Enforcing Context-Aware BYOD Policies with In-Network Security

Adam Morrison[†‡], Lei Xue[*], Ang Chen[†], Xiapu Luo[*]

[†]Rice University, [*]The Hong Kong Polytechnic University

Abstract

Bring Your Own Device, or BYOD, has become the new norm for many enterprise networks; but it also raises security concerns. We present our vision of *programmable in-network security*, and sketch an initial system design, Poise. Poise has a high-level policy language that can express a wide range of existing and new security policies. These policies can then be compiled to device configurations to collect device/apps information, as well as switch programs in P4 that enforce security inside the network. Our initial results seem promising—Poise runs with reasonable overhead, and it successfully detects policy violations for seven useful BYOD policies.

1 Introduction

BYOD, or Bring Your Own Device, refers to the policy that allows enterprise employees to use privately owned tablets, phones, and laptops at work [21]. It has become the new norm in many companies [5, 10–12, 16, 20]: it is estimated that the BYOD market will exceed $73 billion by 2021 [12]. The rise of BYOD is not only because it reduces device ownership cost of the companies, but also because it proves to boost productivity and improve employee satisfaction—in part because they can directly work with devices they are already familiar with [1, 3].

Despite its popularity, BYOD devices raise security concerns, because they are less well-managed than their enterprise-owned counterparts [4,6,27,53], and they may be used to access both sensitive services inside the enterprise and untrustworthy services in the wild [3, 15]. In fact, BYOD security represents a concrete instance of a more fundamental challenge, sometimes known as "the end node problem" [8, 9]. The "end nodes" are not subject to the same central control and management as the core infrastructure, such as the network and servers. For instance, the enterprise infrastructure can be updated relatively easily to patch a security problem, but ensuring that all BYOD devices are properly patched is much more challenging. As such, insecure end devices tend to become the weakest link in the security chain [18].

Therefore, it is important to establish and enforce security policies for BYOD devices [32]. Although some BYOD policies, such as blacklisting certain services [37], can be supported by conventional enterprise security solutions, plenty of them go beyond what off-the-shelf solutions can offer. For instance, a policy may deny access from a client if its TLS library is not the most recent version [51], it may allow access to a sensitive service only if the client is physically located in the server room [45], or it may grant access to a client only if another admin device is online [31, 44]. These policies refer to device-specific *context*, such as sensor readings, library versions, and active apps, which are not directly visible from a request's packet header. Such context awareness also makes the policies rather *diverse*—enterprises may require drastically different policies depending on their device types and services, and even the same enterprise may need to update its policies dynamically in response to new security problems.

Existing work has developed security solutions at the server side [46, 50] and at the client side [32, 46, 48, 50, 51]. Security enforcement at the server side is easier to manage and update, as the enterprise has central control over the servers; this also places a minimum amount of trust on the clients, as sensitive policies are maintained at the server side and their enforcement does not assume client cooperation. However, much of the device-specific context essential to BYOD security is only available at the client side. Client-side solutions, on the other hand, can readily access such information, but delegating policy enforcement to the clients requires a strong trust in the BYOD devices; such solutions also cannot easily support *network-wide* policies—such as granting access to a device only if another admin device is online—as clients only have device-local information. Ideally, we desire a solution that achieves the "best of both worlds": a similar security guarantee as server-side systems, and a similar context awareness as client-side systems.

In this paper, we explore this point in the design space and propose *programmable in-network security*, or Poise. Poise does not modify server-side software; it only requires clients to send occasional context information, and it moves the entire policy enforcement logic inside the network. Poise still relies on the end devices to *collect* their context information, but the policies are kept private to the clients and enforced inside the well-protected enterprise infrastructure, which significantly lowers the amount of trust granted to the end

‡ Morrison, Xue: Co-primary authors; Chen, Luo: Corresponding authors.

devices compared to existing work [32, 48]. This architecture also preserves the ease of policy maintenance and update found in server-side solutions.

Although a number of in-network solutions already exist, they only support *fixed functions* that are difficult to programmatically control or update. For instance, one could install access control list (ACL) rules on switches for traffic filtering [19], but ACL rules can only match on a limited set of header fields. One could also deploy custom hardware appliances for traffic scrubbing [17, 40], but hardware middleboxes are specialized to perform specific tasks and cannot easily support evolving policies. One could in principle virtualize the middleboxes as VMs that run on the end servers for more programmability [51], but software-based middleboxes often cannot match the performance of their hardware counterparts.

Our key insight is to leverage recent advances in *programmable data planes* [26] to achieve both programmability and performance. This emerging switch architecture allows a customized definition of packet headers, which can be used to store and parse user-defined contexts, and a richer set of operations over header fields, which can be used to make policy decisions at linespeed. Poise installs a lightweight module on BYOD devices to collect device context, and embeds it in customized headers recognizable by programmable switches. The switch programs are generated by the Poise compiler, which analyzes the BYOD policies and maps them into P4 programs [14] that run on the data plane.

In the rest of this paper, we sketch an initial system design of Poise, propose a policy language that captures a wide range of existing and new BYOD policies, and present our preliminary experience with Poise.

2 Programmable In-Network Security

Security policies can be enforced at the server side [46, 50], at the client side [32, 48], or inside the network [17, 40]. Server-side and client-solutions typically offer more *programmability* than in-network solutions, as servers and end devices are equipped with general-purpose processors and can implement a wide range of policies, whereas the network only provides fixed functions specialized for packet processing. For instance, PBS [32] installs a software switch inside BYOD clients, which can dynamically accept policy configurations from the controller and enforce them on the clients. However, client-side solutions suffer from three limitations: a) they do not support network-wide policies, as each device only has a local view; b) policy information is completely revealed to the clients; and c) policy enforcement on the clients introduce extra processing overhead to resource-constraint mobile devices. Server-side solutions are free from these limitations, but they cannot easily obtain device-specific context information, which is a critical part of security policies [32, 33].

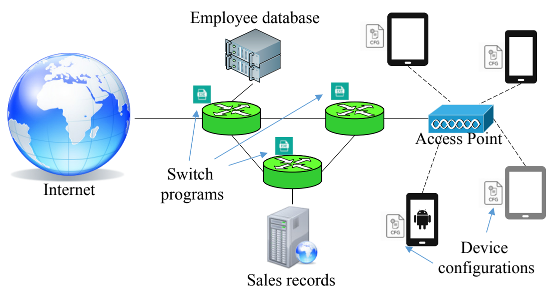

Figure 1: Poise compiles a high-level policy into a) switch programs, and b) device configurations, and enforces the policy inside the network.

In proposing to move the security enforcement inside the network, Poise must address the lack of programmability in traditional networks. In order to support dynamic, evolving policies that depend on application-layer and/or device-specific context, the network needs to be able to a) recognize context information associated with a packet, and b) perform customized processing depending on the context. Unfortunately, neither is feasible in a traditional network with fixed-function switches.

Recent advances in *programmable data planes*, however, are quickly changing the landscape. Emerging switch ASICs, such as Barefoot Tofino [2] and Cavium XPliant [7], are designed with a reconfigurable match-action architecture [26]. Switch programs written in P4 [25] or PoF [43] can be compiled to run on these switches, parse customized packet formats, and perform a richer set of operations over header fields at line speed.

Poise leverages programmable switches to enforce context-aware BYOD policies in the network. Concretely, enterprise administrators can write security policies in a high-level language. The Poise compiler will automatically generate BYOD configurations to collect device-level contexts, as well as switch programs in P4 that make policy decisions using the contexts carried in packets. This incurs minimum overhead, as BYOD clients only occasionally send out context packets, and all packet processing happens in the network at hardware speeds. Figure 1 shows the architecture of Poise, which we will describe in more detail in the following sections.

Threat model: Poise assumes that BYOD devices can be compromised, e.g., by malicious apps, but that a) the enterprise network is trustworthy, and b) the OS kernel in the device and the BYOD module that collects device contexts in the kernel are intact. a) is reasonable because the enterprise infrastructure tends to be more well-protected than the BYOD clients. b) still requires some trust on the devices, but it is already much weaker than existing work [32] that enforces security on-device.

3 The Poise Policy Language

Poise supports a wide range of existing and new security policies for BYOD. Similar with other SDN languages such as NetKAT [24] and Pyretic [36], a Poise policy represents a function that maps an incoming packet to zero (i.e., drop), one (i.e., unicast), or more (i.e., multicast) outgoing packets. A policy could be as simple as `drop`, which drops all packets, although more practically, the policy would make a decision based on the context a packet carries, such as `match(dip==66.220.144.0) >> drop`, which blacklists certain destination addresses, or `match((time>=0800)&(time<=1800)) >> drop`, which blocks access depending on the time of day.

More generally, the Poise policies are written in a new language that we have designed based on Pyretic NetCore [36]. Policies consist of a series of match-action statements, such as the ones discussed above. Additionally, some policies may involve a monitor expression, which collects traffic statistics that will be used for making decisions. A monitor expression is written as `count(pred)`, which counts the number of packets that satisfy the predicate `pred` in the current time window; for instance, `count(match(is_admin))` counts the number of packets generated from a device with an administrative role. The counters are periodically reset to zero when a new time window begins. These monitors enable programmers to write network-wide policies, where the processing of a packet depends on not only its own context, but also the context of other traffic.

Poise policies could also contain pre-defined constant lists, which are typically used to encode membership relations. For instance, one could define a list of devices with administrative roles as `def adminlst = ["dev1", "dev2"]`. Then, the decision subpolicy could refer to the lists as part of the decision-making process, such as `match(!dev in adminlist) >> fwd(mbox)`, which forwards traffic from non-admin devices to a middlebox for traffic scrubbing.

Poise policies can be composed sequentially or in parallel, similar to NetCore. A sequential composition `P1>>P2` pipes a packet through both policies in order. Parallel compositions, written as `P1|P2`, apply both policies to the same packet at the same time. Figure 2 summarizes the syntax of our policy language, and the highlighted portions show the differences from NetCore.

Primitive Actions

$A \;::=\; \mathsf{drop} \mid \mathsf{fwd(port)} \mid \mathsf{flood}$

Expressions

$E \;::=\; \mathsf{v} \mid \mathsf{e_1 + e_2} \mid \mathsf{e_1 - e_2} \mid \mathsf{e_1 * e_2} \mid M$

Constant Lists

$L \;::=\; \mathsf{def\ l = [v]}$

Predicates

$P \;::=\; \mathsf{match(e_1 \circ e_2)} \mid \mathsf{match(h \circ e)} \mid$
$\qquad\quad \mathsf{match(h\ in\ l)} \mid \mathsf{P\&P} \mid \mathsf{(P|P)} \mid \mathsf{!P}$

Monitors

$M \;::=\; \mathsf{count(P)}$

Policies

$C \;::=\; \mathsf{A} \mid \mathsf{(C|C)} \mid \mathsf{C >> C}$
$\qquad\quad \mathsf{if\ P\ then\ C\ else\ C}$

Figure 2: The language syntax for Poise policies.

3.1 Policies by example

Next, we describe seven practical BYOD policies, where the first two are adapted from existing work [32] and the rest are new policies supported by Poise. Variables `dev`, `time`, `lat`, `lon`, and `usr` are customized header fields.

P1: Block certain services in work hours [32]: A common BYOD policy is to block access from certain devices to entertainment websites during work hours:

```
def businesslst = ["dev1", "dev2"]
match(dip==66.220.144.0) >>
match(dev in businesslst) >>
match((time>=0800)&(time<=1800)) >> drop
```

P2: Direct traffic from guest devices through a middlebox [32]: Another useful policy is to distinguish traffic from authorized devices and guest devices, and direct guest traffic through a middlebox for traffic scrubbing:

```
def authlst = ["dev1", "dev2"]
if match(dev in authlst)
then  fwd(server)
else  fwd(mbox)
```

New policies: There are also useful policies in Poise that cannot be easily supported in traditional networks; they are implementable in Poise due to the use of programmable data planes, which can perform simple arithmetic operations and maintain switch-local states.

P3: Distance-based access control: This policy grants access to a service only if the user is within a certain distance from a physical location (e.g., the server room); this requires performing arithmetic operations over GPS coordinates embedded in the packet header:

```
if ((lat-x)*(lat-x)+(lon-y)*(lon-y) < D)
then  fwd(server)
else  drop
```

P4: Allow access only if admin is online: Poise can support network-wide policies by monitoring certain events and making decisions based on the result. One such example is to grant access to a service only if the admin device is online:

```
def adminlst = ["Bob", "Alice"]
c = count(match(usr in adminlst))
match(c>0) >> fwd(server)
```

Advanced policies: Inspired by the literature of "continuous authentication" [23, 28, 29, 49], we propose a set of advanced policies that leverage device contexts to detect subtle but important indicators of potential attacks. Due to space constraints, we only describe the high-level policy, but not the programs. *P5: Block requests without explicit user interaction*, which denies access to a sensitive service if all apps are running in the background and there is no user interaction with the touchscreen, because such requests are likely generated by malware. *P6: Scrub traffic if UIs are overlapping*, which forwards traffic through a middlebox if the context information shows that app UIs are overlapping — a potential sign for attacks [30]. *P7: Conduct deep packet inspection if camera/recorder is on*, which detects if sensitive information is being leaked through an active camera/recorder app [22].

4 The Poise System

Next, we describe the architecture of Poise, which has three components: a) a client module that runs on each BYOD device to collect context information and tag packets, b) a compiler that generates switch programs and device configurations, and c) a runtime system.

4.1 The client module

The Poise client module is responsible for collecting device context and embedding it in the network traffic. As shown in Figure 3, it has a userspace component and a kernel module. The kernel module uses `netfilter` hooks to intercept network packets, and tags selected packets with three types of tags: a) app information, such as `UID`, b) system information, such as screen light status, and c) sensor readings, such as accelerometer and gyroscope readings. This kernel module can be further extended to protect the user-space component [41].

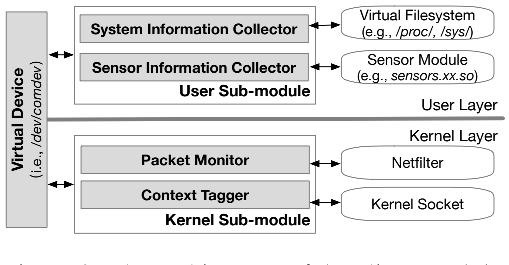

Figure 3: The architecture of the client module.

The kernel module periodically crafts special "context" packets with tags containing b) and c), and sends them out to the network. Moreover, when an app opens a new socket, or an existing socket sends out packets after being dormant for a while, the module generates a context packet tagged with a), b) and c). The kernel module obtains a) by getting `UID` from the `sk_buff` structure of each packet in the socket's send buffer and looking up its owning app; the current implementation obtains b) and c) from the userspace component through a registered virtual device, although our final prototype will directly obtain this from the Hardware Abstraction Layer (HAL).

The userspace component runs as a system service. It starts upon boot, and periodically collects system information from the `/proc` and `/sys` virtual filesystems and sensor readings from various sensor modules. For instance, it gets the screen brightness information from the virtual file `/sys/class/leds/lcd-blacklight/brightness`, and it obtains the sensor readings from the HAL module *sensor.msm8794.so* on Nexus 5, which is loaded by invoking the native interface *hw_get_module()* of *libhardware.so*. This component only collects information relevant to the desired policy, based on the device configuration generated by the compiler.

4.2 Compiler

Our compiler takes in a policy program, and generates two types of outputs: a) a configuration file for the BYOD devices, which describes the information the client module should collect and embed in the packets, and b) switch programs written in P4, which is to be deployed on the programmable switches to enforce the policy in-network. At the time of writing, our compiler prototype is still unfinished, but we have designed the compilation strategies our prototype will use.

Switch programs: If the Poise policy contains a monitoring subpolicy, our compiler will instantiate a read-write register to hold the monitored value, and generate logic to update the register when the monitoring predicate is fulfilled, and to reset the register after a predefined period of time has elapsed. For a constant list, our compiler generates a match-action table, where the match keys are the elements in the list, and the values are simply dummy values since they are unimportant. A membership test on the list can be implemented as matching a packet's tag with the table, and seeing whether there is a successful match. The expressions in the Poise program can be mapped directly to their P4 counterparts.

Device configurations: Our compiler will additionally generate configuration files that specify which types of information to collect from the BYOD devices, the format and sequence of tags in the context packets. The compiler ensures that the switch programs in P4 contain the same definition and order of tags, so that the context packets can be parsed correctly.

4.3 Runtime

The Poise runtime is implemented in an SDN controller that configures P4 programs on the switches, and com-

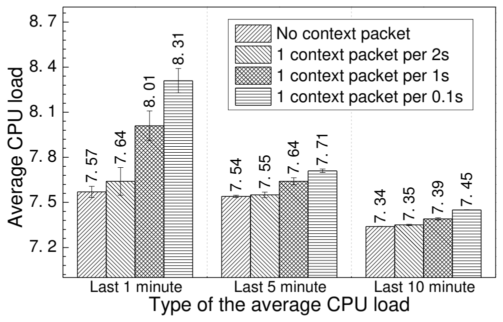

Figure 4: Poise incurs small CPU overhead.

municates with BYOD devices to deploy context configurations. The centralized controller is not a performance bottleneck, as it is not involved in packet processing; per-packet decisions are made directly on the switch data plane, and the controller is only used for policy deployment or change, which is typically infrequent.

5 Initial Results

Although a full implementation of our Poise system is still ongoing, we have conducted an initial validation using a preliminary prototype. The client module is implemented in C, the policies are hand-written in P4, and the runtime consists of several Python scripts integrated with the Mininet [35] environment to instantiate network topologies with programmable switches. The P4 programs are compiled to the `bmv2` model [13] and run inside the software switches in Mininet.

Experimental setup: Our mobile traces are collected from a Nexus 5 device running the client module; these traces contain requests that pass or violate the security policies we have described in Section 3. We then injected these traces to Mininet with one programmable switch and six end hosts, three as clients and three as servers in the enterprise network. Our primary metrics include the overhead incurred by the client module, the traffic overhead due to tagging, and, most importantly, the effectiveness of Poise in detecting security violations.

Overhead: Figure 4 shows the average CPU overhead with different frequencies of context packets; the overhead results were collected for the last 1/5/15 minutes, respectively, from `/proc/loadavg`. As the results show, Poise incurs less than 1% overhead consistently. In terms of traffic overhead, at a sending rate of one context packet per second, the additional traffic is under 11.2 kbps, which also seems to be within a reasonable range.

Effectiveness: We have tested all seven policies P1–P7 in Section 3, and our Poise prototype successfully detected all requests that violate one of the policies, and it granted access to all other policy-compliant requests. This is good news, because it shows that programmable in-network security indeed seems to be a promising basis for new BYOD security solutions.

6 Related Work

BYOD security: Security policies can be enforced at the client side, such as in DeepDroid [48] and PBS [32], or at the server side [46, 50]; we have discussed their pros and cons in Section 2. [52] sketches a solution for context-aware IoT security, and proposes to use software middleboxes to enforce security. In contrast, Poise leverages emerging programmable switches to implement security policies at hardware speeds inside the network.

Enterprise security: There is a long body of work in enterprise security, including PSI [51], which uses virtualized middleboxes for security, Kinetic [34] which supports automatic verification of control programs, FortNOX [38], which supports policy conflict detection and resolution, and OFX [42], which uses SDNs to install arbitrary security applications in existing physical middleboxes. These traditional enterprise security solutions do not address challenges that arise in a BYOD context, such as supporting context-aware security policies.

Policy languages: Many domain-specific languages have been proposed for networking, such as in NetKAT [24], Concurrent NetCore [39], PSI [51], Nettle [47], PBS [32], Pyretic [36]. Although our key idea of programmable in-network security is not tied to any specific language, the Poise language is closest to NetCore [36], an SDN language recognized for its ease of use. Our language also extends NetCore to a) specify application contexts, and b) capture a richer set of operations supported on P4 switches.

7 Summary and Future Work

In this paper, we have sketched our vision of programmable in-network security for BYOD devices, as well as a system design that we call Poise. In Poise, administrators can express a rich set of BYOD policies in a high-level language; our compiler then generates device configurations to collect BYOD contexts, as well as switch programs to enforce the policies on emerging programmable data planes. Our initial prototype and results demonstrate that Poise supports a wide range of existing and new policies with reasonable overhead. Nevertheless, much is left to be done: in ongoing work, we are building a full prototype of Poise, and performing larger-scale experiments on real-world BYOD traces.

Acknowledgements We thank Ryan Beckett for his feedback on the policy language, and the anonymous reviewers for their insightful comments; we acknowledge support from the Hong Kong GRF (152279/16E) and the Hong Kong RGC Project (CityU C1008-16G).

References

[1] Algarytm: BYOD. https://goo.gl/6PU5gq.

[2] Barefoot tofino. https://goo.gl/gXynFN.

[3] The benefits and risks of BYOD. https://goo.gl/ym9ATg.

[4] Bring your own risk with BYOD. https://goo.gl/bn1rN4.

[5] BYOD: A global perspective. https://goo.gl/BTrSm4.

[6] BYOD: Mobile devices threats and vulnerabilities. https://goo.gl/phTav6.

[7] Cavium xpliant. https://www.cavium.com/.

[8] End node. https://goo.gl/D99C39.

[9] How to solve the end node problem. https://goo.gl/9wWqJr.

[10] IBM Mobile: BYOD. https://goo.gl/zafGxN.

[11] IBM opens up smartphone, tablet support for its workers. https://goo.gl/WBn3vP.

[12] Market reports. https://goo.gl/25SX7K.

[13] P4 behavioral model repository. https://github.com/p4lang/behavioral-model.

[14] The P4 language repositories. https://github.com/p4lang.

[15] The rise and risk of BYOD. https://www.druva.com/blog/the-rise-and-risk-of-byod/.

[16] Samsung BYOD solutions. https://goo.gl/GmZ1io.

[17] Scrubbing away DDoS attacks. https://goo.gl/HRV3Ce.

[18] Securing your weakest link: Your mobile devices. https://goo.gl/Z769MG.

[19] Security configuration guide: Access control lists, Cisco IOS XE Release 3S. https://goo.gl/zTJaUL.

[20] Top 21 companies in the BYOD market. https://goo.gl/MuRr66.

[21] What is BYOD and why is it important? https://goo.gl/H71Nji.

[22] P. Aditya, R. Sen, P. Druschel, S. Joon Oh, R. Benenson, M. Fritz, B. Schiele, B. Bhattacharjee, and T. T. Wu. I-pic: A platform for privacy-compliant image capture. In *Proc. MobiSys*, 2016.

[23] A. Alzubaidi and J. Kalita. Authentication of smartphone users using behavioral biometrics. *IEEE Communications Surveys& Tutorials,*, 18, 2016.

[24] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL*, 2014.

[25] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3), 2014.

[26] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proc. SIGCOMM*, 2013.

[27] D. Dang-Pham and S. Pittayachawan. Comparing intention to avoid malware across contexts in a byod-enabled australian university: A protection motivation theory approach. *Computers & Security*, 48, Feb. 2015.

[28] S. Eberz, K. B. Rasmussen, V. Lenders, and I. Martinovic. Evaluating behavioral biometrics for continuous authentication: Challenges and metrics. In *Proc. AsiaCCS*, 2017.

[29] M. Ehatisham-ul-Haqa, M. A. Azama, U. Naeemb, Y. Amina, and J. Looc. Continuous authentication of smartphone users based on activity pattern recognition using passive mobile sensing. *Journal of Network and Computer Applications*, 109, 2018.

[30] Y. Fratantonio, C. Qian, P. Chung, and W. Lee. Cloak and dagger: From two permissions to complete control of the UI feedback loop. In *Proc. IEEE S&P*, 2017.

[31] C. K. Georgiadis, I. Mavridis, G. Pangalos, and R. K. Thomas. Flexible team-based access control using contexts. In *Proc. SACMAT*, 2001.

[32] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu. Towards SDN-defined programmable BYOD (bring your own device) security. In *Proc. NDSS*, 2016.

[33] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *Proc. NDSS*, 2016.

[34] H. Kim, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *Proc. NSDI*, 2015.

[35] Mininet. `http://mininet.org/`.

[36] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proc. NDSI*, 2013.

[37] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *Proc. LISA*, 2010.

[38] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for openflow networks. In *Proc. SIGCOMM*, 2012.

[39] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent NetCore: From policies to pipelines. In *Proc. ICFP*, 2014.

[40] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The middlebox manifesto: Enabling innovation in middlebox deployment. In *Proc. HotNets*, 2011.

[41] Y. Shao, X. Luo, and C. Qian. Rootguard: Protecting rooted android phones. *IEEE Computer*, 47(6), 2014.

[42] J. Sonchack, A. Aviv, E. Keller, and J. Smith. Enabling practical software-defined networking security applications with OFX. In *Proc. NDSS*, 2016.

[43] H. Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proc. HotSDN*, 2013.

[44] W. Tolone, G.-J. Ahn, and T. Pai. Access control in collaborative systems. *ACM Computing Surveys*, 37, 2005.

[45] N. Ulltveit-Moe and V. Oleshchuk. Enforcing mobile security with location-aware role-based access control. *Security and Communication Networks*, 9, 2016.

[46] VMware. Next generation security with VMware NSX and Palo Alto Networks VM-series. In *White Paper*, 2013.

[47] A. Voellmy, A. Agarwal, P. Hudak, N. Feamster, S. Burnett, and J. Launchbury. Don't configure the network, program it! domain-specific programming languages for network systems. In *Proc. SIGCOMM*, 2010.

[48] X. Wang, K. Sun, Y. Wang, and J. Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *Proc. NDSS*, 2015.

[49] X. Wang, T. Yu, O. Mengshoel, and P. Tague. Towards continuous and passive authentication across mobile devices: an empirical study. In *Proc. WiSec*, 2017.

[50] R. Ward and B. Beyer. BeyondCorp: A new approach to enterprise security. *USENIX ;login:*, 39, 2014.

[51] T. Yu, S. K. Fayaz, M. Collins, V. Sekar, and S. Seshan. PSI: Precise security instrumentation for enterprise networks. In *Proc. NDSS*, 2017.

[52] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-Things. In *Proc. HotNets*, 2016.

[53] N. Zahadat, P. Blessner, T. Blackburn, and B. Olson. BYOD security engineering: A framework and its analysis. *Computers & Security*, 55, Nov. 2015.