (i) HTTP service (Python `SimpleHTTPServer`): used for bulk data transfer via web clients, (ii) distributed database service (`Cassandra`): for data-driven applications, and (iii) distributed file system service (`Gluster`): used for file sharing and fault-tolerant file replication over a distributed platform. We further compare the performance of Aloe with BLAC [17], a distributed SDN control platform. To emulate realistic fault models in the system, we have injected the faults using Netflix *Chaos Monkey* fault injection tool. We have taken the measurements under all possible fault combinations in the testbed and 100 different random fault combinations in AWS.

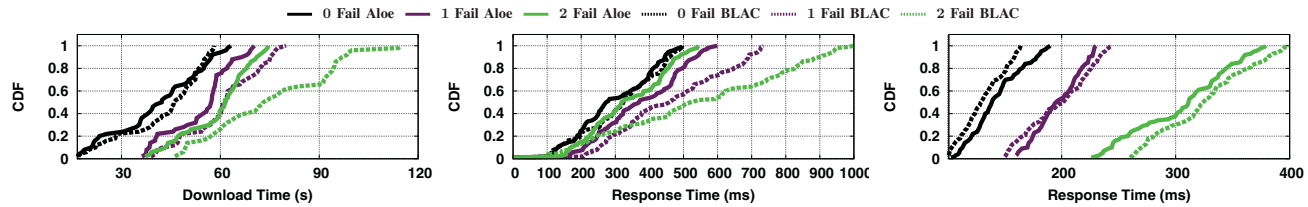
A. Application Performance

Fig. 5a compares the download time of a 512MB file hosted using the HTTP service under the influence of both BLAC and Aloe over the in-house testbed. The results are obtained by varying all possible source-destination pairs in the topology. We observe that, even though Aloe results in higher download time compared to BLAC when there is no failure in the system, the performance improves rapidly in the presence of link outage. While injecting failure, we observe that approximately 30% connections are timed-out while operating under the governance of the BLAC controller. However, Aloe reduces such flow termination[2] ($< 5\%$ connection time-out for Aloe). To compare the differences of the nature of the results for each service, we performed a Wilcoxon rank sum test. The p-values and conclusion from the test is summarised in Table II.

Fig. 5b shows the response time of `Cassandra` search queries. Here, we observe a significant difference in the characteristics of the plots due to the nature of the service. Unlike `HTTP`, `Cassandra` utilizes short flows to fetch query results. Therefore, we observe that Aloe provides a significant improvement in the query response time. However, in case of `Gluster`, Aloe performance is marginally poor compared to BLAC until there are 3 link failures (Fig. 5c). `Gluster` flows are short-distant flows, usually within one-hop. The flow-setup delay is almost negligible for a one-hop flow. Therefore the μC deployment overhead of Aloe is more when the number of failures is less.

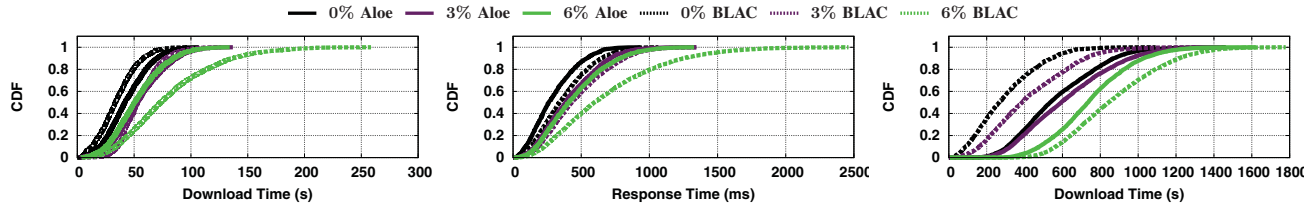
Similar behaviors are observed in the large-scale deployment of Aloe in the AWS cloud. In Figs. 6a and 6c, `HTTP` and `Gluster` response times show similar characteristics as observed in the testbed. In the case of `Cassandra` (Fig. 6b), all the cases perform significantly better than BLAC. From these observations, we conclude that Aloe performs significantly better for the services that generate long-distant mice flows (like database synchronization). For a long-distant flow, the flow setup delay is high, which gets further affected by

[2]Only in such cases, where the network is partitioned



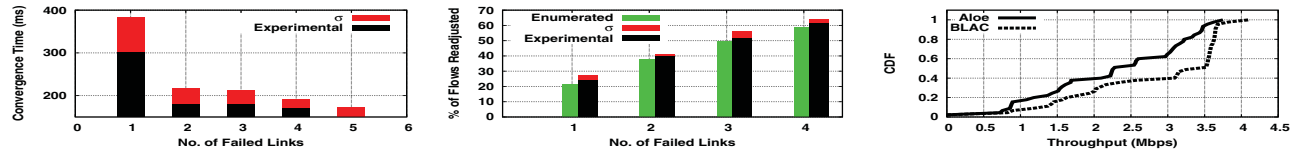
(a) HTTP Server (b) Cassandra (c) Gluster

Fig. 5: Comparison of response time of services obtained from testbed: Average percentage improvement of Aloe – (a) HTTP server: 4% (0-fail), 11% (2-fails), 21% (3-fails), (b) Cassandra: 9% (0-fail), 26% (2-fails), 37% (3-fails), (c) Gluster: -8% (0-fail), 0.1% (2-fails), 6% (3-fails)



(a) HTTP Server (b) Cassandra (c) Gluster

Fig. 6: Comparison of response time of services obtained from AWS cloud: Average percentage improvement of Aloe – (a) HTTP server: -2% (0-fail), 0.1% (2%-fails), 34% (6%-fails), (b) Cassandra: 20% (0-fail), 21% (2%-fails), 34% (6%-fails), (c) Gluster: -12% (0-fail), -6% (2%-fails), 14% (6%-fails)



(a) Avg. Aloe Convergence Time (b) Avg. No. Flow Restoration (c) Throughput comparison

Fig. 7: Testbed: Effect of failure on Aloe performance (σ= standard deviation)

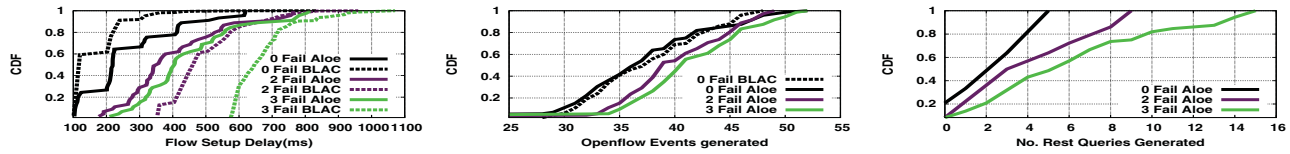
link failures. As a consequence, Aloe performance is better for failure-prone systems, like IoT clouds, as the flow-setup delay gets increased with the recovery time due to a failure.

B. Dissecting Aloe

Aloe flow-setup time is dependent upon the convergence time of μPM and the path restoration time. Fig. 7a shows the distribution of the average convergence time of Aloe in the presence of failure. We have an interesting observation here that as the number of simultaneous failures increases, the convergence time drops. This can be explained as follows. Let us consider two different faults. If the two faults are at two different sides of the network, then two waves of μPM starts executing simultaneously from two different ends of the network. These two waves get diffused in the network and meet in the middle of the network at convergence. That way, multiple faults create multiple such μPM waves in the network in parallel, and as these individual waves need to deal with a smaller part of the network, they converge quickly.

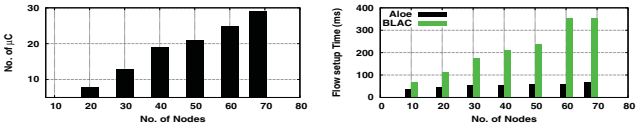
The convergence phase is followed by the path restoration due to a change of the controller positions in case of a failure. To identify the performance of the path restoration, we measure the average number of flow adjustments done by the framework. The number of flow adjustment depends on the topology and the locations of the failed links. Therefore, to compare the result, we provide the enumerated number of flow adjustment required for all possible cases of failures in Fig. 7b. We observe that the experimental observations closely match with the results obtained by enumeration. Further, these metrics have a direct impact on the flow-setup delay. To understand the effect of these factors, we compare the flow-setup delay for BLAC control plane and Aloe. To identify the flow-setup delay, we use `ping` to transfer a single `ICMP` packet. Fig. 8a shows that although Aloe marginally increases the flow setup-delay in the absence of a failure, it provides quick flow-setup when multiple faults occur in the network.

We observed that the overhead of distributed μC in Aloe is responsible for the increase in the flow-setup delay during the no-failure scenario. However, it is difficult to compare the exact overhead of the BLAC control plane and Aloe due to the difference of the nature the overhead. We measure the overhead regarding two different factors. Fig. 8b shows the comparison between BLAC and Aloe regarding the number of `openflow events` generated over a period of 100s. However, Aloe additionally generates `REST` queries to support



(a) Flow Setup Delay:Measured using `ping` (b) Control Plane Overhead (c) `REST` queries generated in Aloe

Fig. 8: Testbed: Flow setup delay and overhead comparison



(a) No. of μC (b) Avg. Flow Setup Time

Fig. 9: AWS: Effect of Scaling

inter-controller communication. Fig. 8c depicts the number of `REST` queries generated in Aloe. Due to these overheads, the Aloe incurs higher communication overhead than the BLAC control plane. However, due to the significant reduction in flow-setup time, Aloe ensures better flow throughput than BLAC, as shown in Fig. 7c.

Although Aloe incurs communication overhead, Aloe ensures a significant drop in the average flow-setup delay. To limit the flow-setup delay, Aloe provides elastic auto-scaling by increasing the number of μC instances to guarantee that each node can find a μC in its neighborhood. Fig. 9a shows the average number of μC instances when the network scales, as obtained from the AWS implementation. The effect of elastic auto-scaling is shown in Fig. 9b which indicates that the flow-setup delay only increases marginally in comparison to the BLAC controller, which incurs a significantly high flow-setup delay as the number of nodes in the network increases.

VII. Conclusion

In this paper, we present Aloe, an orchestration framework, for IoT in-network processing infrastructures. Aloe uses docker container to support lightweight migration capable in-band controllers. This design choice helps Aloe to provide elastic auto-scaling while keeping flow setup time under control. Aloe provides controller as a service to exploit in-network processing infrastructure and supports fault and partition tolerance. The performance of Aloe has been tested thoroughly using two real testbeds and compared with a very recent orchestration framework (BLAC). The results indicate a significant improvement in response times for distributed IoT services.

References

[1] A. C. Baktir, A. Ozgovde, and C. Ersoy, "How can edge computing benefit from software-defined networking: a survey, use cases, and future directions," *IEEE Comm. Surveys & Tutorials*, pp. 2359–2391, 2017.

[2] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *SIGCOMM Comput. Commun. Rev.*, pp. 27–32, 2014.

[3] M. Chiang, S. Ha, I. Chih-Lin, F. Risso, and T. Zhang, "Clarifying fog computing and networking: 10 questions and answers," *IEEE Comm. Magazine*, pp. 18–20, 2017.

[4] M. Selimi, L. Cerdà-Alabern, M. Sánchez-Artigas, F. Freitag, and L. Veiga, "Practical service placement approach for microservices architecture," in *Proceedings of the 17th IEEE/ACM CCGRID*, 2017, pp. 401–410.

[5] M. Z. Shafiq, L. Ji, A. X. Liu, J. Pang, and J. Wang, "Large-Scale Measurement and Characterization of Cellular Machine-to-Machine Traffic," *IEEE/ACM Tran. on Networking*, pp. 1960–1973, 2013.

[6] O. Kaiwartya, A. H. Abdullah, Y. Cao, J. Lloret, S. Kumar, R. R. Shah, M. Prasad, and S. Prakash, "Virtualization in wireless sensor networks: fault tolerant embedding for internet of things," *IEEE Internet of Things Journal*, pp. 571–580, 2018.

[7] L. Galluccio, S. Milardo, G. Morabito, and S. Palazzo, "SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for wireless sensor networks," in *Proceedings of IEEE INFOCOM*, 2015, pp. 513–521.

[8] D. Wu, D. I. Arkhipov, E. Asmare, Z. Qin, and J. A. McCann, "UbiFlow: Mobility management in urban-scale software defined IoT," in *proceedings of IEEE INFOCOM*, 2015, pp. 208–216.

[9] M. T. I. ul Huque, W. Si, G. Jourjon, and V. Gramoli, "Large-scale dynamic controller placement," *IEEE Tran. on Network and Service Management*, pp. 63–76, 2017.

[10] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proceedings of the 2nd ACM/IEEE SEC*, 2017, p. 11.

[11] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *Proceedings of the 9th USENIX Conference on OSDI, 2010*. USENIX Association, 2010, pp. 1–6.

[12] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "Onos: towards an open, distributed sdn os," in *Proceedings of the 3rd HotSDN, 2014*. ACM, 2014, pp. 1–6.

[13] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proceedings of the 1st Workshop on HotSDN*, 2012, pp. 19–24.

[14] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. R. Kompella, "Elasticon: an elastic distributed sdn controller," in *Proceedings of the 10th ANCS , 2014*. ACM, 2014, pp. 17–28.

[15] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," *SIGCOMM Comput. Commun. Rev.*, pp. 351–362, 2010.

[16] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *SIGCOMM Comput. Commun. Rev.* ACM, 2011, pp. 254–265.

[17] V. Huang, Q. Fu, G. Chen, E. Wen, and J. Hart, "BLAC: A Bindingless Architecture for Distributed SDN Controllers," in *proceedings of 42nd IEEE LCN*, 2017, pp. 146–154.

[18] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, and S. Shenker, "SCL: simplifying distributed SDN control planes," in *Proceedings of the 14th USENIX Conference on NSDI*. USENIX Association, 2017, pp. 329–345.

[19] "UW CSE | Systems Research | Rocketfuel," May 2005, [accessed 28. Jul. 2018]. [Online]. Available: https://research.cs.washington.edu/networking/rocketfuel

[20] (2018) OpenDaylight. [Online]. Available: http://www.opendaylight.org/

[21] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, pp. 80–83, 1945.

Amalgam: Distributed Network Control With Scalable Service Chaining

Subhrendu Chattopadhyay
IIT Guwahati
Guwahati, India 781039
subhrendu@iitg.ac.in

Sukumar Nandi
IIT Guwahati
Guwahati, India 781039
sukumar@iitg.ac.in

Sandip Chakraborty
IIT Kharagpur
Kharagpur, India 721302
sandipc@cse.iitkgp.ernet.in

Abhinandan S. Prasad
NIE Mysuru
Karnataka, India 570008
abhinandansp@nie.ac.in

***Abstract*—Management of virtual network function (VNF) service chaining for a large scale network spanning across multiple administrative domains is difficult due to the decentralized nature of the underlying system. Existing session-based and software-defined networking (SDN) oriented approaches to manage service function chains (SFCs) fall short to cater to the plug-and-play nature of the constituent devices of a large scale eco-system such as the Internet of Things (IoT). In this paper, we propose *Amalgam*, a composition of a distributed SDN control plane along with a distributed SFC manager, that is capable of managing SFCs dynamically by exploiting the in-network processing platform composed of plug-and-play devices. To ensure the distributed placement of VNFs in the in-network processing platform, we propose a greedy heuristic. Further, to test the performance, we develop a complete container driven emulation framework *MiniDockNet* on top of standard *Mininet* APIs. Our experiments on a large scale realistic topology reveal that *Amalgam* significantly reduces flow-setup time and exhibits better performance in terms of end-to-end delay for short flows.**

***Index Terms*—Service function chaining, Virtual network function, In-network processing, Programmable network, software defined network**

I. Introduction and Related Works

Due to the rapid deployments of connected environments, large-scale Internet of Things (IoT) networks [1] have become prevalent in recent years. Management of such large-scale heterogeneous ecosystems requires various network services such as network address translator (NAT), firewall, proxy, and local domain name server (DNS); these network services are called network function (NF)s. Generally, the network functions are deployed using virtual machines (VM)(s) to provide service isolation and reduce CapEx and OpEx; therefore, they are termed as virtualized network function (VNF) [1]. VNFs execution require computation platform to host the VM and execute the NF within the VM. Depending on network management policies, the application messages require steering through an ordered set of VNFs known as service function chaining (SFC) [2].

Among various existing architectures to execute VNFs over a network infrastructure [3]–[5] relies on software-defined network (SDN) [6] to steer flows from one VNF to another. On the other hand, [7], [8] takes a session-based approach where the end hosts control the SFC. Session-based approaches achieve lower host-based state management of VNFs, where SDN-based approaches achieve fine-grained quality of service (QoS). However, for a large-scale network spanning across multiple administrative domains, both of the SFC management (SCM) approaches fall short in several aspects as follows.

(a) Lack of scalability: Existing SCMs [6], [9] use a central controller that monitors the resource usage of the devices and use as the basis for the VNFs deployment. The use of a central controller for VNF deployment becomes challenging, especially when the network spans across multiple autonomous administrative domains that interconnected through different network service providers. On the other hand, the VNF placement is $\mathcal{NP}$-hard [10]. Existing distributed heuristics for VNF placement [11] require multiple rounds to deploy VNFs, which increases flow initiation delay leading to reduced IoT application performance since the majority of the IoT flows are short-lived [12].

(b) Dynamic service chaining: Usually, VNFss modifying the headers are common in a large-scale network. Consequently, the participating VNFs can change the SFCs during the lifetime of a flow based on the flow characteristics. For instance, a classifier VNF can add a load balancer based on the arrival rate of the packets in a flow. Existing scalable distributed VNF placement methods [11] and IP based traffic steering proposals [13] are not suitable for dynamic service chaining. On the other hand, [7] ascertains dynamic service chaining by adding an agent in each device, including hosts. Installation of agents on a large scale IoT becomes infeasible, where the devices with plug-and-play capability can dynamically enter and exit the ecosystem.

(c) Issues of flow monitoring over multi-administrative platforms: To steer the traffic through proper service chains while ensuring QoS, requires fine-grained flow monitoring. Existing flow identification methods using packet header fields are insufficient in the presence of a header modifying VNF in the SFC (such as NAT, load balancer, and proxy). Existing SDN-based flow monitoring schemes like FlowTags [9], Stratos [14] utilize *"vlan/mpls"* tagging which does not work through multiple administrative domains. On the other hand, the use of packet encapsulation in session-based approaches

This work was supported by TCS India Pvt. Ltd.

[1]https://www.sigfoxcanada.com/

Annex to ISBN 978-3-903176-28-7 © 2020 IFIP

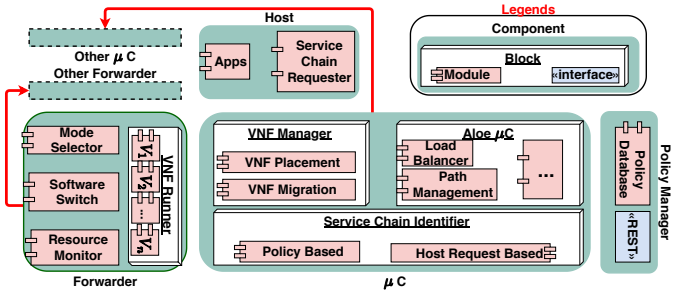


Fig. 1: Component diagram of Amalgam

[8] also fails to ensure QoS.

Therefore, in this paper, we propose *Amalgam* to provide a scalable SFC orchestration platform to provide fine-grained QoS over multiple administrative domains and dynamic SFC. The SCM *Amalgam* provides significant performance improvement in terms of flow initiation delay. The step by step contributions is as follows.

- *Amalgam* achieves distributed state management while ensuring fine-grained QoS by coupling distributed SDN and distributed SCM.
- We develop a distributed heuristic for the distributed deployment of VNFs, which provides performance improvement in terms of flow initiation delay.
- *Amalgam* design exploits the "micro-service"(μS) architecture of the in-network processing platform [15] to support dynamic service chaining and "`zero-touch deployment`" to support the plug and play nature of the devices.
- For performance evaluation, we develop an emulation framework *MiniDockNet* for VNF deployment using "`docker`" [2] over in-network processing, as the existing network name-space oriented mininet [3] emulator is not sufficient for in-network processing.
- Based on the experimental results over a realistic large-scale system (with 70 devices and 6 different service chain scenarios), we found that *Amalgam* can ensure fine-grained QoS without significant increase of resource utilization of the devices. After comparing the performance of *Amalgam* with two state-of-the-art distributed SFC platform *"Dysco"* [7] and *"WGT"* [11] we found that, *Amalgam* is capable of a significant reduction in the flow initiation delay and improves the performances of short-duration flows in terms of end-to-end delay.

This paper is organized as follows. Section II describes the architecture of proposed *Amalgam*. Section III describes the design choices taken during the development of *Amalgam*. We describe the experimental setup details, results and our observations in Section IV before concluding in Section V.

II. Architecture

The proposed *Amalgam*[4] is constructed on the top of *Aloe* [16] framework. *Aloe* provides an orchestration framework for a fault-tolerant self-stabilizing distributed control plane on top of the in-network processing platform using μCs (microcontrollers) instead of the standard SDN controller. Like *Aloe*, *Amalgam* supports 3 operating modes for each device; (a) **Host/(default) mode**: if the device executes only the client/server application, (b) **Forwarding mode**: If the device has multiple active network interfaces, (c) **μC mode**: based on the topology, if *Aloe* selects the device as a μC. At any instant, each device is in "`atleast`" any one of the above modes and each mode is a component of *Amalgam* framework as shown in Fig. 1. A mode selector module in *"forwarder"* component identifies the mode of operation and activates the respective components. Apart from the mode functional components, we add a seperate *"policy manager"* component in the *Amalgam* framework. Policy manager is a distributed database with a *"REST"* driven interface which contains (i) the flow identifier (i.e., "`OpenFlow`" match field) and (ii) ordered list of the types of VNF (service chain) through which the flow should be steered. This component is consulted by the mode functional components at time of determination of SFC.

Among the mode functional components, the host is the simplest. This component is responsible for traffic generation through the *"App"* module, which represents the client/server applications. Additionally, this module can request the nearest μC to change the SFC for the flows generated by the host. The *"mode selector"* module elevates the mode of the device with multiple active interfaces to *"forwarder"*. The forwarder component consists of *"software switch"*, *"VNF runner"*, and a *"resource monitor"*. The software switch module is responsible for forwarding data from one interface to another. On the other hand, the *"VNF runner"* block is reserved for execution of VNFs (e.g., V_1, V_2 etc. as shown in Fig. 1). This VNF runner block from each device constructs the in-network processing framework. During the execution of the VNFs *"resource monitor"* module periodically monitors the available resources in the device. The resource monitor module forwards the collected resource utilization statistics to the μC component.

The μC component is composed of three functional blocks namely Service chain identifier (SCI), ""*VNF manager*"", and "*Aloe* μC". The tasks of these blocks are as follows.

1) Service Chain Identifier: At the startup phase of the μC, SCI caches the policy in a local cache. The local cache is updated whenever the policy manager database is updated. SCI module is consulted when an "`OpenFlow`" "`packet in`" event is initiated at the μC. From the list of VNFs in the service chain, SCI chooses the first VNF, and it's execution status in the local domain. If the VNF is executing inside a forwarder connected to the μC, the SCI consults the path management module to establish the data flow path by installing flow table entries via standard "`OpenFlow`" protocol. Otherwise, it sends a search query to the other μCs to identify the target VNF address. If the address of the VNF is not found, then SCI consults the VNF manager module (Section II-2) to start the execution of the VNF. This procedure is iterated for all the VNFs in the service chain.

[2]https://www.docker.com/
[3]http://mininet.org/
[4]https://github.com/subhrendu1987/NFV_MiniDockNet

2) VNF Manager: The VNF manager module (VMM) works in a distributed fashion and communicates with the neighbor μCs. VMM tries to answers the following two questions; (a) should the VNF be placed in any of the forwarders associated with the μC? (b) which forwarder should take care of the VNF?. The detailed protocol to find an answer to these questions is described in next section. Additionally, VMM also takes care of the dynamic addition or removal of the VNFs to an ongoing flow.

3) Aloe μC: The *Aloe* μC module is the containerized SDN μ controller module as described in [16]. *"Aloe"* provides a distributed fault-tolerant controller module suitable for in-network processing frameworks that are responsible for path management. Use of *Aloe* ensures quick flow initiation along with fault and partition tolerance in *Amalgam*. However, during the design of *Amalgam*, we face several challenges exclusive to SFC deployment of IoT

III. Challenges and Design Choices

The goal of *Amalgam* is to provide a highly dynamic in-network processing platform. In this section, we describe the implementation challenges and the proposed solutions to overcome the scalability issues without affecting the dynamic behavior of the platform.

(a) Plug-and-Play Capability:A typical IoT platform is composed of plug-and-play devices where "`zero-touch deployment`" [5] is highly desired. It is necessary to configure a new device as soon as it enters the eco-system. To avoid individually configuring the devices, we design each component of *Amalgam* (except the host component) as Docker containers. Once a device enters the eco-system, it assumes the host mode of operation. Since the host mode does not require anything more than the IoT applications (clients and servers), they can work smoothly. Whenever the device wishes to change its mode, it can pull the container image of the *Amalgam* component from the nearest forwarder.

(b)Distributed VNF Placement: In a short-lived flow heavy system, minimization of the flow initiation delay is critical. The flow initiation delay consists of following components namely (a) Controller consultation delay (b) SFC deployment delay, and (c) path setup delay. The proposed VNF placement reduces the SFC deployment delay. A SFC for a particular flow is composed of multiple VNFs, which requires resource consumption. Each device of a IoT in-network processing platform has residual resources that can be used for deployment of these VNFs's. The proposed VNF placement identifies the set of devices where the VNFs of the SFCs can be placed for a given network and flow profile while satisfying the capacity constraints of the devices. Maintaining capacity constraints in a multi-domain system is non-trivial since the residual capacity of a device residing in a different administrative domain is difficult to collect. Therefore, we propose the greedy heuristic as given in the Algorithm 1.

[5]https://www.etsi.org/technologies/zero-touch-network-service-management

Algorithm 1: Distributed Placement of VNF

Function `GreedyPlace` (*Path:* P^a, *Service Chain:* C^j, μC: l):
  Find ordered set of unplaced VNFs from C^j;
  $I \leftarrow \{i : i \in P^a, \varphi_i = l\}$;
  Place as many VNFs as possible among I;
  return number of VNFs placed;

Function `Main` (*Flow:* f^j,μC: l):
  /* Find VNF placement profile for f^j in φ_i */
  Find set of paths (P) from s_j to d_j by querying "Path Management" module of l;
  $\underset{P^a \in P}{maximize}$ `GreedyPlace` (P^a, C^j, l);
  if $\exists c_{j,k} \cdots c_{j,j_{max}}$ *not placed* **then**
    /* All devices under l */
    Obtain the list of adjacent μC of l and store it in $N\mu$**foreach** $l' \in N\mu$ **do**
      `Main` (f^j, l');
  return;

Each μC in the end-to-end path (P) executes the proposed heuristic for each flow (f^j represents jth flow) from source (s_j) to destination (d_j). We denote SFC of f^j with C^j. Certain μC with ID l maintains the topology information as the list of devices (D_l) and list of links (E_l) where each link $e_{i,i'} \in E_l$ represents the physical connection between two devices (i and i'). For the sake of simplicity, we denote the μC associated with $i$as φ_i. The proposed heuristic identifies a path P^a between s_j to d_j from the set of P such that, most of the VNFs of C^j are placed near s_j in a distributed fashion. This way, one μC does not need the resource utilization of devices from other administrative domains. Once the flow is established, the resource utilization of devices in the path (`info`) is piggybacked with the data packets. The VNF manager can re-solve the Algorithm 1 and find a new allocation of VNF with updated utilization.

(c)Migrations of the VNFs: A VNF may be relocated during (a) VNF readjustment due to prior sub-optimal placement and (b) addition or removal of the device. The μC nearest to the source node of the flow decides the VNF readjustment after it receives the piggybacked resource utilization of the devices. On the other hand, the addition and removal of devices trigger "`topology_change`" event, and the local μC initiates the decision about the VNF deployment. In both cases, the decider μC starts the migration process at the source device. Initially, the source device saves a snapshot of the executing container, and the snapshot is transferred and restored in the target device. Finally, the μC updates the existing flow table entries accordingly.

(d)Dynamic management of service chains: *Amalgam* provides support for dynamic service chaining. Dynamic service chaining enables VNFs to meet changing service requirements. For instance, consider a flow passes through a firewall VNF. Based on the signature of the flow, the firewall conditionally decides to steer the flow through an additional deep packet inspector (DPI) without interrupting the flow. To implement this, *Amalgam* allows the VNFs to interact with the local μC via "`REST`" interface. The local μC can deploy the DPI if it is not available and sends the "`OFPT_FLOW_MOD`" events to

the forwarder component to enable the flow steering without terminating the flow.

(e)Flow Tags for Monitoring: Once the VNFs are placed, the path management module of μCs set-up the flow table entries of the participating forwarders via `OpenFlow` protocol. One issue regarding path management through service chains is to identify an end-to-end flow that arises in the presence of the "5-tuple" changing VNFs (e.g., Load balancer, web proxy cache, and NATs). Since such VNFs may alter the packets in unpredictable ways, fine-grained management and monitoring of the flows passing through becomes difficult. To avoid this issue, *Amalgam* attaches a "`VLAN`" tag to the packets before it enters the VNF. The nearest μC of the VNF maintains a table for flows like f^a, which keeps track of the original match field of the flow and the modified field (alias).

(f)Providing QoS: *Amalgam* is developed on top of the SDN decentralized control plane, which enables us to ensure flow specific QoS guarantees. On the other hand, since the VNF deployment is done using containers, using "`cgroups`"can ensure the VNF specific QoS like reservation of CPU, Memory, etc. The policy server module contains the "`cgroups`" parameters for each VNFs of a service chain, which is used to ensure VNF specific QoS.

IV. Prototype and Experimental Results

The existing namespace oriented emulation frameworks (e.g. Mininet) is not suitable for VNF migration and in-networking platform emulation. Therefore, we develop docker based MiniDockNet which mimics real-life VNFs using the *"Docker-in-Docker"* configuration. This feature ensures rapid deployment from MiniDockNet emulation to real in-network processing environment. To implement the VNF migration, we use standard live container migration using CRIU[6]. For the emulation of the links between any two nodes, we use *"l2tp"*

A. Experimental Setup

For experimental purpose, we use "`rocketfuel topology`"[7]. Each link is configured to emulate $3ms$ of delay and $10Mbps$ of bandwidth using linux "`tc`" utility. We use "`iperf`" to generate long flows; for shorter flows we use "`ping`". The clients and server applications are hosted on the *diameter* of the topology. For background traffic we use python based "`HTTP`" client and server.

We use "`Apache cassendra`" to implement the policy server module. Rest of the *Amalgam* modules targeted for μC are implemented on top of "`Ryu`"[8], a python based SDN controller framework. For experiments, we use 3 different VNFs (NAT (N), Load Balancer (L) and Web Proxy(W)) to create 6 different combination of service chain as given in Fig. 5. In order to ensure the confidence on the results, each experiment is repeated atleast 30 times.

[6]https://criu.org/Live_migration

[7]http://tiny.cc/nv70mz

[8]https://ryu.readthedocs.io/en/latest/

B. Results

We compare the performance of *Amalgam* with the existing *"P4"*[9] based distributed session-oriented service function chaining framework called Dysco [7]. Since, Dysco ensures session related performance and does not provide any VNF placement strategy, the performance evaluation of the proposed distributed VNF placement algorithm is done with another existing work WGT [11] which proposes a distributed heuristic for VNF placement for the multi-domain network.

1) Session Related Performances: Fig. 2 shows the comparison between Dysco and *Amalgam* in terms of flow initialization delay. We found that *Amalgam* is capable of quicker flow initialization than Dysco. This reduction in flow initialization delay comes from the parallel deployment of VNFs as opposed to the hop by hop deployment of VNFs in Dysco. The advantage of flow initialization delay becomes much evident in the case of longer service chains like C^6 than the smaller service chain like C^1.

Since *Amalgam* uses containers to deploy the VNFs as opposed to the P4 applications used in Dysco, the deployment of VNFs using *Amalgam* incurs greater latency, as shown in Fig. 3. The increase in VNF deployment time for *Amalgam* depends on the VNF container size. Therefore, the deployment latency is higher for C^6 in compared to C^1. However, in a large scale network, VNF deployment events are far rare than a flow generation event. On the other hand, the use of containers provide greater flexibility as the creation of new middlebox application using container requires less programming overhead than the creation of a new P4 application. As a result, state management during the migration of VNFs from one node to another becomes easy when they are running inside a container as compared to the P4 applications of Dysco. However, these management benefits of containers come at the cost of resource utilization.

The placement of VNFs requires resource occupancy in the deployed devices, which is an important aspect of resource constraint IoT devices. In Fig. 6, we compare the performance of *Amalgam* with Dysco in terms of CPU utilization of devices due to the placement of VNFs. In order to normalize the additional resource consumption of *Amalgam* due to the use of containers, we also compare the resource utilization of *Amalgam* without using docker. Similarly, we provide a comparison of memory utilization for *Amalgam* and Dysco in Fig. 7. Based on these two experiments, we observe that Dysco incurs less utilization of resources than the proposed *Amalgam* with the container. However, based on the *"Wilcoxon Rank Sum test"* we find that, the difference of resource utilization of *Amalgam* without Docker and Dysco is statistically insignificant (i.e. $p-value > 0.05$) for C^4, C^5 and C^6. Fig. 8 shows the comparison of throughput between *Amalgam* and Dysco. The Wilcoxson rank sum test reveals that the throughput between *Amalgam* and Dysco are statistically indistinguishable (Here our alternate hypothesis H_a is *Amalgam* provides less throughput than Dysco).

[9]https://p4.org/

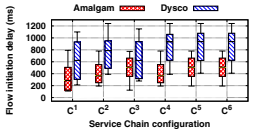


Fig. 2: Flow Initialization Delay

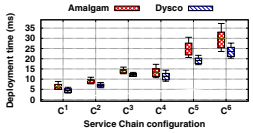


Fig. 3: Latency of Deployment

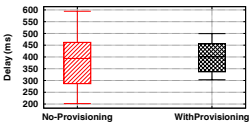


Fig. 4: Effect of QoS

Name	SFC	Name	SFC
C^1	(N)	C^2	(L)
C^3	(W)	C^4	(N→L)
C^5	(L→W)	C^6	(N→L→W)

Fig. 5: List of Service Chains

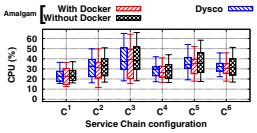


Fig. 6: CPU Utilization

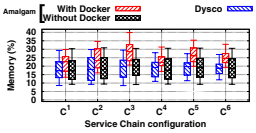


Fig. 7: Memory Utilization

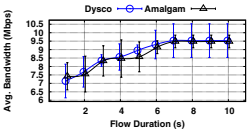


Fig. 8: Average Throughput

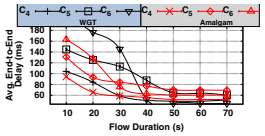


Fig. 9: Delay for C^4, C^5 and C^6

2) Performance of Distributed VNF Placement: To measure the performance of distributed VNF placement heuristic used in Amalgam, as mentioned earlier, we deploy WGT [11] on top of Dysco. However, it is difficult to deploy a centralized controller for a large scale multi-domain system. Therefore, we place the WGT in the micro-Controller (μC) nearest to the source device. We measure and compare the effect of delay for all the service chains when the flow duration increases. Based on the experimental results, we found that the effect of delay for single VNF does not change since *Amalgam* and WGT provides the same results for VNF placement. Hence, we omit the plots for C^1, C^2, C^3. For multiple VNF oriented service chains like C^4, C^5 and C^6, we provide the average end-to-end delay in Fig. 9. Based on the results, we can observe that *Amalgam* can perform significantly well for shorter flows as the iterative WGT requires a significant amount of feedback rounds to find the proper placements of VNFs.

C. QoS Provisioning

Amalgam is capable of showing QoS provisioning by reserving resources limiting CPU, memory, bandwidth, and link delay. We perform two experiments for each resource type, one with no provisioning and another with resource reservation limit set as the mean value found in the previous experiment. Based on the Wilcoxson rank-sum test on these results we found that, except the memory utilization (P-value $= 0.42$) rest of the resource reservation works significantly well (with P-value< 0.05). We also find that the resource reservation can reduce the jitter of the flow, as shown in Fig. 4.

V. Conclusion

In this paper, we present *Amalgam*, which integrates the distributed SDN orchestration framework with the distributed service chain management framework. The proposed *Amalgam* is suitable for large scale multi-domain IoT in-networking platforms. We also provide a distributed heuristics for the placement of constituent VNFs of service chains. The lack of an existing emulation platform for container oriented VNF service chain has motivated us to develop "MiniDockNet". Using this emulation platform, we found that *Amalgam* incurs a lesser flow initialization delay than that of a very recent distributed service chain management framework (Dysco). We also show that *Amalgam* is capable of ensuring less end-to-end delay for short flows.

References

[1] F. Duchene, D. Lebrun, and O. Bonaventure, "Srv6pipes: enabling in-network bytestream functions," in *17th IFIP Networking Conference and Workshops*, May 2018, pp. 1–9.

[2] W. Haeffner, J. Napper, M. Stiemerling, D. Lopez, and J. Uttaro, "Service function chaining use cases in mobile networks," *IETF*, 2015.

[3] D. Bhamare, R. Jain, M. Samaka, and A. Erbad, "A survey on service function chaining," *J. Netw. Comput. Appl.*, vol. 75, no. C, pp. 138–155, nov 2016.

[4] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *10th USENIX Symposium on NSDI*. Lombard, IL: USENIX, 2013, pp. 227–240.

[5] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," *SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 163–174, 2015.

[6] A. Mohammadkhan, S. Ghapani, G. Liu, W. Zhang, K. K. Ramakrishnan, and T. Wood, "Virtual function placement and traffic steering in flexible and dynamic software defined networks," in *The 21st IEEE International Workshop on LAN/MAN*, April 2015, pp. 1–6.

[7] P. Zave, R. A. Ferreira, X. K. Zou, M. Morimoto, and J. Rexford, "Dynamic service chaining with dysco," in *30th Proceedings of the ACM SIGCOMM*. ACM, 2017, pp. 57–70.

[8] IETF, "Rfc 8300 - network service header (NSH)," Sept 2019, [accessed 10. Sept. 2019]. [Online]. Available: https://tools.ietf.org/html/rfc8300

[9] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *11th USENIX Symposium on NSDI*. Seattle, WA: USENIX Association, 2014, pp. 543–546.

[10] M. Ghaznavi, N. Shahriar, S. Kamali, R. Ahmed, and R. Boutaba, "Distributed service function chaining," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2479–2489, Nov 2017.

[11] G. Sun, Y. Li, D. Liao, and V. Chang, "Service function chain orchestration across multiple domains: A full mesh aggregation approach," *IEEE Transactions on Network and Service Management*, vol. 15, no. 3, pp. 1175–1191, Sep. 2018.

[12] M. Z. Shafiq, L. Ji, A. X. Liu, J. Pang, and J. Wang, "Large-scale measurement and characterization of cellular machine-to-machine traffic," *IEEE/ACM Transactions on Networking*, vol. 21, no. 6, pp. 1960–1973, Dec 2013.

[13] A. Wion, M. Bouet, L. Iannone, and V. Conan, "Let there be chaining: How to augment your igp to chain your services," in *18th IFIP Networking Conference*, May 2019, pp. 1–9.

[14] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, "Stratos: A network-aware orchestration layer for middleboxes in the cloud," *arXiv CoRR*, vol. abs/1305.0209, 2013.

[15] M. Charikar, Y. Naamad, J. Rexford, and X. K. Zou, "Multi-commodity flow with in-network processing," in *Algorithmic Aspects of Cloud Computing*. Cham: Springer International Publishing, 2019, pp. 73–101.

[16] S. Chattopadhyay, S. Chatterjee, S. Nandi, and S. Chakraborty, "Aloe: An elastic auto-scaled and self-stabilized orchestration framework for iot applications," in *37th IEEE Proceedings of INFOCOM*. Paris, France: IEEE, 2019, pp. 802–810.

DisProTrack: Distributed Provenance Tracking over Serverless Applications

Utkalika Satapathy*, Rishabh Thakur*, Subhrendu Chattopadhyay†, Sandip Chakraborty*
*IIT Kharagpur, Kharagpur, India 721302 †IDRBT, Hyderabad, India 500057

***Abstract*—Provenance tracking has been widely used in the recent literature to debug system vulnerabilities and find the root causes behind faults, errors, or crashes over a running system. However, the existing approaches primarily developed graph-based models for provenance tracking over monolithic applications running directly over the operating system kernel. In contrast, the modern DevOps-based service-oriented architecture relies on distributed platforms, like serverless computing that uses container-based sandboxing over the kernel. Provenance tracking over such a distributed micro-service architecture is challenging, as the application and system logs are generated asynchronously and follow heterogeneous nomenclature and logging formats. This paper develops a novel approach to combining system and micro-services logs together to generate a Universal Provenance Graph (UPG) that can be used for provenance tracking over serverless architecture. We develop a Loadable Kernel Module (LKM) for runtime unit identification over the logs by intercepting the system calls with the help from the control flow graphs over the static application binaries. Finally, we design a regular expression-based log optimization method for reverse query parsing over the generated UPG. A thorough evaluation of the proposed UPG model with different benchmarked serverless applications shows the system's effectiveness.**

***Index Terms*—Distributed provenance tracking,**

I. Introduction

Modern service-oriented architecture adopts DevOps [1], [2] practices and technologies to provide Software as a service (SaaS) [3] by leveraging distributed cloud infrastructure. Service deployment on top of the cloud widely adopts serverless computing (SLC) [4] to reduce operational expenditure whenever the service computations are stateless, elastic, and possibly distributed. Micro-services deployed on top of SLC architecture provide an abstraction of the underlying infrastructure where the developer can write, deploy and execute the code without configuring and managing the shared environment [4]–[7]. However, the available serverless-specific industry solutions [8]–[10] provide limited support for error reporting, execution tracing, and provenance tracking. Consequently, developers can only provide little attention to log vital forensic information. Some of the third-party observability tools [11]–[14] support distributed tracing as well as cost analysis features by instrumenting the source codes. However, these tools only support applications developed using a particular programming language. So, it is difficult to analyze the actual behavior of these micro-services.

Provenance Graphs: The non-invasive [1] frameworks rely on the logs generated by applications to identify the execution states. Since most of the production-grade micro-services are chosen from an available stable release, the executable files already contain meaningful log messages that can be used to identify event handling loops. In the domain of system security, provenance data is the metadata of a process that records the details of the origin and the history of modification or transformation that happened over time throughout its lifecycle [15]–[20]. The graph generated from this information is called the *provenance graph* of a process which is a causal graph that stores the dependencies between system subjects (e.g., processes) and system objects (e.g., files, network sockets). For example, an application event may generate a separate application log, error log, and operating system calls specific log, etc. In this context, a provenance graph is a directed acyclic graph (DAG) where individual log entries are the nodes, and the edges represent the causality relationship between the log entries. During attack investigation, an administrator queries this graph to find out the root cause and ramifications of an event. While a malicious entity performs some illegal events, the corresponding system logs are recorded as the provenance data. For example, when a compromised process tries to open a sensitive file, the OS level provenance can record the file-open activity, which can be referred for vulnerability analysis whenever an attack is detected in the system [21], [22]. Real-world enterprises widely use kernel or OS level information (logs) to perform provenance analysis to monitor their systems and identify the malicious events performed back in time.

A. Limitations of Existing Works and the Research Challenges

There have been several works that attempted to generate the provenance graphs by combining system and application logs together [16]–[20], [23]. However, these works primarily considered a monolithic application running directly over the Kernel, and thus combining the system log with the application log is not difficult. In contrast, the individual log files for each micro-service over an SLC application are physically distributed across the entire eco-system, which makes the generation of the provenance graph non-trivial. In such a scenario, a Universal Provenance Graph (UPG) that combines the interactions among all the micro-services can provide a meaningful platform for distributed provenance tracking. A few existing works [20], [24] have tried to address the issue of encoding all forensically-relevant causal dependencies regardless of their origin. Nevertheless, we have some non-trivial

[1]The non-invasive tools neither inject any piece of code inside the micro-service/container instances nor injects any special services into the SLC platform.

challenges in constructing a UPG for an SLC application.

• Challenge (1) – *Combining application logs from different micro-services:* Different micro-services generate separate logs that vary across the format for log messages, naming of the events and process descriptors, timestamp formatting, etc. Combining these logs towards generating the UPG is non-trivial as they may result in confounding nodes and edges in the graph.

• Challenge (2) – *Combining the system log with the application logs:* The next challenge comes in terms of combining the application logs with the system log (kernel `audit log`). The first issue is that the container-based sandboxing uses a common process identifier (`pid`) space; thus, mapping the kernel process logs with the micro-services event logs is not straight-forward, particularly when a micro-service run a multi-threaded process.

• Challenge (3) – *Identification of execution units:* As different micro-services may come from different production endpoints, there exists heterogeneity in terms of their implementation standards. Thus it is non-trivial to identify the functions or execution units that generate a specific entry in the log. In the same context, it is also difficult to identify the event handling loops, as the loops might produce asynchronous log entries. This problem magnifies in the case of SLC as the micro-services are deployed in different sandboxed environments.

• Challenge (4) – *Dependency explosion and handling confounding root causes:* To find out the root cause behind an event, the system administrator needs to execute a reverse query on the UPG. The results obtained from the reverse query can be multiple due to partial matching of the input query string. This results in more than one root cause behind a query event. Further, due to plausible circular dependencies among the micro-services, the resultant UPG might not be a DAG. Therefore, it is non-trivial to track all the causal paths behind an event.

Owing to the above challenges, in this paper, we develop a provenance tracing system, `DisProTrack` that generates a UPG which helps in root cause analysis of an event running on serverless by combining the system provenance with application's container logs. The core idea behind `DisProTrack` is to judicially use the applications' control flow graph (CFG) to avoid the runtime log tracking complexities.

B. Our Contributions

In contrast to the existing works, our contributions in this paper are as follows.

1. Design of the UPG from application and system logs: We implement a static analyzer module that generates the application-specific *Log Message String - Control Flow Graph* (LMS-CFG) from the application binaries. The LMS-CFG provides a profile of the application. We provide a novel approach for constructing a UPG from application logs and system logs using the LMS-CFG profiles for different application micro-services.

2. Runtime execution unit identification: We develop a Linux LKM (loadable kernel module) which can intercept the system calls generated during execution time to identify the semantic relationship between the system logs and the application logs. Furthermore, we propose a heuristic to segregate execution units which are challenging in a distributed system. Our proposed heuristic identifies the system calls (syscalls) to mark the application's event handling loops by tracing back the application binaries. These event handling loops can be refereed during the runtime partitioning of execution units across the micro-services.

3. Utilization of Regular Expression to improve search efficacy: Instead of storing the raw log messages in the UPG, we propose conversion and storage of an equivalent regular expression. This method improves the matching accuracy of log messages during the investigation phase and reduces the runtime search complexity by providing a faster response time. This method also reduces dependency explosion by decreasing the number of nodes in the generated UPG.

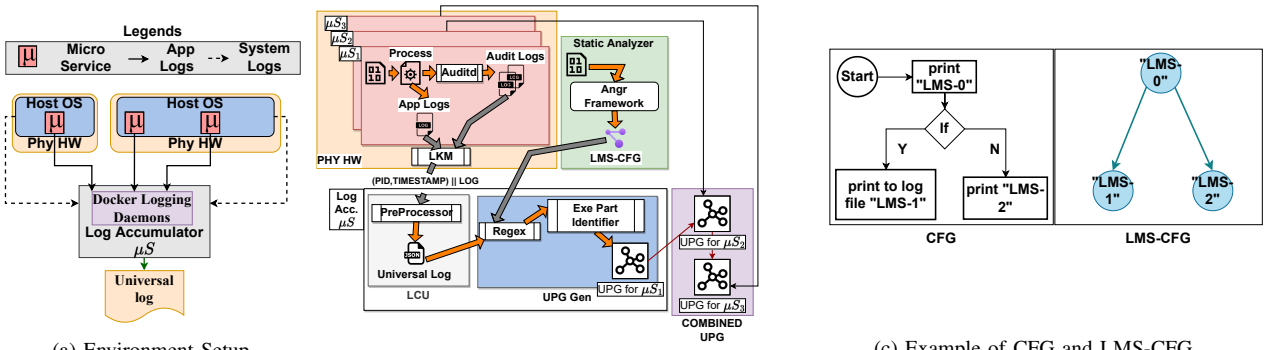
4. Implementation and evaluation: We have implemented the proposed framework, which can be deployed as a micro-service on top of the SLC without instrumenting the source code of the applications. We have made the implementation open-sourced[2]. Based on the experimental evaluation of `DisProTrack` with several benchmarked SLC applications, we found that the proposed method works well for identifying adversary activities. The framework has a minimal memory footprint (in the order of KB) and responds within $20s$-$30s$. The efficiency and efficacy of `DisProTrack` have also been tested with a proof-of-concept SLC application scenario.

II. Related Work

Existing third party log collection tools like, `FUSE` [25], `LSM` [26], `CamFlow` [27], `SPADE` [28], etc., allow hooking the kernel level objects and system calls; however, these methods require instrumentation of the OS. Additionally, the collection of system-level provenance in a containerized or serverless environment is difficult due to the micro-services' distributed and minimalistic deployed nature. Therefore, the existing threat detection and investigations with system-level provenance graph using kernel audit-log data, such as, `ETW` [29], `SLEUTH` [30], `Pagoda` [31], `POIROT` [32], `HOLMES` [33], `WATSON` [34], etc., do not suit an SLC environment. One common challenge for causality analysis with system provenance graph is the *"Dependency Explosion"* problem [35], [36] where too many *"root causes"* are behind one suspicious event. This dependency explosion increases the probability of *"security alert fatigue"* and *"missing threats"*.

`BEEP` [36] and `OMEGALOG` [16] have addressed dependency explosion problem by increasing input and output edge identification accuracy in the provenance graph. Other methods

[2]https://anonymous.4open.science/r/Project_ALV_2022-CEFD/ (Accessed: January 13, 2023)



(a) Environment Setup

(b) DisProTrack Execution Flow

(c) Example of CFG and LMS-CFG

Fig. 1: DisProTrack Overview

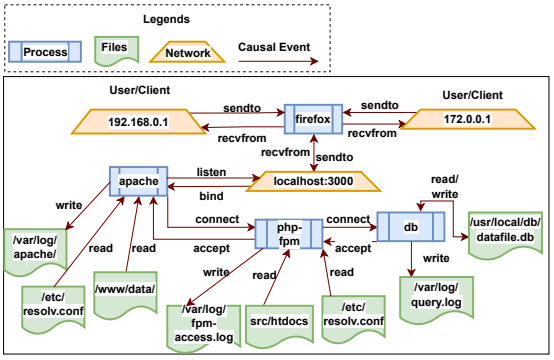


Fig. 2: An Example of System Provenance

like ETW [29], LogGC [37], MPI [35], ProTracer [38], etc., used execution partitioning to reduce the dependency explosion. However, most of these works require some instrumentation over the application source code. On the other hand, ALCHEMIST [20] and OMEGALOG used a combination of application-specific logs with system-level logs to track the information flow more accurately. In the same line, UIScope [39] proposed an instrumentation free, execution partition-based causality analysis for attack investigation system. However, the existing works are targeted toward monolithic systems and can not be deployed to secure SLC applications. Although, a few existing tools like, X-trace [40] and PivotTracing [41] are used to instrument SLC applications, they do not provide non-invasive property. Similarly, [11]–[14] are reliant on the used programming languages of the micro-service applications; thus lack flexibility [24]. The absence of language-independent, non-invasive causality analysis frameworks for SLC has motivated us to design DisProTrack.

III. DisProTrack Overview

The proposed DisProTrack is an open source provenance tracking system for containerized SLC as shown in Fig. 1a. DisProTrack accumulates the system and application logs of all the micro-service containers (μS) to generate the underlying directed edge-labeled provenance graph. The nodes of the provenance graph represent the system artifacts, for example, processes, socket connections, files, etc. The edges represent the causal dependencies between the nodes. Each edge is labeled with the generated system call (syscall) events (e.g. *read*, *write*, *connect*, *exec*, etc.) as shown in Fig. 2. The accumulated logs are merged together to create a *"universal log"* which is further used to identify the nodes of the provenance graph. Additionally, the framework also analyzes the CFG of the individual applications to understand the event handling loops before the deployment. So, the provenance graph generation requires two phases and, depending on that, DisProTrack is sub-divided into two major components; (a) *static analyzer* and (b) *runtime engine* as shown in Fig. 1b.

A. DisProTrack Static Analyzer

The static analyzer module analyzes the application executables and generates a semantic relationship between multiple *Log Message String* (LMS). Here, we define a LMS as a string present in the executable responsible for printing some log message. Typical LMSes contains format specifiers, error codes, debug level identifiers, etc. Since we only require the causal relationships between LMSes, the static analyzer identifies only the *Log Message Generating Functions* (LMGF).

Definition 1 (Log Message Generating Functions)**.** We define a LMGF as a library function that is directly used for printing a LMSes in either terminals, specific log store files, or log databases.

For example, consider the following C code snippet with a popular logging library Log4C.

```
log4c_category_log(NULL,
LOG4C_PRIORITY_ERROR,"Hello World!");
```

Here the LMS is Hello World! and the LMGF is log4c_category_log. More details about LMGF is described in Section IV-A.

Definition 2 (Log Message String CFG)**.** A LMS-CFG is formally defined as a directed graph $G' = (V', E')$ where $\forall_{e'_{i,j} \in E'} e'_{i,j} = (v'_i, v'_j) : v'_i, v'_j \in V'$ represents a directed edge between v'_i, v'_j where each $v' \in V'$ represents an LMGF. G' is constructed from CFG $G = (V, E)$ such that, $V' \subseteq V$ and,

$\underset{e' \in E'}{\exists} e' = (v'_i, v'_k) : v'_i, v'_j, v'_k \in V', \nexists_{v'_j} v'_j \in \mathfrak{P}(G, v'_i, v'_k)$. In this case $\mathfrak{P}(G, v'_i, v'_k)$ represents the directed path from v'_i to v'_k in G.

An example of a CFG and the corresponding LMS-CFG is provided in Fig. 1c where the constructed LMS-CFG contains only the LMS nodes from the CFG.

1) Contextualization of application log and system log (handling Challenges 1 & 2): The LMS-CFG is stored separately and frequently consulted by the runtime engine. During the execution, when the different levels of logs are generated, the constructed LMS-CFG is used to understand the relationship among those log messages. Based on the relationship, the logs coming from different sources are combined together (details in Section IV-B2).

B. `DisProTrack` Execution Path (Runtime Analyzer)

During the execution of the system, the applications generate multiple logs depending on the user's activities. So the task of the runtime engine is to collect and contextualize the log messages generated at the systems and application levels. We assume that the micro-services containers have `auditd` [42] installed for monitoring system-level logs. This assumption does not violate the *"without instrumentation of the application source code"* constraint, as it is straightforward to add an `auditd` layer to the existing containers without knowing anything about the application source codes.

1) Tracing the execution units from processes belonging to different micro-services (handling Challenge 3): To avoid confusion during the contextualization process of the accumulated logs, we append the Process ID (PID) of the process responsible for generating the log and the timestamp to add the semantic context to individual log entries. For this purpose, we develop a Loadable Kernel Module (LKM) which intercepts all the write syscalls caused by log printing functions to extract the PID information and timestamp of the system and append it to the log preamble. This LKM is deployed in the bare metal infrastructure where the containers are executing (details in Section IV-B2).

2) Dependency explosion and handling confounding root causes (handling Challenge 4): During the execution of the applications, the runtime engine periodically fetches the marked log entries from the micro-services. It generates the UPG after consulting the LMS-CFG. The LMS-CFG provides a relationship between the applications and file, which is exploited to construct UPG as depicted in Fig. 2. The details of the UPG construction procedure are described in the next section. The generated UPG is consulted by the system administrators for the system provenance tracking. The root cause of any suspicious log entry can be identified by backtracing the UPG (details in Section IV-C).

IV. Components of DisProTrack

In this section, we describe the design of individual components of `DisProTrack`. As mentioned previously, `DisProTrack` is subdivided into two parts; (a) *Static Analyzer*, and (b) *Runtime Engine*.

A. Static Analyzer

The static analyzer takes individual micro-service's application binaries as input and constructs the corresponding LMS-CFG for that micro-service application. A typical application will contain a set of statements to print the LMSes with syscalls in between. In our proposed framework, we load the executable application binary and use the python `Angr` module [43], [44] to identify the CFG from it. The generated CFG is a directed graph having the basic instruction blocks (syscalls, printing of LMSes, etc.) as nodes, and the edges of the graph represent jump/call/return statements from one block to another. Let the CFG be represented as $G = (V, E)$, where V and E are the nodes and the edges of the CFG, respectively. We then use the same `Angr` module to extract all the nodes $\mathbb{F}$ from the CFG $G(V, E)$ corresponding to various function calls. One significant issue with the generated CFG is that it does not always provide the complete graph due to the missing hardware-dependent features and system call information. Therefore, in this case, we only concentrate on the LMGF. Typically the stable versions of the applications use standard LMGF (e.g. `printf`, `log4c`, `syslog`, etc.). Let $\mathcal{L}$ be a list of standard LMGF names. We find $\mathbb{L} \subseteq \mathbb{F}$ where $\mathbb{L}$ contains the LMGFs from $\mathcal{L}$. We construct a LMS-CFG $G'(V', E')$ from $G(V, E)$ with the help of $\mathbb{L}$.

We propose Algorithm 1 to convert $G = (V, E)$ to $G' = (V', E')$ for a given $\mathbb{L}$ as the input. Each node in G represents a sequence of instructions. If a node v contains a LMGF name, then the algorithm extracts the arguments of the LMGF, which has been defined as a LMS apriori. For example, for a given LMGF `fprintf`, consider the following log entry function.

"`fprintf(stderr, "AH00526: Syntax error on line %d of %s:");`"

In this case, the algorithm extracts the LMS as `"AH00526: Syntax error on line %d of %s:"`, which is the constant string reference passed as an argument to the LMGF.

During the construction of LMS-CFG, we need to identify the caller functions of the LMGF, which is a time-consuming operation. Therefore, we limit the depth of backward tracing up to a certain threshold (BT), as shown in Algorithm 1:Line 13. The optimal value of BT depends on the complexity of the generated CFG, which we shall discuss during the experimental evaluation (Section V-B1). For accurate identification of the LMSes, we need to avoid the programming language-specific format specifiers. For that, we replace the format specifier of LMS with equivalent regular expressions (see Algorithm 1:Line 10). For example, consider the LMS as shown earlier: "`AH00526: Syntax error on line %d of %s:`". The regular expression equivalent to this LMS becomes "`AH00526: Syntax error on line -?[0-9]+ of .*:`", where `%d` and `%s` are replaced with `[0-9]+` and `.*`, respectively. Additionally, we mark the starting and ending LMS positions of the event handling loops with a flag. This generated and marked LMS-CFG is used by *Runtime Engine* explained next.

Algorithm 1: CFG to LMS-CFG Generation

Procedure `Main`
  Input: $G(V, E)$: CFG, $\mathbb{L}$: Set of LMGFs in $G(V, E)$
  Output: LMS-CFG $G' = (V', E')$
  Initialization:
    $V' \leftarrow \emptyset, E' \leftarrow \emptyset$
  for $v \in \mathbb{L}$ **do**
    /* The Angr tools return whether a node is a loop */
    if v *has a loop* **then**
      /* For a loop, we need to find out all the LMSes ($\mathbb{A}$) that are printed through the loop. */
      $\mathbb{A} \leftarrow$ `BackTraceOptimization`$(BT_{th}, G(V, E), v)$;
      /* Construct the subgraph $G_A(V_A, E_A)$ for A */
      $G_A(V_A, E_A) \leftarrow$ `CreateSubGraph`(A_i);
      $V' \leftarrow V' \cup V_A$;
      $E' \leftarrow E' \cup E_A$;
  for $v' \in V'$ **do**
    Identify format specifiers in v' and replace them with suitable Regular Expressions;
  return $G'(V', E')$;

Function `BackTraceOptimization`
  Input: BT: Backtrace Threshold, $G(V, E)$: CFG, v: A node from $\mathbb{L}$ having a loop
  Output: $\mathbb{A}$: A set of LMSes in the loop
  Initialization:
    $\mathbb{A} \leftarrow \emptyset$
  if v *has a set of LMSes* $\{l_0, \cdots l_k\}$ *printed through the loop with syscalls in between the LMSes* **then**
    /* We consider the loops having a syscall and associated LMSes */
    $\mathbb{A} \leftarrow \mathbb{A} \cup \{l_0, \cdots, l_k\}$;
    return $\mathbb{A}$;
  else
    /* The loop within v does not print any LMSes */
    do
      /* Backtrack to the LMGF that called v, until the backtrack threshold BT is reached. */
      if $\langle \bar{v} \rightarrow v \rangle$ *is a directed edge in* $G(V, E)$ **then**
        if $\bar{v}$ *has a loop with a syscall and a set of LMSes* $\{l_0, \cdots l_k\}$ **then**
          $\mathbb{A} \leftarrow \mathbb{A} \cup \{l_0, \cdots, l_k\}$;
          return $\mathbb{A}$;
      $v \leftarrow \bar{v}$;
      $BT \leftarrow BT - 1$;
    while $BT > 0$;
  return $\mathbb{A}$;

Function `CreateSubGraph`
  Input: $\mathbb{A}$
  Output: $G_A(V_A, E_A)$: A Subgraph generated from $\mathbb{A}$
  Initialization:
    $V_A \leftarrow \emptyset, E_A \leftarrow \emptyset$
  foreach $\{l_i, l_{i+1}\} \in \mathbb{A}$ **do**
    $V_A \leftarrow V_A \cup \{l_i, l_{i+1}\}$;
    $E_A \leftarrow E_A \cup \{\langle l_i \rightarrow l_{i+1} \rangle\}$;
  return $G_A(V_A, E_A)$;

B. Runtime Engine

We design `DisProTrack` Runtime Engine as a micro-service that can be deployed in the serverless platform. Since most of the serverless functions are deployed using multiple containers, log collection and analysis are challenging. `DisProTrack` is concerned about two types of logs; (i) Logs generated by the applications (i.e., inside the container) and (ii) System and/or serverless daemon logs (i.e., logs from the physical servers systems) – we call them Syslogs. The major challenge of obtaining a container's internal logs is that as the process spaces of the containers and the host system are isolated, identification of processes and syscalls become difficult. To avoid such issues, we use an audit daemon in each container by deploying a separate image layer[3] over the containers. The audit daemon is used to track the syscalls generated by the applications inside the containers.

However, audit logs provide the container internal PID, which might conflict with the audit logs obtained from different containers. The conflict must be resolved before the aggregation of the system log and application log. Therefore, we develop a LKM which intercepts LMS before it is printed in the log file and appends a unique tag (which is a combination of container ID, PID, and timestamp) to each LMS entry. This unique tag is used to establish a relationship by adding semantic context among the LMS.

The scope of operation for the runtime engine starts whenever the applications start generating logs. We have segregated Runtime Engine into three submodules; (a) Log accumulator, (b) Log processor, and (c) Provenance builder, which are described as follows.

1) Log Accumulator: Once the log files are generated, the *Log Accumulator* module periodically pulls the log files from all levels and performs operations on them to correlate them with the events. The non-persistent and ephemeral nature of micro-services implies the risk of data loss or the loss of logs generated during the execution phase of the container lifecycle when the container shuts down. Therefore, the proposed module periodically pulls the logs. To prevent data loss due to "`SIGKILL`", we deploy a signal handler inside the deployed image layer to instruct the container to save the logs in a persistent data volume.

2) Log Processor: Once the logs are accumulated, the *Log Processor* module aggregates the logs collected from various sources. However, simple concatenation of log files does not preserve the semantics relationship. Therefore, we convert the text-based log files into an equivalent `JSON` format. From the formatted application log files, the PID of the application is extracted from the tag generated by the LKM. Using the PID, the syscalls are identified from the audit logs. Now the LMS and the syscall-generated logs are merged to create an Application-Specific Common Log (ASCL) file such that the application logs appear just before the corresponding Syslogs.

C. Provenance Builder

At the runtime, the *Provenance Builder* takes the ASCL file and LMS-CFG of a process as input and constructs the corresponding UPG component for that process. The ASCL file contains interleaved application-level logs ($\langle pid, ts, lms \rangle$) and Syslog ($\langle pid, ts, syscall, path, exe \rangle$) entries. Using Algorithm 2, we identify the execution units in the ASCL file with

[3]https://docs.docker.com/storage/storagedriver/ (Accessed: January 13, 2023)

Algorithm 2: UPG Construction

```
1  Procedure Main
       Input: G' = (V', E'): LMS-CFG, F_pid: ASCL file for process pid
       Output: G^U_pid = (V^U_pid, E^U_pid): UPG component for process pid
2      Initialization:
3          V^U_pid ← ∅, E^U_pid ← ∅, U_pid ← ∅, E_pid ← ∅, G_pid ← ∅;
           /* U_pid: An execution unit denoting the
              set of LMSes matched with V'          */
           /* E_pid: Set of system logs corresponding
              to an execution unit                  */
           /* G_pid: Set of all E_pid from F_pid    */
4          end_unit ← false /* tracks execution units */
5      foreach e in F_pid do
           /* e can be an application-level entry
              ⟨pid, ts, lms⟩ or a syslog entry
              ⟨pid, ts, syscall, path, exe⟩         */
           /* pid: Process ID, ts: Log timestamp    */
           /* lms: LMS, syscall: Syscall in log     */
           /* path: System object (dir, socket,
              etc.) accessed by the executable      */
           /* exe: Name of the executable           */
6          if e is an application-level entry then
7              if e is the first entry in F then
                   /* Perform a regex match to find a
                      node n from G'(V', E') which
                      matches with e. n denotes the
                      current state of the search.  */
8                  n ← FindLMSinGraph(G', e);
9                  U_pid ← U_pid ∪ {n}
10             else
                   /* Perform a regex match to find a
                      node in the neighbor of n over
                      the graph G'(V', E'); the new
                      node becomes the current state
                      of the search                 */
11                 n ← MatchNeighbourNodes(G',n,e);
12                 U_pid ← U_pid ∪ n ;
13             if n is a leaf node in G'(V', E') then
14                 end_unit ← true /* Seen a block of
                      LMSes logged by the application
                      from a LMGF                   */
15         if end_unit is true then
16             E_pid ← E_pid ∪ e;
17             end_unit ← false /* tracked syslogs for an
                  execution unit                    */
18             G_pid ← G_pid ∪ E_pid;
19             E_pid ← ∅
20         else
               /* e is a syslog entry and end_unit is
                  false                             */
21             E_pid ← E_pid ∪ e;
       /* Tracked all syslogs for individual
          execution units, now construct the UPG */
22     partition ← 0 /* keep tracks of individual
          execution units                           */
23     foreach E_pid ∈ G_pid do
24         partition ← partition + 1 ;
25         foreach e ∈ E_pid do
26             if e has a valid exe and syscall then
27                 V^U_pid ← V^U_pid ∪ {e.exe} /* the name of
                      the executable application
                      binary becomes a node         */
28                 V^U_pid ← V^U_pid ∪ {partition||e.path} /* the
                      partition value appended with the
                      object path accessed in e
                      becomes the second node       */
29                 E^U_pid ← E^U_pid ∪ {⟨e.exe →
                      partition||e.path, e.syscall⟩} /* an
                      edge is added between the above
                      two nodes with e.syscall as the
                      edge label                    */
```

the help of LMS-CFG, where an execution unit represents a sequence of LMSes execution of a process. In this heuristic, we intelligently apply the LMS-CFG to mark the end of an execution unit by using the leaf nodes of the graph. In this heuristic, we assume that the micro-services are *weakly time-synchronized* with a small time drift λ. The bound on λ depends on how frequently the log messages from two different execution units are getting printed. In reality, λ can be in the order of a few hundred milliseconds as printing the logs too frequently is also an overhead for an application.

Extract Execution Units with the help from LMS-CFGs of application micro-services. Initially, the execution unit is empty. When the algorithm encounters an application-level log entry from the file, which also happens to be the first entry, it performs a regular expression matching on the LMS-CFG to find a node with a match of that entry. If a valid match, say, n, is found, then n becomes the current state. For the rest of the log entries, it performs the regular expression matching with the neighbors of n from the LMS-CFG to find the next state. This step is repeated for all the application-level entries till we find n as a leaf node in the LMS-CFG, which denotes an end unit of the execution unit. The intermediate Syslog entries are added to their respective PID's execution unit E_{pid}. When the end unit becomes true, it implies a block of LMSes has been printed along with a syscall execution. Hence, we save the state of this execution unit in a set G_{pid} and make the execution unit E_{pid} empty for the next set of LMSes to be added. We also set the end unit flag to false.

Interconnect execution units. Once all the syslogs for an execution unit is extracted, Algorithm 2 (Line: 23-29) constructs the UPG for that execution unit G_{pid}. We keep track of individual execution units in G_{pid} using a variable called $partition$. For every execution partition in G_{pid}, if an entry e contains exe and $syscall$, then the nodes of the UPG are – (i) the name of the executable application binary ($e.exe$), and (ii) the system objects such as files, socket, directory, etc. ($e.path$) appended with the $partition$ value. The edges are added between the above two nodes, where the corresponding Syscall denotes the edge labels from the Syslog entry ($e.syscall$). The resulting UPG is the union of all the UPG components constructed for each PID.

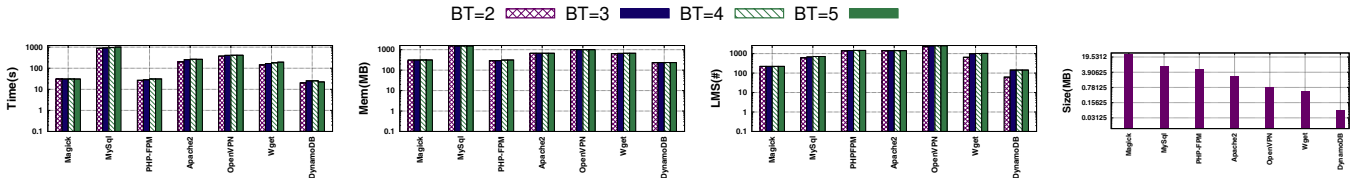
V. Performance Evaluation

The objective of our experimentation is two-fold as follows.

1) Since `DisProTrack` is targeted for serverless applications; resource overhead is a major concern. Therefore, experimentally we want to understand the resource overhead of the framework.
2) We also want to understand how effective `DisProTrack` is for identifying malicious activities.

For experimental analysis, we have implemented `DisProTrack`[4] and executed it in a testbed deployed in our lab. The implementation details are as follows.

[4]https://anonymous.4open.science/r/Project_ALV_2022-CEFD/



(a) Required Time (b) Memory Consumption (c) LMS Found (d) Application Size

Fig. 3: Static Analysis – The Y-axes are in logarithmic scale

A. Experimental Setup

The experiments are executed on a workstation having Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz dual-core processor and 32 GB of memory. To ensure bare metal infrastructural abstraction, we use multiple Virtual Machine (VM) instances running with Ubuntu 22.04 LTS with Kernel version $5.15.0 - 39$-generic. The compute functions are deployed using Docker[5] across the VMs. We use docker-swarm[6] for container management where one container instance is deployed as the manager. For communication, we use an overlay network driver to ensure direct connectivity among the containers. During our experimentation, we have used the standard docker images of the applications collected from `Dockerhub`[7] with an added image layer of `auditd` as described in Section IV-B1. On the other hand, the proposed static analyzer and runtime engine are deployed as a containerized micro-service.

B. Resource Utilization

We conducted experiments to analyze the resource utilization and overhead imposed by `DisProTrack`, which has a two-fold view – (i) the overhead of the static analyzer, which is a one-time event for every deployment scenario, and (ii) the overhead of the runtime engine, which is a periodic event. Therefore, we present different sets of experiments to understand these overheads as follows. Each experiment is repeated 10 times, and the mean is shown in the plots.

1) Overhead of Static analyzer: To understand the resource utilization overhead during static analysis, we have considered 7 different applications as given in Table I.

TABLE I: List of Benchmarked Applications

Name	Type of Application
`Apache Webserver`	Together popular as LAMP-Stack and widely used for web service deployment
`PHP-FPM`	
`MySql`	
`DynamoDB`	A noSQL based database used in Amazon Lambda stream processing usecase
`ImageMagick`	An image processing framework can be used for various Amazon Lambda file processing usecases
`OpenVPN`	A popular Secure IP tunnel daemon
`Wget`	Linux network downloader

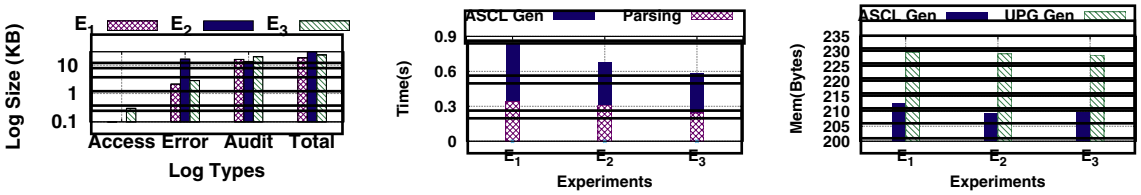
Based on the obtained results (Figs. 3a and 3b), we observe that the static analysis for `MySql` takes the longest time to complete and needs a greater amount of memory. The time and memory requirements increase when the BT increases. This is justified as the value of BT increases, the number of backtraces also increases (as mentioned in Section IV-A). However, an increase in the number of backtraces can identify more number of LMSes as shown in Fig. 3c. We also observe that even though the size of `Magick` is greater than the size of `MySql` (Fig. 3d), it takes more time and memory for `MySql` to trace the LMSes. This is due to the nature of the program control instructions used by developers while developing the application. Hence, it takes more time and memory to complete the backtrace in the presence of higher number of indirect branch instructions. For the same reason, `PHP-FPM` takes less amount of time and memory than `MySql`; however, it identifies more number of LMSes.

2) Overhead of Runtime Engine: Since the runtime engine works for a pipeline of micro-services, we execute multiple experiments on a web service application. Our deployed LAMP stack web service is composed of 3 micro-services (similar to Fig. 2); (a) [μ_1]:`Apache` based web server, (b) [μ_2]: `PHP-FHM` for server-side request handling, and (c) [μ_3]: `MySQL` for handling queries generated by the `PHP-FHM`. Each HTTP request incident on μ_1 forwards a `FastCGI` requests to μ_2. μ_2 executes the handler functions and accesses the database deployed in μ_3 for dynamic content. Once the page is constructed, μ_1 returns it as a response to the client. Specifically, we have hosted an application authorization portal where μ_1 interacts with the μ_2 via μ_3 to validate the user credentials. Upon receiving the valid credentials, μ_1 redirects from the login page to a user-specific home page. Otherwise, it remains on the same page with an error message pop-up. On top of the authorization service, we consider three experimental login scenarios as follows.

- E_1 New users register themselves, and the details are updated in the database. Once registration is successful, the user tries to log in and then log out of the application with valid credentials.
- E_2 A user logs in with a valid credential and goes to the home page. Once logged in, they try to reset the password and re-login with a new password.
- E_3 A user tries to log in to the deployed web service. On the homepage, they click on a link that triggers a background script and redirects the user to a different IP.

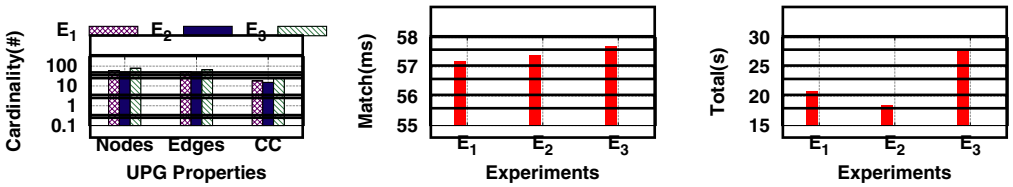
The size and types of log files generated during these experiments are presented in Fig. 4a. The results show that the error logs generated during E_2 are more significant than E_3. In contrary, the access log generated by E_3 is much greater than

[5]https://www.docker.com/ (Accessed: January 13, 2023)
[6]https://docs.docker.com/engine/swarm/ (Accessed: January 13, 2023)
[7]https://hub.docker.com/ (Accessed: January 13, 2023)



(a) Log File Size (b) Phase wise execution time (c) Phase wise Memory utilization

Fig. 4: Runtime Analysis – The Y-axis in (a) is in logarithmic scale



(a) UPG Parameters (b) RegEx Identification Time (c) UPG Construction time

Fig. 5: Runtime Analysis – The Y-axis in (a) is in logarithmic scale

the rest of the two scenarios. However, the audit log generated by E_3 is greater than the other two. The log file size depends on a user's actions while browsing the web service.

Next, we present the phase-wise time consumption analysis in Fig. 4b. Here we observe that the time required to parse the log files, merging them to create an ASCL and generation of UPG during provenance builder (shown in Fig. 5c) is directly related to the total amount of logs generated during the experiments which take approximately 600ms-900ms to complete the runtime processing. After contextualizing the log files, the system administrator provided suspicious log messages converted into an equivalent regular expression (RegEx) to avoid the values of the variables. This generated RegEx string is searched across the LMS-CFG which takes only a few milliseconds (as shown in Fig. 5b). We also provide the memory consumption during the individual phases as presented in Fig. 4c which suggests that the memory footprint is significantly lower for the modules of the runtime engine (in between 200B to 250B).

We observe that depending on the scenarios, the nature of the UPG is different, as shown in Fig. 5a. We find that E_3 has a significantly higher amount of nodes and edges than the rest of the two cases. The number of connected components in the UPG is considerably higher for E_3. Each connected component in the graph represents an execution unit of a process, which helps resolve the dependency explosion problem. Only the related events of a process form a connected component. More number of connected components implies that the graph is more densely partitioned.

VI. Analysis

This section discusses the effectiveness analysis of `DisProTrack`. In this context, we consider an adversarial scenario. The static analyzer can easily detect an adversary who can modify the application's source code. However, an adversary accessing the runtime platform can evade the static analysis and may issue malicious operations during code execution. Therefore, in this section, we primarily focus on detecting runtime adversaries. We have simulated multiple attack scenarios by considering different adversary models. We next discuss one of them.

A. Adversarial model & Attack Scenario

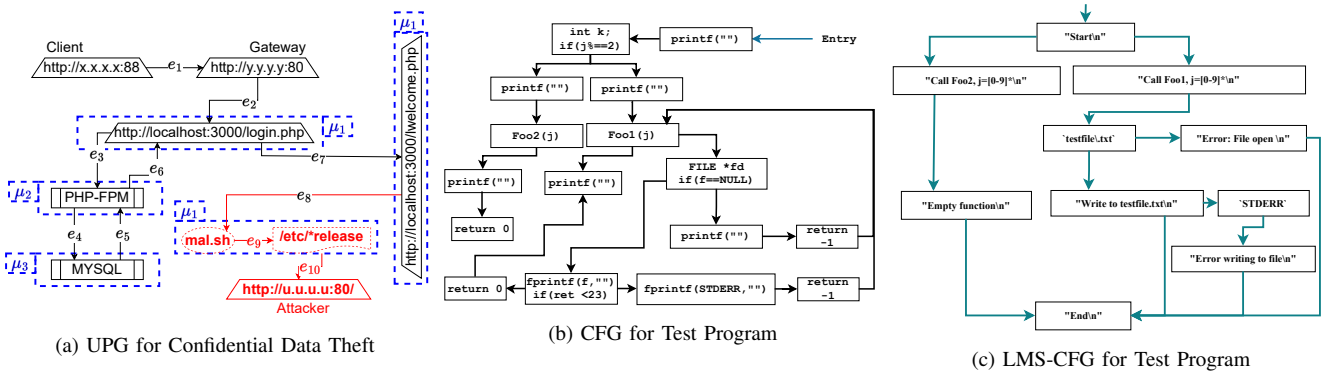
We assume that an adversary can somehow bypass the authorization mechanisms and gain access to one/more application container(s) except the runtime engine container without being detected. In the compromised container(s), the adversary can add, modify, and/or execute scripts and deploy webpages. However, We assume that the logs and audit rules are part of the trusted computing base (TCB). Moreover, the communication between the compromised container and the runtime engine can not be adulterated.

Using this adversarial model, the attacker may perform different types of malicious activities. However, here we present the case study in light of *"confidential data theft"* attack[8] where the attacker attempts to insert a malicious script to steal the confidential information from the compromised system. This script can be triggered during execution time. We simulated the attack on top of our web service application described in Section V-B2 by placing a malicious script named *"mal.sh"* in `PHP-FPM` container. This script is executed when an authorized entity login to the website after successful authentication and clicks on a masked link on the welcome webpage. Once the script gets triggered in the background, alongside normal execution, it tries to access and read a sensitive file on the server and forward them to an attacker's server IP address.

B. Provenance Builder for Attack Detection

Let us understand how this attack can be detected using `DisProTrack`. The UPG constructed by `DisProTrack` for above attack scenario is presented in Fig. 6a. To ensure

[8]https://bit.ly/trendmicro_shading_light (Accessed: January 13, 2023)



(a) UPG for Confidential Data Theft

(b) CFG for Test Program

(c) LMS-CFG for Test Program

Fig. 6: A PoC Case Study to Analyze the Accuracy of DisProTrack

readability, we have omitted a few UPG metadata and masked IP addresses. Using the relative event sequence numbers/edge identifier, the sequence of events can be traced in order. From the particular UPG instance, we can find that a client with IP address $x.x.x.x$ is connected to the service via port 88. The client takes the normal authentication route (from e_1 to e_7) to reach the `welcome.php` page. After that, the `mal.sh` script is triggered which results in collection of data from `/etc/*release` directory and forward it to $u.u.u.u$ (from e_8 to e_{10}). This step is a deviation from the standard behavior; therefore, the system administrator must intervene in this case and take some preventive action (e.g., suspend user access, block IP, etc.).

```c
#include<unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#define Foo2(int j) ({printf("Empty function\n");})

int Foo1(int j){
    FILE *f = fopen("testfile.txt", "a");
    if (f == NULL) {
        printf("Error: File open\n");return -1;
    }
    if (fprintf(f, "Write to testfile.txt\n") < 23) {
        fprintf(stderr, "Error writing to file\n");
        return -1;
    }
    return(0);
}
/***********************/
int main(void){
    printf("Start\n");int j = rand();
    if(j%2==0){
      printf("Call Foo1(), j = %d\n", j);Foo1(j);
    }else{
      printf("Call Foo2(), j = %d\n", j);Foo2(j);
    }
    printf("End\n");
    return 0;
}
```

Listing 1: PoC Program for Accuracy Analysis

C. Accuracy analysis of DisProTrack

During our development of DisProTrack and experimentation with several other attack scenarios, we observed that the detection of attack depends on the accuracy of LMS-CFG construction from CFG in the static analyzer. Although we have presented the LMS identification results in Fig. 3c for different applications, the lack of gold standard values restricts us from claiming the accuracy of the proposed Algorithm 1. Therefore, to justify the accuracy of the static analyzer, we present a small sample proof-of-concept (PoC) program (Listing 1) here. The sample program presents two function calls depending on a random number generated. The functions can either generate a log message in the STDERR console or in a log file. As the program is simplistic in nature, it is easy to ascertain the accuracy of the generated LMS-CFG from the framework presented in Fig. 6c. For comparison purposes, we present the corresponding CFG in Fig. 6b, which is also obtained from the static analyzer module. From these two figures, we observe that both the figures have 8 listed LMSes and two file handles. The causal paths among the nodes are also verified to ensure that all the paths are covered in the corresponding LMS-CFG.

VII. Conclusion

This paper developed a non-invasive causality analysis framework, called DisProTrack, for provenance tracking over distributed serverless applications. The proposed framework is capable of adversarial attack analysis by identifying the root causes effectively. DisProTrack can be deployed on top of the SLC as a micro-service and has the virtue of being lightweight and provides results within 0.5 minutes. A PoC analysis of DisProTrack also shows its efficiency and efficacy in detecting attack instances for an SLC application.

A critical aspect of DisProTrack is that it uses a heuristic to identify the execution units by matching the CFGs generated from the micro-service binaries with the runtime application and system logs based on their temporal execution patterns. Thus, the framework might wrongly identify the execution units if the underlying servers running the micro-services are not weakly time synchronized (time drift within a small threshold). Nevertheless, this condition rarely occurs in a typical distributed SLC platform with multiple micro-services interacting with each other. Further, DisProTrack can be deployed as an additional micro-service along with the other application micro-services running over the servers, making it robust to be applied for a wide range of production-grade serverless application scenarios.

The authors have provided public access to their code and/or data at https://github.com/usatpath01/DisProTrack.

REFERENCES

[1] K. Kuusinen, V. Balakumar, S. C. Jepsen, S. H. Larsen, T. A. Lemqvist, A. Muric, A. Nielsen, and O. Vestergaard, "A large agile organization on its journey towards devops," in *Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2018)*, 2018, pp. 60–63.

[2] K. Ojo-Gonzalez, R. Prosper-Heredia, L. Dominguez-Quintero, and M. Vargas-Lombardo, "A model devops framework for saas in the cloud," in *Advances and Applications in Computer Science, Electronics and Industrial Engineering*, 2021, pp. 37–51.

[3] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, "S-faas: Trustworthy and accountable function-as-a-service using intel sgx," in *ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019, pp. 185–199.

[4] K. S.-P. Chang and S. J. Fink, "Visualizing serverless cloud application logs for program understanding," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2017)*, 2017, pp. 261–265.

[5] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Communications of the ACM*, pp. 76–84, 2021.

[6] D. Taibi, J. Spillner, and K. Wawruch, "Serverless computing-where are we now, and where are we heading?" *IEEE Software*, pp. 25–31, 2020.

[7] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, "Exploring layered container structure for cost efficient microservice deployment," in *IEEE Conference on Computer Communications (IEEE INFOCOM 2021)*, 2021, pp. 1–9.

[8] "Aws x-ray," https://aws.amazon.com/xray/.

[9] "Google cloud - viewing monitored metrics," https://cloud.google.com/functions/docs/monitoring/metrics/.

[10] "Microsoft azure monitor," https://cloud.google.com/functions/docs/monitoring/metrics/.

[11] "Iopipe - monitor serverless applications," https://www.iopipe.com/.

[12] "Dashbird - monitor serverless applications," https://dashbird.io/.

[13] "Thundra - monitor serverless applications," https://www.thundra.io/.

[14] "Epsagon - monitor serverless applications," https://epsagon.com/.

[15] S. Zawoad, R. Hasan, and K. Islam, "Secprov: Trustworthy and efficient provenance management in the cloud," in *IEEE Conference on Computer Communications (IEEE INFOCOM 2018)*, 2018, pp. 1241–1249.

[16] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis," in *The Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

[17] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *IEEE Symposium on Security and Privacy (SP 2020)*. IEEE, 2020, pp. 1172–1189.

[18] M. M. Anjum, S. Iqbal, and B. Hamelin, "ANUBIS: a provenance graph-based framework for advanced persistent threat detection," in *ACM/SIGAPP Symposium on Applied Computing*, 2022, pp. 1684–1693.

[19] H. Irshad, G. Ciocarlie, A. Gehani, V. Yegneswaran, K. H. Lee, J. Patel, S. Jha, Y. Kwon, D. Xu, and X. Zhang, "Trace: Enterprise-wide provenance tracking for real-time apt detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4363–4376, 2021.

[20] L. Yu, S. Ma, Z. Zhang, G. Tao, X. Zhang, D. Xu, V. E. Urias, H. W. Lin, G. F. Ciocarlie, V. Yegneswaran *et al.*, "ALchemist: Fusing application and audit logs for precise attack provenance without instrumentation." in *The Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[21] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy {Whole-System} provenance for the linux kernel," in *USENIX Security Symposium (USENIX Security 2015)*, 2015, pp. 319–334.

[22] J. Tavori and H. Levy, "Tornadoes in the cloud: Worst-case attacks on distributed resources systems," in *IEEE Conference on Computer Communications (IEEE INFOCOM 2021)*, 2021, pp. 1–10.

[23] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. A. Gunter *et al.*, "You are what you do: Hunting stealthy malware via data provenance analysis." in *The Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

[24] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "ALASTOR: Reconstructing the provenance of serverless intrusions," in *USENIX Security Symposium (USENIX Security 2022)*, 2022.

[25] "Linux fuse," https://man7.org/linux/man-pages/man4/fuse.4.html.

[26] C. Schaufler, "Lsm: Stacking for major security modules," https://lwn.net/Articles/697259, 2016.

[27] T. F. J.-M. Pasquier, J. Singh, D. Eyers, and J. Bacon, "Camflow: Managed data-sharing for cloud services," *IEEE Transactions on Cloud Computing*, vol. 5, no. 3, pp. 472–484, 2017.

[28] A. Gehani and D. Tariq, "Spade: Support for prmiscovenance auditing in distributed environments," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012, pp. 101–120.

[29] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for windows," in *Annual Computer Security Applications Conference (ACSAC 2015)*, 2015, pp. 401–410.

[30] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatakrishnan, "SLEUTH: Real-time attack scenario reconstruction from COTS audit data," in *USENIX Security Symposium (USENIX Security 2017)*, 2017, pp. 487–504.

[31] Y. Xie, D. Feng, Y. Hu, Y. Li, S. Sample, and D. Long, "Pagoda: A hybrid approach to enable efficient real-time provenance based intrusion detection in big data environments," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 6, pp. 1283–1296, 2018.

[32] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *ACM SIGSAC Conference on Computer and Communications Security*, 2019, p. 1795–1812.

[33] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: real-time apt detection through correlation of suspicious information flows," in *IEEE Symposium on Security and Privacy (SP 2019)*. IEEE, 2019, pp. 1137–1152.

[34] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao, "Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics," in *The Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[35] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple perspective attack investigation with semantic aware execution partitioning," in *USENIX Security Symposium (USENIX Security 2017)*, 2017, pp. 1111–1128.

[36] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition." in *The Network and Distributed System Security Symposium (NDSS 2013)*, vol. 2, 2013, p. 4.

[37] ——, "Loggc: Garbage collecting audit log," in *ACM conference on Computer & communications security (SIGSAC 2013)*, 2013, pp. 1005–1016.

[38] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting." in *The Network and Distributed System Security Symposium (NDSS 2016)*, vol. 2, 2016, p. 4.

[39] R. Yang, S. Ma, H. Xu, X. Zhang, and Y. Chen, "UIScope: Accurate, instrumentation-free, and visible attack investigation for gui applications." in *The Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

[40] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker, "X-Trace: A pervasive network tracing framework," in *USENIX Symposium on Networked Systems Design & Implementation (NSDI 2007)*, 2007.

[41] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Symposium on Operating Systems Principles (SOSP 2015)*, 2015, pp. 378–393.

[42] (2007, Mar) auditd(8) - linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man8/auditd.8.html

[43] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (state of) the art of war: Offensive techniques in binary analysis," *IEEE Symposium on Security and Privacy (SP 2016)*, 2016.

[44] "Angr," https://angr.io/.