Malware Detection for Mobile Devices Using Software-Defined Networking

Ruofan Jin, Bing Wang
Department of Computer Science and Engineering
University of Connecticut, Storrs, Connecticut 06269

***Abstract*—The rapid adoption of mobile devices comes with the growing prevalence of mobile malware. Mobile malware poses serious threats to personal information and creates challenges in securing network. Traditional network services provide connectivity but do not have any direct mechanism for security protection. The emergence of Software-Defined Networking (SDN) provides a unique opportunity to achieve network security in a more efficient and flexible manner. In this paper, we analyze the behaviors of mobile malware, propose several mobile malware detection algorithms, and design and implement a malware detection system using SDN. Our system detects mobile malware by identifying suspicious network activities through real-time traffic analysis, which only requires connection establishment packets. Specifically, our detection algorithms are implemented as modules inside the OpenFlow controller, and the security rules can be imposed in real time. We have tested our system prototype using both a local testbed and GENI infrastructure. Test results confirm the feasibility of our approach. In addition, the stress testing results show that even unoptimized implementations of our algorithms do not affect the performance of the OpenFlow controller significantly.**

I. Introduction

The mobile era is underway. People are more dependent than ever on their mobile devices in their daily lives and work. It is predicted that mobile devices will surpass PCs as the most common type of Internet access device in the world by 2013 [1]. The growing popularity of mobile devices has made them a more attractive target for attackers. The first malicious software aimed specifically at smartphones hit in 2004 [2]. Recently, one of the most famous malware on Android, DroidDream, infected more than 260,000 devices before Google was able to remove the related malicious apps from its official market [3]. There were also reported cases that malicious applications had passed the strict review process of Apple and user information was stolen [4]. Experts have warned the dangers of mobile malware for years, and alerted that security research community should pay more attention to mobile malware [5], [6].

Several factors make mobile malware an even greater threat. First, people are placing growing reliance and valuable information on mobile devices, which are enticing to attackers. Second, mobile devices are more likely to connect to insecure networks where malicious hosts may exist. Additionally, people update/patch the software/firmware of mobile devices less frequently, which makes zero-day vulnerability exploit possible [7]. To make matters worse, giant app stores (e.g., Google's Play store, Apple's App Store, etc.), while enabling apps to be distributed at an unprecedented speed, also provide attackers opportunities to exploit a larger number of devices. Last, many users jailbreak iOS devices or root Android devices in order to install software from third parties, which usually involves exploiting vulnerabilities in the operating systems to gain root access and thus makes the devices more vulnerable to malware [8], [9].

The growth of mobile malware and its security threats have created challenges in securing networks. Traditional network services provide connectivity but do not have any direct mechanism for security protection. Most network security mechanisms are provided as add-on features, such as web authentication, firewalls, and intrusion detection tools, etc. These features, however, may lack flexibility and responsiveness. For instance, if an infected host needs to be quarantined, the administrator has to install firewalls and update firewall rules, which involves protocols at multiple layers and is slow to respond. The emergence of Software-Defined Networking (SDN) provides a unique opportunity to protect networks in a more efficient and flexible manner. In SDN, as the control and data planes are decoupled [10], the network can be managed in a logically centralized way and the traffic control rules can be imposed in real time. For instance, when a host is found misbehaving, the controller can instantly update the control rules of access points (or switches), so that the host will be disconnected immediately.

In this paper, we present a mobile malware detection system using the SDN architecture. Our system can detect mobile malware by identifying suspicious network activities through traffic monitoring. It can further collaborate with OpenFlow controllers to impose security rules directly at the network layer by changing how the switches (access points) handle the incoming traffic. Our main contributions are as follows.

- To the best of our knowledge, this is the first effort that uses the SDN architecture to tackle malware on mobile devices.
- Inspired by existing malware detection methods, we design several light-weight algorithms that are applicable to SDN and require only connection establishment packets.
- We implement a proof-of-concept mobile malware detection system based on OpenFlow protocol, and test it using both a local testbed and GENI infrastructure. The test results confirm the feasibility of our approach. Furthermore, we have stress tested our system through extensive simulation, and the test results demonstrate that even unoptimized implementations of our algorithms do not lead to significant slowdown of the

OpenFlow controller.

The rest of the paper is organized as follows. Section II describes the SDN architecture and malicious behaviors of mobile malware. Section III presents our mobile malware detection system architecture and detection algorithms. Section IV presents the system implementation and performance. Section V describes related work. Finally, Section VI concludes the paper and presents future directions.

II. Preliminaries

In this section, we describe the architecture of SDN and major malicious behaviors of mobile malware.

A. SDN and OpenFlow Network

Software-Defined Networking (SDN) has emerged as a new paradigm for enabling innovation in networking research and development. In the SDN architecture, the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications [10]. OpenFlow protocol [11] is one element of the SDN architecture, which allows switches to perform flows-level control[1]. An OpenFlow switch usually contains at least one *flow table* that is managed by an external *controller*. The controller connects to the switch through a secured OpenFlow channel and manages the switch via the OpenFlow protocol. Using this protocol, the controller can add, update, and delete flow entries, both reactively in response to packets and proactively with pre-defined rules. The flow table inside the switch consists of a set of *flow entries*. Each flow entry contains match fields (to match against packets), counters (that are updated for matching packets), as well as a set of instructions to be applied to matching packets. The actions for a matching packet range from dropping the packet, outputting the packet on specified port(s), and modifying header fields.

TABLE I. Selected match fields in flow entry (applicable to TCP packets)

Field Name	Note
Ingress Port	Physical or switch-defined virtual port
Ethernet source	MAC address of source host
Ethernet destination	MAC address of destination host
IP source address	IP address of the source host (can use subnet mask or bitmask)
IP destination address	IP address of the destination host (can use subnet mask or bitmask)
IPv4 protocol	E.g., TCP, UDP, ICMP, etc.
Source port	TCP/UDP source port
Destination port	TCP/UDP destination port

Table I shows several selected match fields that an incoming packet is compared against. Each field can either contain a specific value, or be wild-carded (which matches any value). It thus allows flexibility in specifying the exact group of packets on which a certain set of instructions is to be executed. For instance, if we want to set up a rule of forwarding all HTTP traffic to a server connected to a certain port of the switch, we can simply set up a flow entry which has all match fields wild-carded except 'IPv4 protocol' being 'TCP' and 'Destination port' being '80'.

[1]The term flow can refer to a single data flow connection between two hosts, defined uniquely by its five-tuple (source IP address, source port, destination IP address, destination port, protocol type TCP/UDP).

Whenever a packet comes to an OpenFlow switch, it will be compared against the flow tables in the switch. If a matching flow entry is found, the actions associated with the flow entry will be executed. Otherwise, the packet will be forwarded to the controller, which then decides how to deal with the packet and installs flow entries to the switch if needed.

B. Malware Behavior

Malicious behaviors of malware on both PC and mobile platforms have been analyzed in many existing studies. Here we summarize the major malicious behaviors that can be potentially identified through traffic analysis.

1) Repackaging and Updating Attacks: One major way for a malware program to infect mobile devices is to disguise itself as a legitimate application through repackaging (i.e., the malicious payload is piggybacked into popular applications). It happens more often in third-party app stores. Sometimes, the legitimate software and the malware look the same, and thus users may get malware installed inadvertently. The malware payload can be exposed using static analysis tools. To avoid this, some recent malware does not include the malicious payload in its installation package. Rather, it only encapsulates a component that is able to fetch the malicious payload after being installed on the devices. Whenever an Internet connection is available, that component will download and install the actual malicious payload by stealth.

2) Drive-by Download Attacks: Another technique adopted by malicious software is the 'drive-by download' attack. Users may inadvertently download malware by visiting a compromised website, viewing a malicious e-mail or clicking a misleading link. This kind of attack is even more likely to succeed on mobile devices that have limited screen size. This is because many mobile browsers hide the address bar to display more page content, which removes the visual cues for users to confirm the locations they are visiting.

3) Remote Control: A large portion of malware turns the infected phones into bots for remote control [12], and most of them use HTTP-based web traffic to receive bot command from the attack's server. While most of those servers are registered in domains controlled by attackers themselves, a recent study [12] shows that the servers can be hosted in public cloud, such as Amazon cloud, which makes malware detection even more difficult.

4) Information Collection: Malware also collects valuable personal information stored on mobile devices, including messages, emails, phone numbers, and account information. The information collected will be sent to the attacker's server secretly. Some of the malware can circumvent static analysis by encrypting the URLs of their remote servers.

From the above analysis, we observe that most of the malicious behaviors are directly or indirectly related to network activities. Therefore, we believe a network-level protection is necessary in case static-analysis based examination techniques fail.

III. Mobile Malware Detection Using SDN

In this section, we first describe the architecture of our mobile malware detection system, and then present the algorithms we adopt for malware detection based on OpenFlow protocol.

A. System architecture

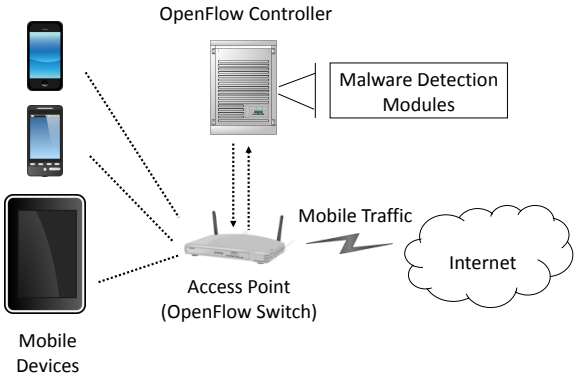

Fig. 1. Architecture of malware detection system.

Motivated by the flexibility of the SDN architecture and the observation that most mobile malware requires Internet connections (see Section II-B), we design a system that detects mobile malware through real-time traffic analysis using the SDN architecture. Fig. 1 illustrates the high level architecture of our system. For ease of exposition, we assume only one wireless access point to which all the mobile devices are connected. We assume that the mobile traffic can be filtered out using the methods described in [13]. In this architecture, the access point, which is essentially an OpenFlow-enabled switch, is controlled by an OpenFlow controller. The access point forwards mobile traffic to the OpenFlow controller, and receives and installs flow entries from the controller through a secured channel. The malware detection system is a module inside the OpenFlow controller, which can extract the traffic information we are interested in. Based on the traffic information, our malware detection system detects malware in real time. The analysis results from the malware detection module will instruct the controller to regulate the traffic by setting up flow entries to the switch. In this way, the control of the network is logically centralized and the security rules can be imposed in real time right from the network layer.

Note that the entire process is transparent to mobile device users and users do not need to install any special hardware or software, which makes our system readily deployable.

B. Detection Algorithms

In traditional networks, realizing a real-time traffic inspection scheme requires additional devices and complicated collaboration. Under SDN architecture, we can design detection algorithms that are readily integrated with the OpenFlow controller, which enables controlling the traffic in a logically centralized way. The algorithms we adopted are described as follows.

1) IP Blacklist: A straightforward way to protect a network is maintaining a blacklist of malicious IP addresses which can either be obtained from public available sources (e.g., [14], [15]) or from historic data, and denying immediately any network flow that involves an IP address in that blacklist. In OpenFlow protocol, such a blacklist based protection mechanism can be readily implemented as follows. When a new connection is to be established, since any connection request packet that does not match any flow entry in the OpenFlow switch will be forwarded to the OpenFlow controller, the controller can readily obtain the destination IP address of the connection. If the IP address is in the blacklist, the connection will be blocked. Otherwise, the connection request packet will be forwarded to the destination host.

2) Connection Success Ratio: Inspired by [16], which leverages the observation that the probability that a connection attempt is successful should be much higher for a benign host than a malicious host, we design a malware detection algorithm based on connection success ratio. The algorithm aims at detecting scanning worms that try to search for compromised devices in the network and try to infect them.

To implement this algorithm using OpenFlow protocol, the controller maintains a list of pending connections which have yet to receive a response for each host. The more such pending connections originated from a host, the higher the likelihood that the host has been infected. Once the response to a connection request is received, the corresponding pending connection will be removed from the list. And then our algorithm installs the two-way flow entries to handle the subsequent packets in that flow. If the number of pending connections for a host exceeds a pre-defined threshold T_1, the controller will consider the host as infected, raise an alarm and disconnect the device from network by dropping all the packets from the host.

Specifically, when a host x sends a connection request to a new external destination y, the following steps will be executed:

(a) If no matching flow entry (for flow from x to y) can be found at the access point, the packet will be forwarded to the controller.

(b) The controller simply forwards the packet to destination y directly through the access point, without setting any flow entry. Meanwhile, it adds the connection request to x's list of pending connections.

(c) The controller checks the number of x's pending connections. If it is larger than T_1, the controller raises an alarm and disconnects the host.

(d) Once a reply packet from y is received, the packet will be forwarded to the controller (as there is no matching flow entry yet). Then the controller sets up two-way flow entries in the access point, for traffic from x to y and traffic from y to x, respectively. Meantime, the controller removes the connection request from x's list of pending connections.

3) Throttling Connection: The study of [17] observed that during virus propagation, an infected machine will try to connect to as many different machines as fast as possible, while an uninfected machine behaves differently: connections are made at a lower rate, and are locally correlated (since repeated connections to recently accessed machines are likely). On mobile platforms, we expect to see a similar trend. To limit the rate of connections to new hosts and identify the infected clients, we implement an algorithm that maintains a list of *Recently Accessed Hosts* (RAH) for each host and check the list whenever a new connection request is received. Specifically, if the destination is in the RAH list, then the

connection will be allowed. Otherwise, the connection request will be placed into a *waiting queue* and the requests in the queue will be popped out at a rate of R requests per second. If the length of the waiting queue of a host becomes larger than a pre-defined threshold T_2, that host will be considered suspicious and be disconnected. Specifically, when a host x sends a connection request to an external destination y, the following steps will be executed:

(a) If the destination y has been accessed before (i.e., it is in the RAH list), then the controller sets up the two-way flow entries directly.

(b) If the destination y is a new host (i.e., not in the RAH list), the controller places the connection request at the end of the queue without setting up any flow entry. If the size of the waiting queue becomes larger than the threshold T_2, the controller raises an alarm and disconnects the host from the network.

(c) At the rate of R requests per second, the controller pops out the connection requests from the waiting queue and adds the destination host to the RAH list.

(d) When a reply packet from destination y is received, the controller sets up the two-way flow entries in the access point.

4) Aggregation Analysis: Malware rarely infects only a single victim. Therefore, we expect that when one host is infected by malware, multiple other hosts may be infected as well, especially in a large scale network. In addition, we expect that the infected client share behavioral characteristics in their network activities that are distinct from those of benign clients. Inspired by [18], we design an algorithm that detects the infected hosts by identifying aggregates of "similar" communications. Specifically, given a collection of bi-directional flow records observed, we collect the flows with the following features.

- Common destination: Since most of the malware needs to communicate with the control server, either receiving command, downloading malicious payload or uploading information, we expect to see the flows to have common destination hosts. Intuitively, it is especially suspicious when the URL or the IP address of the destination host does not belong to a well-known site.
- Common connection time: The duration of each flow will be recorded. The rationale is that the same malware usually exhibits similar duration of connection.
- Common platform: Most mobile malware programs are platform specific (i.e., cross-platform malware is rare, if any). Therefore, we also identify the flows that originate from the same platform.

Alone, each of the characteristics does not provide much information since normal clients can also exhibit similar features. But in combination, we can possibly extract aggregates of malware communications as claimed in [18].

5) Discussion: Remark that all the above four algorithms have the following merits. First, they only require the packets that are related to connection establishments, so user privacy is protected. In addition, since once the connection is established and then later packets can go through the switch directly, it does not incur large overhead. Secondly, the above algorithms are independent from each other, so we have the flexibility to use them selectively.

IV. System Evaluation

In this section, we first present the system implementation of our proof-of-concept mobile malware detection system, and then evaluate the performance of our system on a local testbed. In the end, we present our experiments using the Global Environment for Network Innovation (GENI) [19] infrastructure.

A. System Implementation

To validate our scheme, we implemented a prototype of our mobile malware detection system. In particular, for the OpenFlow controller, we choose an Apache-licensed, Java-based OpenFlow controller, FloodLight [20], whose modular design enables us to embed our detection algorithms as independent modules. For the OpenFlow switch, we choose a Netgear WNDR3800 wireless router as the access point for mobile devices, along with the Pantou project [21] with OpenWrt [22] backfire distribution to enable OpenFlow. Further, we use four smartphones, two HTC wildfire with Android and two Apple iPhone4S with iOS, as the mobile clients. For ease of exposition, we label the four clients as A, B, C and D. The topology of our experiment is shown in Fig. 2.

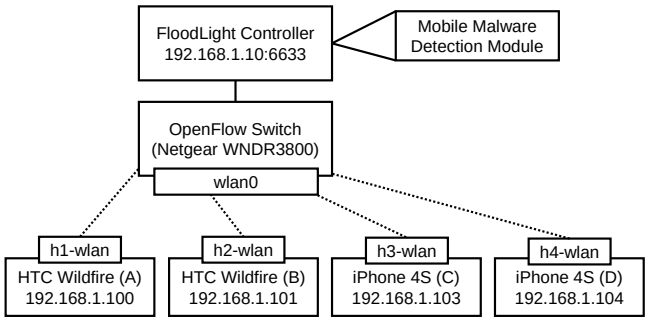

Fig. 2. Topology of small-scale experiments.

Specifically, the Floodlight controller connects to the OpenFlow switch via a secured channel. Our four mobile detection algorithms are implemented as independent modules inside the Floodlight controller. And the four mobile devices are connected to the access point through Wi-Fi.

B. Functionality Verification

To verify the functionalities of the four proposed detection algorithms, we run scripts on the smartphones that mimic malware behaviors[2]. The four algorithms are tested as follows.

- IP Blacklist. To test the blacklist functionality, we let client A try to connect to a set of IP addresses that are on the blacklist (obtained from [15]). We observed

[2]This is more controllable than running the real malware, and does not affect the purpose of showing the feasibility of our approach.

that the client was disconnected from the network immediately, and the OpenFlow controller raised an alarm.

- Connection successful ratio. For this algorithm, we set the parameter T_1 (detection threshold, as defined in Section III-B2) to be 20. Then we let client B connect to a set of non-existing IP addresses sequentially. We observed that when the number of unsuccessful connections exceeded T_1, the client was disconnected immediately, and the controller raised an alarm.
- Connection throttling. For this algorithm, we set the parameters T_2 (maximum size of queue) to be 20 and R (connection rate) to be 1 request per second (see Section III-B3). Then we let client C initiate connections to a list of 30 valid IP addresses. We observed that 20 of the connection requests were allowed at the expected rate (one request per second). Once the twenty-first connection was initiated, the controller raised an alarm.
- Traffic aggregation. For this algorithm, we let all the four clients connect to 30 IP addresses that are randomly selected from a list of 100 IP addresses. Our detection module was able to identify the common destinations of the four clients. Based on the analysis results, the network administer can identify suspicious network activity more conveniently.

The above experiments confirm the feasibility of our malware detection algorithms. We should note that our system is still at the proof-of-concept stage and the parameters of the algorithms can be further optimized based on real traffic characteristics.

C. Efficiency Evaluation

A natural question is whether our detection algorithms have adverse effects on the performance of the network. To answer this question, we benchmark the performance of the Floodlight controller with and without our detection algorithms enabled using Cbench [23].

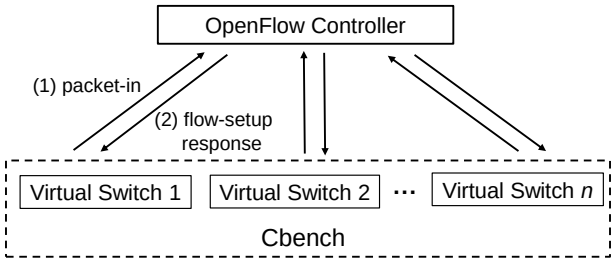

Fig. 3. Benchmark controller performance using Cbench.

Cbench is a tool for testing an OpenFlow controller. As illustrated in Fig. 3, it simulates a set of switches that are connected to the controller being tested. Each switch generates 'packet-in' messages and watches for flow-setup responses from the controller. Every packet-in message will be examined by the controller (and our detection algorithms as well). We measure the performance of the controller with the following metrics:

- Latency. It measures the packet processing delay of the controller. To measure latency, Cbench allows only one packet on the fly, and measures how fast a response is received back from the controller. This metric reflects the performance of the controller under light traffic conditions.
- Throughput. It measures the maximum packet processing rate of the controller. To measure throughput, Cbench sends as many packet-ins as possible to the controller, and counts how many responses per second are received back from the controller. This metric indicates the performance of the controller under heavy traffic conditions.

Furthermore, when measuring the performance, we let all the traffic generated by Cbench be benign (i.e., will not be blocked by our algorithms). This is because the malicious traffic will be discarded by our detection algorithms, which could affect the performance measurement results. We remark that the results under benign traffic can reflect the real world performance well, as the malicious traffic usually takes up only a very low percentage of the overall traffic in the real world.

1) Testbed Setup: In the simulation, the FloodLight controller runs on a server with a quad-core Intel Core i7 CPU and 16GB of RAM running Ubuntu 12.04. Cbench runs on the same server and connect to FloodLight controller via loopback[3]. We adopt the configuration for FloodLight as suggested in [24] and evaluate the performance of the controller in the following scenarios.

- The controller runs without any detection algorithms enabled. This scenario serves as the baseline.
- The controller runs with each one of the four detection algorithms being enabled individually.
- The controller runs with all of the four detection algorithms being enabled at the same time.

For each scenario, we let Cbench emulate 16 switches, sending packet-in messages from ten thousand unique MACs per switch for 10 seconds. Each test is repeated for 10 times. And we present the average and standard deviation of the tests.

2) Simulation Results: The latency results are shown in Table II. While the Floodlight controller itself, with no detection algorithm enabled, achieves a latency of 2.09μs on average (baseline), we are more interested in the relative performance degradation when enabling our detection algorithms. In comparison, when our detection algorithms are enabled, we observed that there is no significant latency degradation. In particular, when the four detection algorithms are enabled individually, the average latency is increased by less than 3% in the worst case. Even when all the four detection algorithm are enabled at the same time, the average latency is still hardly affected (+3.2%). In addition, the results of standard deviation is relatively low, indicating a steady performance of our algorithms.

The throughput results are shown in Table III. The baseline result is 1034963 responses per second. When the four

[3]We let Floodlight controller and Cbench run on the same machine and connect to each other via loopback, so that the results are not affected by network related factors.

TABLE II. PERFORMANCE - LATENCY (μs, LOWER IS BETTER. THE NUMBERS IN PARENTHESIS REPRESENT THE RELATIVE PERFORMANCE COMPARED TO THE BASELINE RESULT)

	Average	Std. deviation
Baseline	2.09	0.01
Blacklist	2.11 (+1.0%)	0.02
Conn. Success Ratio	2.14 (+2.1%)	0.01
Throttling Conn.	2.15 (+2.7%)	0.06
Traffic Aggregation	2.13 (+1.8%)	0.02
All Four Together	2.16 (+3.2%)	0.02

detection algorithms are enabled individually, we observe that the average throughput is decreased by less than 10% in the worst case. When enabling all the four algorithms at the same time, we observe a throughput decrease of 27.9%. The results of standard deviation stay relatively low, indicating a steady performance of our algorithms under heavy traffic conditions.

TABLE III. PERFORMANCE - THROUGHPUT (RESPONSES PER SECOND, HIGHER IS BETTER. THE NUMBERS IN PARENTHESIS REPRESENT THE RELATIVE PERFORMANCE COMPARED TO THE BASELINE RESULT)

	Average	Std. deviation
Baseline	1034963	7147
Blacklist	1005333 (-2.9%)	22203
Conn. Success Ratio	956151 (-7.6%)	13724
Throttling Conn.	977168 (-5.6%)	20408
Traffic Aggregation	938217 (-9.3%)	25176
All Four Together	746347 (-27.9%)	5978

The performance of our algorithms can potentially be improved in many ways. First, we did not perform any special optimization of our source code. For example, one of the extensively used data structures, native Java 'map' and 'set' can be replaced by much faster ones provided by third-party libraries. Secondly, the four algorithms independently extract and save the traffic information, which can be potentially optimized by information sharing. Then the performance of four algorithm running at the same time can be improved considerably. Last, the OpenFlow controller can be run on a much more powerful server and the processing time will be further reduced.

D. Experiments on GENI testbed

The Global Environment for Network Innovations (GENI) is a suite of network research infrastructure, providing a laboratory environment for networking and distributed systems research and experimentation. The programmable hosts and programmable networks, especially the OpenFlow resources, provide us a unique opportunity to test and evaluate our malware detection system in SDN.

To experiment on GENI, we use the GENI experiment control tool, Omni [25], to reserve resources at GENI multiple aggregate managers. The steps are as follows. First, we need to create and renew our slice, which is essentially a collection of computation and communications resources, using the following commands:

```
omni.py createslice [Slice] % [Slice] represents the name of the slice
omni.py renewslice [Slice] [YYYYMMDD]
```

After a slice has been created, we create slivers in it. While slices are global containers of resources, slivers are small collections of resources from each source. For our experiments, we need both the computation resource and network resource. Therefore, we query two aggregate managers (one is for computation resource, the other is for networking resource) of GPOLab for their available resources with the following commands:

```
omni.py -a http://myplc.gpolab.bbn.com:12346/ -o listresources
omni.py -a https://foam.gpolab.bbn.com:3626/foam/gapi/1 -o listresources
```

The commands return two advertisement RSpec files, which list all the available resource that each aggregate manager currently controls. We pick the resources needed in two RSpect files and then reserve the resources using the following commands:

```
omni.py -a http://myplc.gpolab.bbn.com:12346/ createsliver [Slice] [RSpec]
omni.py -a https://foam.gpolab.bbn.com:3626/foam/gapi/1 createsliver [Slice] [RSpec]
```

Specifically, we have reserved three hosts (Sardis, Ganel and Bain), as well as an OpenFlow switch from the GPO aggregates. With the above steps, we are able to conduct experiments on GENI testbed. In our experiment, we design two scenarios as follows.

1) Single Switch: In this scenario, we connect only one switch to the controller. The topology of the experiment is shown in Fig. 4.

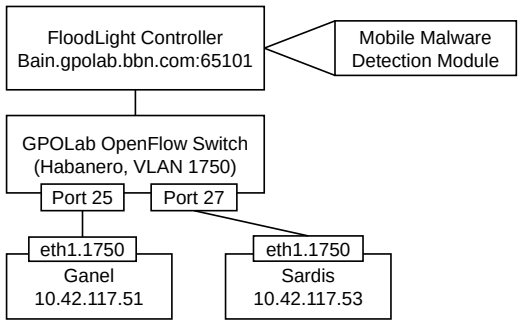

Fig. 4. Topology of single switch scenario.

Specifically, we use Bain to run our java-based Floodlight controller, and use the other two hosts (Sardis, Ganel) as the clients. To make the Floodlight controller run on Bain, we need to SSH into the node, and transfer the files required, and setup the execution environment using the following commands:

```
scp controller.jar pgenigpolabbbncom_[Slice]@bain.gpolab.bbn.com:
ssh pgenigpolabbbncom_[Slice]@bain.gpolab.bbn.com
sudo yum install java %the controller is java-based
```

After that, we ran the scripts that mimic the clients A, B on Ganel and scripts that mimic the clients C, D on Sardis (see Section IV-B). We verified the four algorithms in the same way as described in Section IV-A, confirming our algorithms worked as expected on GENI testbed.

2) Two Switches: To verify that the system can work in a more complicated environment, i.e., multiple switches, we connect two switches to the controller running on Bain. One switch is the same one as in the previous scenario, the other

is the switch used in our local testbed. The topology of the experiment is shown in Fig. 5.

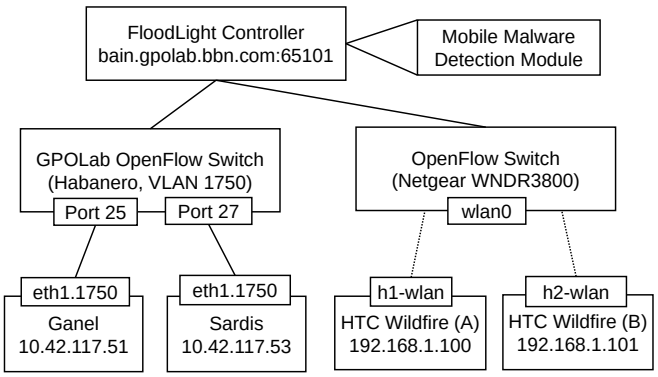

Fig. 5. Topology of two switches scenario.

In this experiment, we use the four hosts to simulate the four clients (A, B, C and D). And they work as expected. These simple experiments on GENI demonstrate the feasibility of our SDN-based approach in real networks. We note that our system is at its early stage, but we plan to extend our experiments to a larger scale and will keep improving the system design as the future work.

V. Related Work

The rapid adoption of mobile devices comes with the growing prevalence of mobile malware. To the best of our knowledge, there is no existing work on network level mobile malware detection in the context of SDN. In the following, we first describe the studies that address security issues using SDN, and then review existing researches on malware.

Several recent studies use SDN for security related issues. For instance, Shin and Gu proposed a framework that routes network packets to the pre-installed security devices using SDN [26]. Yap et al. proposed an approach that decouples authentication, access and accounting in guest Wi-Fi networking using SDN [27]. Clark et al. studied the admission control and monitoring at network level using OpenFlow [28]. Our work differs from the above studies in that we focus on mobile malware detection.

Malware related issues have been extensively studied in the literature. For instance, to understand how deeply malware has penetrated, studies have quantified the size of botnets [29], the number of executables infected with spyware [30], and the number of malicious web sites that launch drive-by downloads [31]. To investigate the behavior of malicious programs, there are been studies that investigate botnet activities [32], malware control mechanisms [33], and propagation mechanisms [34].

Several recent studies are specifically for mobile platforms. Zhou and Jiang showed that mobile malware is rapidly evolving and existing anti-malware solutions are seriously lagging behind based on over 1,2000 mobile malware samples [12]. Porras et al. analyzed an Apple iPhone bot client through reverse engineering [8]. And Damopoulos et al. designed and implemented a stealth and airborne malware that can wirelessly infect and self-propagate to iPhone devices [9]. To detect malware, Egele et al. designed a static analysis based tool to detect data flows for possible leaks of sensitive information [35].

Our approach differs from the above approaches in that we seek to detect malware by identifying suspicious network activity through real-time traffic analysis using SDN architecture.

VI. Conclusion and Future work

In this paper, we analyzed the behaviors of mobile malware and proposed several mobile malware detection algorithms that are implemented using SDN. Our algorithms detect mobile malware by identifying suspicious network activities through real-time traffic analysis. Experiments using both a local testbed and GENI infrastructure confirmed the feasibility of our approach and simulation results show that our detection algorithms do not incur significant performance degradation.

As future work, we will further study the characteristics of mobile malware, investigate more malware detection techniques and explore the possibilities of employing them in the context of SDN. In addition, we plan to take better advantage of GENI infrastructure and test our system at an even larger scale in order to optimize our system design.

Acknowledgement

This work is partially supported by GENI Subcontract 1855. In addition, we would like to thank GENI project office and NSF for sponsoring the First GENI summer camp, which provided us an opportunity to learn about GENI and the related resources. We would like to thank Mark Berman and Niky Riga from BBN for valuable comments and supports on our project. We would also like to thank Hyojoon Kim from Georgia Institute of Technology for his constructive suggestions on hardware configurations.

References

[1] Gartner identifies the top 10 strategic technology trends for 2013. [Online]. Available: http://www.gartner.com/it/page.jsp?id=2209615

[2] M. Hypponen, "Malware goes mobile," *Scientific American*, vol. 295, no. 5, pp. 70–77, 2006.

[3] T. Bradley. (2011) DroidDream becomes android market nightmare. [Online]. Available: http://www.pcworld.com/article/221247/droiddream_becomes_android_market_nightmare.html

[4] D. Goodin, "Backdoor in top iPhone games stole user data, suit claims," *The Register*, 2009.

[5] C. Guo, H. Wang, and W. Zhu, "Smart-phone attacks and defenses," in *HotNets III*, Nov 2004.

[6] J. Jamaluddin, N. Zotou, R. Edwards, and P. Coulton, "Mobile phone vulnerabilities: a new generation of malware," in *IEEE International Symposium on Consumer Electronics*, Sept 2004, pp. 199–202.

[7] Emerging cyber threats report 2012. [Online]. Available: http://www.gtisc.gatech.edu/doc/emerging_cyber_threats_report2012.pdf

[8] P. Porras, H. Saidi, and V. Yegneswaran, "An analysis of the iKee.B iPhone botnet," *Security and Privacy in Mobile Information and Communication Systems*, pp. 141–152, 2010.

[9] D. Damopoulos, G. Kambourakis, and S. Gritzalis, "iSAM: An iPhone stealth airborne malware," *Future Challenges in Security and Privacy for Academia and Industry*, pp. 17–28, 2011.

[10] Open Networking Foundation. Software-Defined Networking: The new norm for networks. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/white-papers/wp-sdn-newnorm.pdf

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[12] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy*, May 2012, pp. 95–109.

[13] X. Chen, R. Jin, K. Suh, B. Wang, and W. Wei, "Network performance of smart mobile handhelds in a university campus WiFi network," in *Proceedings of the Internet Measurement Conference*, Nov 2012, pp. 315–328.

[14] Blocklists of suspected malicious IPs and URLs. [Online]. Available: http://zeltser.com/combating-malicious-software/malicious-ip-blocklists.html

[15] Directory of malicious IPs. [Online]. Available: http://www.projecthoneypot.org/list_of_ips.php

[16] S. Schechter, J. Jung, and A. Berger, "Fast detection of scanning worm infections," in *Recent Advances in Intrusion Detection (RAID)*. Springer, Sept 2004, pp. 59–81.

[17] M. Williamson, "Throttling viruses: Restricting propagation to defeat malicious mobile code," in *Proceedings of the Annual Computer Security Applications Conference*, Dec 2002, pp. 61–68.

[18] T. Yen and M. Reiter, "Traffic aggregation for malware detection," *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 207–227, 2008.

[19] Global environment for network innovations. [Online]. Available: http://geni.net

[20] Floodlight: An open SDN controller. [Online]. Available: http://floodlight.openflowhub.org/

[21] Pantou: OpenFlow 1.0 for OpenWrt. [Online]. Available: http://www.openflow.org/wk/index.php/Pantou:_OpenFlow_1.0_for_OpenWrt

[22] OpenWrt. [Online]. Available: https://openwrt.org/

[23] CBench. [Online]. Available: http://sourceforge.net/apps/trac/cbench

[24] How to benchmark controller performance. [Online]. Available: http://www.openflowhub.org/display/floodlightcontroller/Cbench

[25] Omni. [Online]. Available: http://trac.gpolab.bbn.com/gcf/wiki/Omni

[26] S. Shin and G. Gu. CloudWatcher: Network security monitoring using openflow in dynamic cloud networks. [Online]. Available: http://people.tamu.edu/~seungwon.shin/Cloudwatcher.pdf

[27] K. Yap, Y. Yiakoumis, M. Kobayashi, S. Katti, G. Parulkar, and N. McKeown, "Separating authentication, access and accounting: A case study with OpenWiFi," Open Networking Foundation, Tech. Rep., 2011.

[28] R. Clark, N. Feamster, A. Nayak, and A. Reimers, "Pushing enterprise security down the network stack," Georgia Institute of Technology, Tech. Rep., 2009.

[29] M. Fabian and M. Terzis, "A multifaceted approach to understanding the botnet phenomenon," in *Proceedings of the Internet Measurement Conference*, Oct 2006.

[30] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy, "A crawler-based study of spyware on the web," in *Proceedings of the Network and Distributed System Security Symposium*, Feb 2006, pp. 17–33.

[31] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All your iFRAMEs point to us," in *Proceedings of the 17th conference on Security Symposium*, Jul 2008, pp. 1–15.

[32] J. Zhuge, T. Holz, X. Han, J. Guo, and W. Zou1, "Characterizing the IRC-based Botnet Phenomenon," Reihne Informatik, Tech. Rep., Dec. 2007.

[33] P. Barford and V. Yegneswaran, "An inside look at botnets," *Malware Detection*, pp. 171–191, 2007.

[34] A. Karasaridis, B. Rexroad, and D. Hoeflin, "Wide-scale botnet detection and characterization," in *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, 2007, pp. 7–7.

[35] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting privacy leaks in iOS applications," in *Proceedings of the Network and Distributed System Security Symposium*, Feb 2011.