BORDERPATROL: Securing BYOD using fine-grained contextual information

Onur Zungur
Boston University
Boston, USA
zungur@bu.edu

Guillermo Suarez-Tangil
King's College London
London, UK
guillermo.suarez-tangil@kcl.ac.uk

Gianluca Stringhini
Boston University
Boston, USA
gian@bu.edu

Manuel Egele
Boston University
Boston, USA
megele@bu.edu

***Abstract*—Companies adopt Bring Your Own Device (BYOD) policies extensively, for both convenience and cost management. The compelling way of putting private and business related applications (apps) on the same device leads to the widespread usage of employee owned devices to access sensitive company data and services. Such practices create a security risk as a legitimate app may send business-sensitive data to third party servers through detrimental app functions or packaged libraries.**

In this paper, we propose BORDERPATROL, a system for extracting contextual data that businesses can leverage to enforce access control in BYOD-enabled corporate networks through fine-grained policies. BORDERPATROL extracts contextual information, which is the stack trace of the app function that generated the network traffic, on provisioned user devices and transfers this data in IP headers to enforce desired policies at network routers. BORDERPATROL provides a way to selectively prevent undesired functionalities, such as analytics activities or advertisements, and help enforce information dissemination policies of the company while leaving other functions of the app intact. Using 2,000 apps, we demonstrate that BORDERPATROL is effective in preventing packets which originate from previously identified analytics and advertisement libraries from leaving the network premises. In addition, we show BORDERPATROL's capability in selectively preventing undesirable app functions using case studies.

I. INTRODUCTION

Mobile smart-devices are omnipresent not only in a personal setting but also in business environments. To accommodate for this reality, corporations increasingly implement so-called Bring Your Own Device (BYOD) processes that allow employees to access corporate data and applications (apps) on their personal smart-devices. However, corporate priorities, such as protecting intellectual property or preventing data leaks, frequently differ from the priorities of users.

It is therefore customary for corporations to enforce restrictions on which apps are allowed to run on a mobile device or what network properties the device is allowed to access. Such restrictions allow companies to prevent apps deemed unwanted or dangerous from executing while the device is in the corporate network. Widespread solutions often include commercial device management products such as Samsung Knox [1] and software specific built-in capabilities such as the Android Device Management (ADM) framework [2]. However, such policy enforcement systems are prone to failure in restraining communications that originate from mobile devices, causing undesirable information flows that violate company policies. One such example is the news leak of an Apple iPad prototype [3] through apps that use the Flurry analytics software [4]. In this case, Flurry aggregated analytics and geo-location data from apps which included the Flurry library, and identified approximately 50 devices that matched the hardware characteristics of Apple's rumored tablet device at company headquarters well before the official launch.

A straightforward solution to prevent undesirable information flow from employee owned devices is to use a BYOD policy that prevents apps unrelated to the corporate agenda from executing. Unfortunately, most apps, including those that are geared towards businesses, are an amalgamation of developer-authored code and various third party libraries. This poses a challenge from the perspective of designing a BYOD policy that will provide selective permeability in access control. That is, while an app might be in line with the corporate agenda, and hence should be allowed to execute, the app might be bundled with libraries that violate the security and privacy goals of the company. For instance, business apps for word processing such as Docs To Go [5] often include tracking libraries that send detailed information about the device for usage analysis and statistics to the developer.

Preventing communication with network endpoints that the corporation has identified to be against its own priorities is an attractive solution. Unfortunately, this only works for endpoints that are exclusively accessed by unwanted components of an app, such as known tracker, statistics, or advertisement libraries. The network traffic that is visible by the corporate infrastructure does not usually contain sufficient information to make an informed decision about whether the communication should be allowed or prohibited. A concrete example is the Dropbox Android app [6]. This app uses a variety of different developer-created functionalities which serve two main purposes: (i) login and authentication functionality, and (ii) file synchronization with both upload and download functionalities. While all functionalities are necessary for the daily use of the app, uploading confidential information to non-business affiliated servers might violate the corporate policy.

Companies that adopt BYOD policies are also confronted with novel regulatory challenges such as the European Union's General Data Protection Regulation (GDPR) [7]. GDPR expressively forbids employees to upload customer data to third party services unless the company has obtained explicit consent from the customer. Thus, an ideal BYOD policy for

an app should allow authentication and permit file downloads, but prevent any uploads. Unfortunately, existing BYOD policy enforcement mechanisms, such as Samsung KNOX or ADM, lack the granularity to enforce such a policy as they do not inspect network connections with respect to the app context, but merely examine the data residing in IP packets such as source/destination addresses.

Hence, to enable such flexible BYOD policies, we propose to augment the network traffic originating from a device with fine-grained, app-execution context information. This information allows a network-based policy enforcement mechanism to pinpoint what functionalities of an app are responsible for sending the corresponding traffic. The contextual information we specifically leverage is the Java call stack at the time a network socket is connected. We then embed a compressed representation of this call stack in the `options` field of the IP header in network packets. This information allows a policy engine to easily distinguish different app functions, including authentication and file uploads/downloads, and to selectively prevent undesired functionalities.

To demonstrate the practicality and effectiveness of this idea, we implemented a prototype of BORDERPATROL on Android as an example of a BYOD-managed smart device and provided a policy enforcement mechanism that integrates seamlessly into Linux's net-filter mechanism. We then evaluated BORDERPATROL on 2,000 apps from Google Play's BUSINESS and PRODUCTIVITY categories. BORDERPATROL successfully enforces policies that prevent data leakage through tracking libraries as well as more fine-grained policies where a single network endpoint is used for both desired and potentially harmful purposes. In summary, this paper makes the following contributions:

- We propose a novel, network-wide, fine-grained policy enforcement scheme for BYOD devices with a reprogrammable access control framework which is aware of mobile app contexts.
- We design (§IV) BORDERPATROL, as a system that augments network traffic originating from BYOD-managed smart devices with additional contextual information, which allows fine-grained policies to be enforced at the corporate network level. We then implement (§V) a prototype to demonstrate that BORDERPATROL can enforce policies with minimal modifications to existing systems and at negligible throughput, latency, and performance overheads even when seeking to thousands of connections.
- We present our findings from our analysis of 2,000 apps from the BUSINESS/PRODUCTIVITY categories of Google Play (§VI) and demonstrate the effectiveness and utility of BORDERPATROL through case studies (§VI-C).

II. BACKGROUND

As a basis for the details of our proposed system, BORDERPATROL, this section describes Android application packages, networking subsystem in Linux and protocol specifications for IP packets.

A. Composition of an Android application

Android apps are distributed as Application Package Kit (`apk`) files. This package includes an app's compiled code (commonly compiled from Java source) as well as resources, assets, certificates, and manifest files. The app's code is stored in the Dalvik bytecode format in a file called `classes.dex`. Besides the implementation of the methods in an app, the Dalvik file-format also prescribes how to store meta-information about the app. For our purposes, we are interested in the class hierarchy, method signatures, and debug information contained in a `dex` file.

The class hierarchy of a Java app is a graph that represents the inheritance relationships between classes. Java programs and APIs frequently bundle related classes in so-called packages. Within each class, a method is uniquely identified by the method's signature, which consists of the method's name and the types of the method's parameters. In addition, Java supports method overloading, where within the same class, multiple methods share a common name but have a different list of their parameter types, thus have different method signatures. Hence, a method can be uniquely identified within an app by the method's signature. In addition, the Dalvik format contains provisions to store debug information along with the byte-code to easily determine and debug the source of an exception in stack traces. This information can map individual byte-code instructions to the source file and line number of the Java code that produced the `dex` file.

B. Networking in Linux and Java

1) Sockets: A socket is one of the most central aspects of networking in Linux (and Android) as well as in Java. Any network communication in Linux will commence with a `socket` system call, and the system call's return value (a file descriptor) uniquely identifies the socket within a given process. This requirement holds true independently of whether an Android app establishes a network connection from managed Dalvik code or whether it uses native code. While Java also provides a `java.net.Socket` type, the behavior of Java's `socket` method call and the native `socket` system call are slightly different. Specifically, the Dalvik virtual machine uses a lazy initialization of operating system sockets, where it only issues a `socket` system call when the app either connects or binds to the socket. Hence, a call to the `java.net.Socket` default constructor (i.e., the overloaded constructor without arguments) does not result in a `socket` system call. However, a subsequent call to `connect` or `bind` will automatically issue a `socket` system call before connecting or binding to the socket.

2) IP options: RFC 791 [8] prescribes that IP packet headers can include an optional field called `IP_OPTIONS`, which can contain up to 40 bytes of data, including one byte each for the option's *type* and *length* in bytes. The Linux kernel supports setting these options via the `setsockopt` system call. However, besides a few well-known options (e.g., the `timestamp` option used by the `ping` network utility), the kernel requires administrative privileges (i.e. `CAP_NET_RAW`)

to configure the `IP_OPTIONS` header field. Similar to the socket discussion above, Java also provides a `setOption` API for sockets. Unfortunately, however, this API restricts what values will be passed to the underlying `setsockopt` system call and excludes the value to enable `IP_OPTIONS`.

III. Threat Model and Assumptions

Our threat model is based on a business environment where a company uses BYOD policies that allow employees to use their personal mobile devices to access company owned services and data. To ensure that personal devices do not cause harm to the company's network or assets, personal devices have to be provisioned with a BYOD solution. Similar to the kernel instrumentation of Samsung KNOX [9], these solutions often include vendor-specific Read Only Memory (ROM), therefore vendors can integrate necessary changes for different frameworks. Hence, a production-level system would not need any modifications on user devices other than the provisioning. Following the common practice, work and private applications are separated and root access is disabled in the device by the BYOD framework, which can prevent the rooting process through a hardware-backed chain of trust via Trusted Platform Module and e-fuses [10]. Furthermore, we assume that the enterprise network consists of secured network appliances and previously authenticated devices.

With this business-centric mindset and technology in place, we target a scenario where company-approved applications contain highly desirable functionalities for productivity while also containing features that are detrimental to business interests. This detrimental functionality originates from either (i) developer-authored application functionality that the business does not wish to allow (e.g., file upload), or (ii) third party libraries that are linked into the application (e.g., tracking and advertisement libraries) and violate the business information security policies of the company.

We further assume the operating system on employees' devices is trusted. Similarly, despite the detrimental functionality, our threat model assumes that applications are benign in nature but violate the company's information security policy via app functionalities or packaged libraries. Specifically, we assume that applications do not actively try to circumvent our system. As our system predominantly relies on dynamic analysis, "light" obfuscation, such as the transformations performed by ProGuard, are transparently tolerated. While more advanced obfuscation techniques might thwart our system, we argue that such obfuscation should rarely, if ever, occur in benign applications. Importantly, despite the benign nature of the applications we consider, our system *does not* require access to the application's source code (i.e., the system is compatible with the regular app store distribution model).

IV. System Overview

Our goal is to detect and drop network packets originating from undesired application functionalities (e.g., file uploads and analytics collection) to ensure an execution- and context-aware policy enforcement for BYOD environments. Traditional policy enforcement systems focus on network traffic flow without taking application state into account, which prevents access control with fine-grained rules. Although network administrators can fine-tune an access control scheme for certain cases (e.g. preventing packet flows to a specific IP), the enforcement system should be flexible enough for changing company requirements in terms of policy management and access-control. Additionally, inspecting application state on user devices and enforcing policies at network perimeter requires separate modules. Therefore, communication between such modules should not put extra load on the business network. Specifically, our system design goals are as follows:

Reconfigurability: Providing enterprise system administrators with the flexibility to introduce new applications to the system and inspect specific functionalities of selected apps. This allows IT managers to create fine-grained policies regarding application connections.

App integrity: Compatibility with the existing app store models (i.e., does not require app modifications).

Dynamic context-aware access control: Providing an inspection scheme to monitor smart devices' network connections and distinguish application-specific functionalities to serve as an execution-time context for further inspection.

Fine-grained network policy enforcement: Enforcing fine-grained policies and preventing malicious connection attempts on different enforcement levels to support prohibiting select app functions.

Avoiding extra network load: Using IP headers of device-generated packets for communication between on-device context inspectors and off-device policy enforcers.

In light of these considerations, we design BorderPatrol, which extracts execution-context about established network connections, conveys data by tagging network packets and enforces policies at the business network perimeter.

A. System design

BorderPatrol comprises of four main high-level components for different stages of execution: (i) *Offline Analyzer* provides app specific information for system components and policy creation tools, (ii) *Context Manager* extracts, encodes, and sends the relevant execution-time contextual information from user devices to policy enforcers, (iii) *Policy Enforcer* evaluates contextual data against fine-grained company policies and drops non-conforming packets originating from undesired functions, (iv) *Packet Sanitizer* cleanses the contextual information from policy-conforming packets.

Of all four system components (gold boxes in Figure 1), the *Context Manager* executes on the BYOD-enabled mobile device (gray box in Figure 1) and the remaining three components run on the enterprise's network infrastructure. A general architectural overview of our system with the main components placed at strategic locations within the enterprise network is shown in Figure 1.

1) **Offline Analyzer**: BorderPatrol components generate and interpret the execution-time contextual information in different modules. Therefore, it is mandatory for all modules to

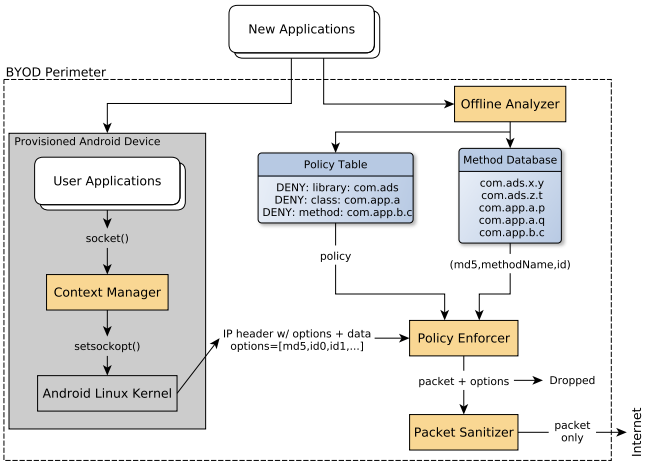

Fig. 1: BORDERPATROL System Architecture.

use the same encoding/decoding method to work in coherence. The purpose of the *Offline Analyzer* is to create a table that *Policy Enforcer* uses for decoding method signatures.

Due to size limitations of `IP_OPTIONS` (§ II-B2), the *Context Manager* transmits contextual information to the *Policy Enforcer* using an index based encoding scheme, where every method signature is deterministically mapped to an index number. *Offline Analyzer* first processes any new app that should be managed by BORDERPATROL and extracts the method signatures. It then organizes method signatures topologically for consistency and assigns sequential indexes to each method. Since the number and class hierarchy of all methods are consistent within the packaged app, the mapping of method signatures to indexes are deterministic in size and ordering. For supporting multiple applications and differentiating methods of same libraries that are in use by different applications, the mappings for different apps are distinguished by the hash of the `apk` in a database.

2) **Context Manager**: BORDERPATROL adds fine-grained contextual information to every network packet that the device sends through the enterprise network. We define the *context* as the Java call stack associated with the socket at the time a connection is established (i.e., a socket is created via the `socket` system call). However the concept of context can be expanded into any other relevant information for the purpose of policy enforcement. As a detrimental activity based on our threat model requires data connectivity to outside the business network perimeter, BORDERPATROL monitors the execution context of the most fundamental network connection object (i.e. socket). Hence, our system requires the following capabilities from the *Context Manager*: (i) monitor the creation of sockets, (ii) gather the call stack when a socket is created, and (iii) add call stack information to packet headers. When an app establishes a network connection, *Context Manager* gathers the call stack and associates stack frames with method signatures. Using method signatures allow a finer-grained policy enforcement and enables BORDERPATROL to distinguish overloaded Java methods within the same library. The *Context Manager* then encodes stack signatures into a compact representation and embeds into `IP_OPTIONS` header field along with an app identifier.

3) **Policy Enforcer**: BORDERPATROL enforces policies on the network and determines whether packets coming from the smart devices in the business network perimeter should be allowed or dropped. The *Policy Enforcer* comprises of three stages: (i) extraction, (ii) decoding and (iii) enforcement. During extraction phase, *Policy Enforcer* extracts the app-identifying hash and the sequence of index numbers that the *Context Manager* has embedded in `IP_OPTIONS`. Then, during decoding stage, it selects the relevant index-to-method signature mapping from the database, indicated by the hash value of the app. *Policy Enforcer* maps each index back to the method signature in the order received, thus creating the stack trace which consists of method signatures. Finally, during enforcement phase, the *Policy Enforcer* uses predetermined policies to decide which network packets violate company policy and drop them accordingly.

4) **Packet Sanitizer**: BORDERPATROL removes contextual information from IP packets before network packets leave the company. Such removal process is necessary as network routers drop IP packets with set options due to network packet filtering specifications such as RFC 7126 [11]. Network hardware providers also recommend dropping these packets to protect appliances from known attack vectors [12]. Therefore, packets which are leaving the company network perimeter should be cleansed of the options that the *Context Manager* injects into IP packets. The *Packet Sanitizer* removes `IP_OPTIONS` from any outgoing IP packets that are in compliance with the company policy. Note that packets violating policy do not reach the *Packet Sanitizer* as the *Policy Enforcer* drops them. Besides ensuring proper routing outside the BYOD perimeter, the *Packet Sanitizer* also provides an important privacy-preserving role by stripping execution-context identifying information (e.g., application names, loaded libraries) from the `IP_OPTIONS`.

B. Policy Specifications

Policies specify the enforcement levels, actions and targets for select app functionalities or the app as a whole. We define policy enforcement action (α) as the decision for a packet, enforcement level (L) as the granularity of inspection into each method signature, and enforcement target (θ) as the unique string which defines a search query for the method signature. With k, c and m denoting library, class, method names in a method signature; h and $s_0 ... s_n$ denoting the app hash and the method signatures of a stack trace in the packet header (H); ℓ_θ denoting the level of target match in a method signature; all possible levels of a target in a stack trace are denoted as ℓ_s. Values of ℓ_s are ordered in accordance to the finer granularity in enforcement such that $\ell_h < \ell_k < \ell_c < \ell_m$. Policy enforcement rules state that; for $s \in H$, $\theta \in s$, $\ell_\theta \in \ell_s$, enforce $\alpha = deny$ if $\exists s$ with $\ell_\theta \geq L \parallel$ enforce $\alpha = allow$ iff $\forall s$ with $\ell_\theta \geq L$. That is, if there is at least one stack signature that contains a match with the search query at the policy level or higher, the packet must be dropped to block an application

functionality (i.e., blacklisting). Alternatively, all the stack signatures should contain a match with the search query at the policy level or higher for a packet to be allowed in the network (i.e., whitelisting). The simplified policy grammar is presented in Snippet 1. Examples 1 through 4 provide sample policy rules at library, class, method and hash enforcement levels respectively.

```
<POLICY> ::= {[<ACTION>] [<LEVEL>] [<TARGET>]}
<ACTION> ::= (allow | deny)
<LEVEL> ::= (hash | library | class | method)
<TARGET> ::= (target_hash | target_library | target_class |
     target_method)

// Example 1: prevent ad library connections
{[deny][library]["com/flurry"]}

// Example 2: prevent functions of an entire class
{[deny][class]["com/google/gms"]}

// Example 3: prevent uploads for Dropbox
{[deny][method]["Lcom/dropbox/android/taskqueue/UploadTask;
->c()Lcom/dropbox/hairball/taskqueue/TaskResult"]}

// Example 4: whitelist company app connections by hash
{[allow][hash]["da6880ab1f9919747d39e2bd895b95a5"]}
```

Snippet 1: Simplified policy grammar.

V. Implementation

This section elaborates on the details of our BorderPatrol prototype. In the spirit of open science and to facilitate reproducible experiments, we will release our implementation of BorderPatrol under an open source license.

A. Offline Analyzer

We implemented the *Offline Analyzer* as a Java program, which accepts a list of `apk` files and produces a database containing mappings for method signatures (§IV-A) in `json` format for its ease of use and portability. During package processing, the *Offline Analyzer* uses the `dexlib2` [13] library to extract method signatures from `dex` files into a sorted list, where the position of the method signature in the list corresponds to the index. The method signatures of a particular app are grouped under the `md5` hash of the `apk`.

B. Context Manager

The *Context Manager* runs on the provisioned device of the user as a user-space program. We implemented *Context Manager* as a module for the Xposed Framework [14], which provides an API for runtime program behavior modification by hooking Java methods and constructors, thus enabling BorderPatrol to monitor the creation of all sockets. When an app is loaded, the *Context Manager* parses the `dex` file using `dexlib2`. The *Context Manager* then generates the mapping of stack signatures to indexes and obtains source line numbers of method signatures. After the app establishes a connection, Xposed *hooks* transfer control to the *Context Manager*, which then gathers the stack trace by invoking Java API's `getStackTrace` method. This method returns a list of active stack frames that the application was executing, each of which corresponds to a method call. The *Context Manager* then uses the source line numbers of each stack frame to associate the method signature of the respective method with an index using the deterministic mapping. We present an example of this process along with the case-study values in Figure 2. For above operations, the *Context Manager* relies on three different submodules:

Hooks: Hooking is a technique to modify application behavior by changing the execution flow in order to alter or augment a function with arbitrary functionality. We use *post-hooks* to `socket` calls for intercepting socket creation, triggering context extraction and `IP_OPTIONS` injection to IP headers. Using *post-hooks* ensure that a socket is present and the connection is established before setting `IP_OPTIONS`. Consequently, the *Context Manager* monitors all connection attempts that are conveyed over all network sockets.

Shared library: BorderPatrol enables the `IP_OPTIONS` field of sockets to tag network packets. However, standard Java API does not allow applications to access this field. As a result, the *Context Manager* requires to execute native code and call the `setsockopt` system call to obtain low level access to socket options via Java Native Interface (JNI). Thus, we compile a shared library which exposes the `setsockopt` system call to the managed Dalvik code via JNI. This library consists of a native function which serves as a wrapper for the `setsockopt` system call.

Instrumented Linux kernel: The default Linux kernel used in Android requires programs to have `CAP_NET_ADMIN` capabilities to construct packet headers, which is exclusive to system applications. Additionally, all non-system applications, such as our *Context Manager*, run in user-space without such privileges and cannot set socket options even in the native space. To overcome such restrictions, we instrument the Linux kernel with a one-line patch such that it allows IP header construction regardless of the privilege level of an application.

C. Policy Enforcer

We implemented *Policy Enforcer* as a user-space program which uses Python's `netfilterqueue` bindings [15] to receive incoming network packets and Scapy network packet processing package [16] to detect and extract the sequence of indexes from IP headers. The *Policy Enforcer* decodes an index to a method signature by using the `json` database that comes from the *Offline Analyzer*, where each index corresponds to the position of a method signature in the list of method signatures. To enforce a policy, the *Policy Enforcer* engine checks policy rules to determine the required course of action according to policy specifications (§IV-B). If *Policy Enforcer* does not detect a policy violation, it allows corresponding IP packets to continue their route first to the *Packet Sanitizer* module and then to their original destination.

D. Packet Sanitizer

Similar to the *Context Manager*, the *Packet Sanitizer* module also employs `netfilterqueue` bindings and the Scapy package to acquire and modify incoming network packets from the *Policy Enforcer*. The *Packet Sanitizer* removes

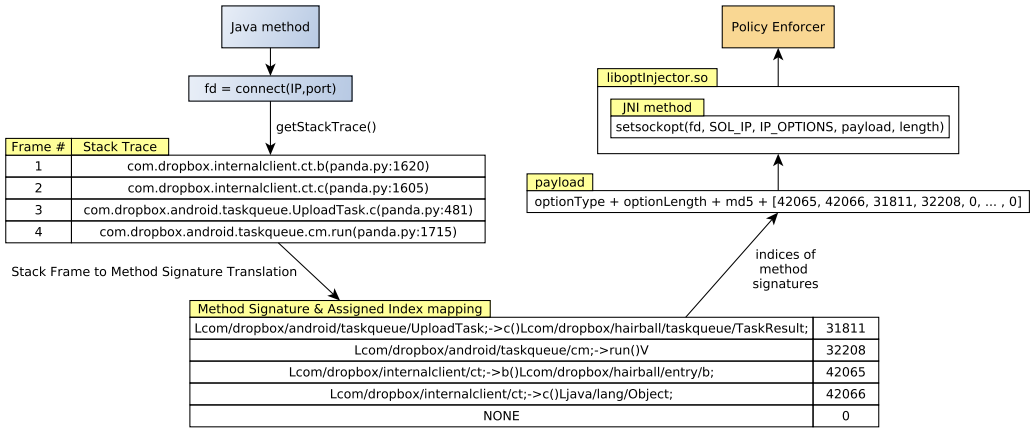

Fig. 2: *Context Manager* work flow. The *Context Manager* obtains the stack trace after a connection is established via `getStackTrace` method. Stack frames include source line numbers that allow matching method names to method signatures. Using sequentially ordered list of method signatures from the `dex` file, each method signature is encoded into an index number. Finally, indexes are injected into IP headers using `setsockopt` system call through our shared library.

`IP_OPTIONS` from the packet header when it detects that the respective field is enabled.

E. Policy Extractor

As an extension to the BORDERPATROL architecture, we also provide a Python analysis tool to assist IT administrators in determining policies. This tool runs an application twice. In the first run, administrators can exercise the app for allowed functionalities, which BORDERPATROL uses to construct a baseline profile. On the second run, human operators are guided to invoke undesirable functionalities in the app. The tool then automatically identifies uniquely appearing method signatures in stack traces and maps them to the set of targeted functions per each run. Subsequently, the *Policy Extractor* parses each unique method signature and constructs policies with specified levels of enforcement.

VI. EVALUATION

A. Experimental setup

We implemented our prototype of BORDERPATROL for Android 7.1.1 Nougat (API 25) and evaluated the system in the Android emulator. A job dispatcher node assigns apps to evaluate to a worker node via RabbitMQ message broker [17]. The worker runs an instance of the Android emulator on QEMU [18] with a modified Android system image with Xposed framework for the x86 architecture and the patched Linux kernel v3.10. The QEMU emulator uses `TAP` virtual network interface for network connectivity. We also use `iptables` to route packets originating from the emulator into `netfilter` queues. During testing, we use the `adb monkey` User Interface exerciser [19] to provide random UI inputs to apps. Finally, the *Packet Sanitizer* module removes `IP_OPTIONS` from outgoing packets to ensure that the packets get routed on the Internet correctly.

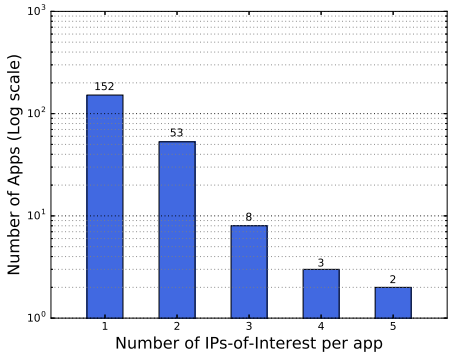

Fig. 3: Number of apps that have different stack traces connecting to the same IP addresses.

B. Analysis

BORDERPATROL is most useful in situations where (i) an app contains a mix of desirable as well as non-desirable functionalities, and (ii) these functionalities cannot be distinguished by existing network-level enforcement mechanisms. Thus, in this section we present an analysis and estimate the prevalence of apps that fulfill these criteria. To this end, we evaluate BORDERPATROL with apps from the PlayDrone [20] dataset.

As BORDERPATROL aims at a BYOD deployment, we chose the 1,000 most popular (i.e., most downloaded) apps in each BUSINESS and PRODUCTIVITY categories for a total of 2,000 apps. We then exercise each app with `adb monkey` and issue 5,000 random events while recording all generated network traffic during this experiment. As discussed above, a network-based enforcement mechanism can easily distinguish traffic based on DNS names or IP addresses. Therefore, we conservatively assume that BORDERPATROL will be most useful if different functionalities within an app connect to the same IP address. Hence, we define an IP-of-interest (i.e., *IoI*) as an IP address that is the destination of multiple IP packets which contain more than one distinct stack trace.

As the calling context is established at the time a socket is created, all packets of a given connection will contain the same stack trace. Thus, stack traces will only differ for packets in connections established at different contexts within the app.

Figure 3 shows the number of apps that connect to one or more *IoI*s. Note that these numbers are extracted from the dynamic analysis described above. As it is unlikely that our `monkey`-based analysis achieves complete code-coverage, the data presented in Figure 3 is a lower bound on the apps and their *IoI*s, and BORDERPATROL might be applicable broader than the figure suggests.

Based on these results, we observe that a total of 218 apps in our experiment had at least one *IoI*. In 75% of the applications with an *IoI*, the methods in the stack traces originate from the same Java package. This corresponds to the case studies presented in §VI-C, where the desirable and undesirable functionality is contained in a third party library (i.e., the same Java package in the Facebook SDK) or both belong to the core-functionality of the app (i.e., the app's main Java package for Box and Dropbox). Interestingly, 25% of the *IoI*s receive traffic that contains stack traces with methods from different Java packages. This will happen if different components within an app reuse a shared common popular library. For example, the Apache HTTP client library [21] frequently occurs here. An advanced network-based enforcement mechanism might try to fingerprint network traffic based on such predictable behavior by the network library in use. However, the reuse of a network library by different app components (as discussed here), would thwart any such fingerprinting attempts. This evaluation on the prevalence and structure of *IoI*s illustrates that BYOD policy enforcement mechanisms would greatly benefit from the fine-grained contextual information that BORDERPATROL provides.

1) Validation: In this section we evaluate whether BORDERPATROL is precise enough to only disable unwanted functionality but leave the remainder of the app intact. Unfortunately, whether a given functionality is beneficial or detrimental to a company is not a global property. That is, some functionality (e.g., "Login with Facebook") might be beneficial for one company but deemed detrimental by another. Hence, for this evaluation, we rely on data collected by Li et al. [22] to determine an unwanted functionality. In their work, Li et al. identified a set of 1,050 third party libraries that exfiltrate sensitive information including a variety of popular analytics and advertisement libraries. Based on these findings, we created a simple policy that drops all network packets that contain stack traces that are associated with any of these libraries. (e.g., `com.flurry` library, Example 1 in Snippet 1) Subsequently, to assess the impact of this policy on app usability, we chose a set of 60 apps and manually evaluated them by sorting the libraries that manifest themselves in *IoI*'s according to their popularity in our app sample of 2,000 apps. We then traverse this list and for each library, chose one app that includes the corresponding library. Finally, we arrive at a data-set of 60 apps that in union include the 60 most popular libraries. To assess the impact created by our policy on these apps, we manually run each app twice — once as a baseline with BORDERPATROL disabled, and once with BORDERPATROL enforcing the above-stated policy. The task of the human evaluator at this point is to distinguish any changes in behavior between the two runs.

As Li's list contains a set of advertisement libraries, one of the obvious differences observed repeatedly was the lack of ads displayed when BORDERPATROL was in effect. Li's list also contains a wealth of analytics and tracking libraries. We verified that BORDERPATROL correctly dropped all network traffic generated by the flagged libraries by inspecting the network traffic before and after the *Policy Enforcer*. Blocking analytics and tracking libraries did not result in any observable differences in any of the apps. In summary, BORDERPATROL correctly enforced the stated policy, prevented the transmission of sensitive information, and did so without negatively affecting app functionality.

C. Case studies

Existing network-based BYOD enforcement mechanisms operate on coarse-grained context information pertaining to the network traffic. The lack of fine-grained context implies that these systems cannot provide a variety of advantageous business cases. More precisely, if a given app contains a mix of useful and detrimental functionalities for the company, additional context can be used by a BYOD deployment to allow the former but prevent the latter.

Our case studies focus on two such scenarios. First, cloud storage apps provide a convenient way to share company data among employees, but at the same time allow employees to upload documents that might violate company policy or the law (e.g., the GDPR or HIPPA). To demonstrate the utility of a BYOD policy that distinguishes upload and any other operations, we present this use-case with the Dropbox and Box cloud storage apps. The second use-case involves apps that rely on an identity provider (e.g., Facebook) for authentication purposes, but at the same time transmit analytics information. The prototypical example in this category is Facebook's SDK which provides access to the Facebook Graph API [23]. This API implements functionality for identity provider capabilities (e.g., "Login with Facebook") as well as functionality that app developers can use to collect usage statistics and implement user tracking. We demonstrate this use-case on the SolCalendar [24] app. To illustrate the utility of fine-grained contextual information for a BYOD deployment, we compare, for both use-cases, a conventional network enforcement approach with the capabilities provided by BORDERPATROL.

Cloud storage: Dropbox [6] and Box [25] are popular cloud-based file synchronization apps available on the Google PlayStore from BUSINESS and PRODUCTIVITY categories, featuring more than 500M and 10M downloads, respectively.

On-network enforcement: In this scenario the policy enforcement mechanism is implemented exclusively on the network, and can allow or reject traffic based on IP addresses, DNS names, packet flow direction and size, or any other information available on the network layer (we refer the reader to §VIII for more details about on-network enforcement). For

our purposes, we record the network traffic generated when using the Dropbox and Box apps to download and upload content, at first without enforcement. Dropbox uses the same DNS names and IP addresses to upload and download content. As such, a network-based enforcement mechanism can only block both or neither of these functionalities, but cannot establish the use-case where BYOD provisioned devices can download documents but not upload (or leak) other data. On first glance, the situation with Box seems easier to handle for a network-based mechanism. Specifically, Box uses different IP addresses for the download and upload functionality. However, merely blocking the IP address that is used to upload data also prevents the listing and browsing of documents, and hence effectively thwarts the download capability too, as users cannot discover the files they might want to download. Additionally, preventing outgoing packet flows that exceed a certain size fails to prevent uploads where file size is below the threshold.

BORDERPATROL: In our approach, we first use BORDERPATROL to profile apps and generate the `json` database. We then use the policy maker to determine which methods uniquely appear in the `IP_OPTIONS` when we upload and download documents in the Dropbox and Box apps. Based on this profiling information, the system creates a policy that drops packets which include method signatures that are only present in the connections used to upload content. Specifically, the policy configuration causes BORDERPATROL to drop packets originating from Dropbox, if the stack trace includes a specific method from the `UploadTask` class. (Example 3 in Snippet 1). Similarly, BORDERPATROL drops packets that originate from Box if the stack trace includes a specific method in the `BoxRequestUpload` class.With this policy in place, we exercised both apps manually by traversing through all available menu items, listing, searching, previewing and downloading a previously uploaded image which is not present on the device. We then downloaded another image from Google Images which is not present in either of the cloud storages and attempted to upload this image to both accounts. We observed that beyond the blocked upload functionality, all other app capabilities remain intact.

Facebook SDK analytics and login: For analytics activity, we examine a calendar app called SolCalendar which uses the Facebook Graph API to provide authentication and report back analytics information. As discussed in §I, the transmission of analytics information can be detrimental to a company's business interests. Hence to assess the capability of a BYOD enforcement mechanism to allow authentication and prevent analytics, we again compare the two different strategies.

On-network enforcement: In this scenario, we first set a policy to drop all packets whose destination IP corresponds to a Facebook Graph API DNS name. We then run the calendar app and immediately observe (unsurprisingly) that the "Login with Facebook" functionality is broken. While the above restriction obviously prevents analytics data from being transmitted to Facebook, it also thwarts the useful authentication functionality. This example illustrates that fine-grained contextual information is necessary to enable BYOD policies that maintain beneficial app functions.

BORDERPATROL: The contextual information provided by BORDERPATROL is sufficient to distinguish between the authentication and analytics work-flows. We use BORDERPATROL and leverage a simple policy to block undesirable analytics activities by dropping the packets which include any of the identified method signatures. During the manual evaluation at the time of policy enforcement, we observe that BORDERPATROL preserves the "Login with Facebook" functionality. Furthermore the policy enforcement does not lead to any observable changes in the app. We also verified, by inspecting the network traffic, that our enforcement mechanism correctly drops the network packets used for analytics.

Takeaway: The above case studies illustrate how BYOD enforcement mechanisms that rely exclusively on a network-viewpoint lack the fine-grained contextual information to enforce policies that are beneficial to the company. However, this shortcoming can be rectified with a system like BORDERPATROL that augments network information with fine-grained contextual information within the BYOD perimeter.

D. Performance Evaluation

We evaluated our prototype implementation of BORDERPATROL on a quad core 3.20GHz Intel ® Core ™ i5-4570 CPU and 24GB of RAM. We performed the Android experiments on emulators with modifications as described in §V. We implemented a network stress test app that repeatedly (for 10,000 iterations) creates a socket, sends a single `HTTP GET` request for a static 297-byte `HTML` page to a server, and closes the socket again as fast as the device allows, therefore representing the worst case scenario for a device's network stack. To avoid network-induced latency fluctuations, we hosted a Python SimpleHTTPServer [26] on the same host that runs the Android emulator. The goal of this performance evaluation is to measure the overhead of every component and modification we used to realize BORDERPATROL by adding one component after another to the default emulator (i.e., baseline) until we obtain the full BORDERPATROL system. These configurations are as follows:

(i) *default-SLIRP (baseline):* This configuration corresponds to an Android emulator as defined by the Android SDK, which uses an unmodified system image, default Android Linux kernel and QEMU's user-mode (SLIRP) networking stack for connection.

(ii) *default-tap:* This configuration modifies the networking setup of the baseline and uses virtual TAP interface, allowing us to measure the performance difference between SLIRP and TAP networking modes.

(iii) *default-tap-nfqueue:* This configuration introduces `iptables` rules to redirect network traffic into an `NFQUEUE`, which is then consumed by a simple Python program that reads all packets and injects them back unmodified. Such setup corresponds to a situation where BORDERPATROL enforces an empty (or allow-all) policy and shows the minimum performance impact that the Python-based *Policy Enforcer* introduces.

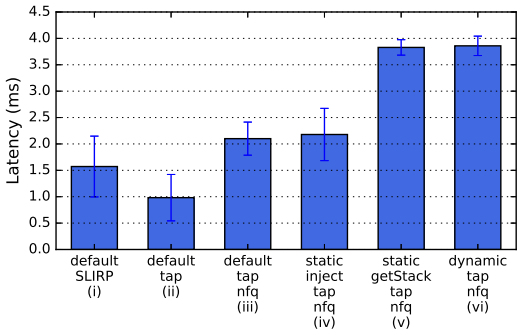

Fig. 4: Average latency of an `HTTP GET` request to a local server across different Android image modifications, network interface and routing configurations.

(iv) *static-inject-tap-nfqueue:* Here, we introduce the instrumented Linux kernel (to enable arbitrary `IP_OPTIONS`) and the system image (to include Xposed). However, instead of using the Xposed module from BORDERPATROL, we use a simple module that sets a static string of characters as `IP_OPTIONS` for each created socket. This setup illustrates the performance impact induced by Xposed when hooking the socket functionality as described in §V.

(v) *static-getStack-tap-nfqueue:* In this configuration we use the same setup as iv. However, hooked functions also make a `getStackTrace` call to obtain a stack trace, which allows us to determine the performance impact of gathering stack trace elements via provided Java API.

(vi) *dynamic-tap-nfqueue:* This configuration corresponds to the full prototype of our BORDERPATROL implementation. In addition to the previous setup, this configuration adds the Xposed module which extracts call stacks when sockets get created and encodes the corresponding information dynamically into the sockets' `IP_OPTIONS`.

To measure the overheads introduced by each component we run our stress-test app for 25 times on each configuration, and show the average time per `HTTP` request in Figure 4. The only overheads worth mentioning are those introduced by the Python-based *Policy Enforcer* (i.e., ii–iii, +1ms) and the `getStackTrace` API call that obtains stack traces (i.e., iv–v, +1.6ms). While the relative overhead (i.e., 2x) seems significant, we note that the absolute overhead of less than 2.5ms is negligible compared to often hundreds of ms network latencies induced by networking equipment over intercontinental distances. Furthermore, BORDERPATROL only performs the most performance sensitive operation of obtaining and encoding stack traces once per socket, and this 2.5ms overhead will thus amortize over the lifetime of the socket (e.g., a socket that is configured as keep-alive serves multiple HTTP requests and responses during its lifetime).

VII. DISCUSSION AND LIMITATIONS

BORDERPATROL implements secure policy enforcement through different policy actions (i.e., blacklisting and whitelisting) and its operating principles: (i) By enforcing policies on the network and minimizing the trusted computing base on user devices, administrators can use BORDERPATROL to ensure that users cannot side-step the policy enforcement mechanism, or tamper with company-determined policies. (ii) During its operation, BORDERPATROL can determine which libraries are in-use for network connectivity of an app by mapping stack frames to method signatures. This feature allows administrators to use a blacklist of libraries and restrict apps from using these libraries to establish network connections (e.g., disallowing tracker and analytics libraries to prevent privacy leaks or prohibit connections via previously-known vulnerable libraries [27]). (iii) BORDERPATROL allows administrators to vet and whitelist *only* the desired functionalities of an app and disallow any other unknown app operations. A whitelisting approach inhibits users from engaging with the app in an unintended way (e.g., file uploads via the chat window of a word processor instead of using the upload button) within the constraints defined by policies. Furthermore, whitelisting prevents socket connections which originate from malicious methods in accidentally-installed repackaged apps, as such functions are not vetted by the administrators.

Related works have delegated the functions of *Policy Enforcer* and *Packet Sanitizer* modules to the device [28], [29]. However, there are a number of reasons why we argue that placing these modules on the network and only making minimal changes to the device is beneficial in a BYOD scenario:

Security: A robust system-wide security mechanism should have a dependable method of conforming with the principles of *complete mediation* [30]. BORDERPATROL achieves this through enforcing policies on *all* packets in the business network perimeter. Enterprise network rules can be configured such that access to the company resources is limited to local network or VPN connections. Since BYOD frameworks can force the packets of work profile applications to go over VPN connections [31], all packets that leave the work profile are subject to BORDERPATROL's policy enforcement.

Ease of use: Traditional security solutions are difficult and inflexible to program, deploy and manage for BYOD scenarios [32]. SDN infrastructure requires network equipment to be SDN-enabled, which is still insufficient to enable fine-grained app control. In comparison, BORDERPATROL can extract detailed contextual data on user devices and enforce policies at the business network perimeter using commodity hardware. By enforcing policies at a centrally managed location in the network, administrators can configure and update all their policies in one spot. Furthermore, since the contextual data extraction happens at the application level, BORDERPATROL's operations are not hindered by changes in Android versions or the underlying hardware structure, therefore making BORDERPATROL compatible with various devices and OS versions. However, as different versions of apps use different sets of methods, BORDERPATROL requires administrators to use the policy extractor tool on updated versions of apps which are in use by the enterprise.

Compatibility: In a provisioned device, work related apps run inside a work profile. The separation of profiles ensures

that BORDERPATROL tags all packets which originate from work related apps and does not interact with apps that fall outside the context of business use. If, however, the user uses their work profile outside the enterprise network premises, tagged packets will be subject to policy enforcement through VPN, while non-work related apps' background network activity is routed through mobile networks. Similarly, if the user does not use the work profile while in the enterprise network, the *Policy Enforcer* will drop packets that do not contain `IP_OPTIONS`, thus ensuring that all packets that are leaving the business network perimeter are originating from sockets which BORDERPATROL controls.

Additionally, BORDERPATROL's enforcement starts from the very first outgoing packet. For instance, in traditional network filtering appliances, it is possible to differentiate uploads from downloads based on measuring outgoing continuous data transfers in a single flow by setting a data transfer size limit as a triggering mechanism. However, using multiple sockets and fragmenting outgoing data would overcome such precautions. Our empirical analysis shows that a legitimate request in a single flow can range from 36 bytes to 480MB, which complicates policy settings for the purpose of setting a threshold. Unlike the traditional approach, BORDERPATROL detects upload attempts irrespective of the data transfer size.

a) Limitations: As our goal in this paper is to demonstrate a proof of concept for augmenting network packets with contextual data and fine-grained policy enforcement, our prototype is subject to several implementation-specific drawbacks. As these drawbacks did not manifest any issues in BORDERPATROL prototype during our evaluations, we do not believe that these drawbacks detract from the contribution put forth by BORDERPATROL. Furthermore limitations of our prototype can be averted by different engineering choices.

Hash collision: Our system identifies the origin of each packet by a truncated (8-byte) hash value of the respective app's `apk` file. As the number of bits in a hash value decreases, the probability of hash collision increases. With existing 3.3M apps in Google Play Store [33], the probability of collision is lower than 10^{-6}, which is reasonable for practical solutions.

Tag-replay: The patch we introduced to Linux kernel as part of the BORDERPATROL prototype permits user-space programs to set `IP_OPTIONS` of *security* type. This allows an app to first use a benign functionality to send packets outside the corporate network perimeter, and then copy the same `IP_OPTIONS` to the socket that a malicious function has initiated. Such behaviors can be thwarted by modifying the Linux kernel so that `setsockopt` system call can only set `IP_OPTIONS` on a socket once. This ensures that other apps cannot alter `IP_OPTIONS` after *Context Manager*.

Multi-dex file applications: For apps which include more than 65,536 methods, the `apk` file packs more than one `dex` file due to Dalvik specifications and 2 bytes per stack frame cannot support apps that have multiple `dex` files. A way of overcoming this limitation is to use a variable length encoding with a single bit to indicate 2 or 3 byte lengths. The length of a stack frame can be scaled up to 3 bytes if the `apk` packs multiple `dex` files to provide coverage for large apps.

Socket reuse: BORDERPATROL encodes the same stack trace which belongs to a socket on all the packets that the app sends over the same socket. Hence, if an app reuses a socket for a different purpose *before* terminating the connection, BORDERPATROL might not be able to attribute individual packets in the same connection to the new context. Note that an app cannot change the endpoint of a socket if it reuses the socket, either. To change the endpoint, the app would have to call `connect` again, which in turn would be correctly handled by BORDERPATROL.

Overloaded methods: As the Java API only provides method names in stack traces, BORDERPATROL relies on line numbers to disambiguate overloaded variants of methods with the same name within one class. However, developers can choose to strip line numbers and other debug information from their apps. While stripped debug information would force BORDERPATROL to over-approximate context (i.e., merge all overloaded variants of methods with the same name in the same class into one identifier), the precision of the context would only reduce to a method name. Furthermore, we observed that in our dataset there were no apps that have overloaded methods and debug information stripped at the same time. Hence, we postulate that for benign apps (as per our threat model) this should not be a significant problem.

Android image: In the Android security framework, apps fork from a parent process called Zygote and run in separate sandboxes as non-privileged users. This clear separation prevents other user-space programs (like *Context Manager*) from monitoring app context from outside the sandbox. In recent literature, we observed several methods to overcome this prevention mechanism for system prototyping, such as: i) rooting the device for hooking into Android and Java API (Xposed) [34], [35], ii) modifying the default Zygote behavior [36], [37], iii) relying on customized system image distributions from hardware vendors [32] and iv) using altered versions of an app [38], [39]. We chose to use Xposed for our implementation purposes to demonstrate the applicability of our idea. In a production level implementation of BORDERPATROL, hardware vendors can provide custom images for BYOD services for supported devices [40], [41], thus incorporating required access controls in the image.

Native functions: Due to its functioning mechanism, Xposed does not support hooking native functions or direct system calls. Hence, our prototype of BORDERPATROL does not handle apps that call the `socket` APIs in `libc` or issue system calls directly from a native component. However, this drawback could be rectified by using a hooking system that supports native code (e.g., Frida [42]), or by implementing the Xposed module's functionality in native code.

VIII. RELATED WORK

Recent works have shown that mobile apps increasingly collect personal and identifiable information [35], [43]. To address this threat, a large body of related work have proposed solu-

tions to enforce corporate BYOD policies. Existing solutions can be classified under two different layers of enforcement:

On-device enforcement: Existing policy administration frameworks provide separation between work and personal data through containerized profiles. While these systems provide the ability to incorporate BYOD solutions into existing business network infrastructure, they are limited in enforcing fine grained context-aware policies. ADM [2] is a remote device management framework for companies to provision devices, control and enforce policies on Android devices. ADM provides the capability to log DNS lookups and TCP connections where IP addresses, ports package names and respective timestamps can be recorded [44]. However, it cannot inspect application context or packets that belong to different sockets, and limited in capacity due to their dependence of the provided ADM SDK. Samsung KNOX [45] provides a more advanced network analysis feature with "Network Platform Analytics", where compatible a network appliances examine detailed information such as PID of the application which originated the network flow. However, unlike our implementation, this approach lacks context of the established connection.

Conti et al. [28] apply a fine-grained policy enforcement for Android smart phones with a system called CRePE and modify the Android framework to introduce a runtime checker that enforces different context-related policies. CRePE can restrict the set of applications authorized to run, however, unlike our system, CRePE cannot restrict access to only certain libraries within an app (i.e., app-level granularity). Zhan et al.'s [46] propose inserting an in-line reference monitor within the application, which requires apps to be modified prior to installation (using repackaging). Contrarily, BORDERPATROL works on unmodified apps. Pearce et al. [29] present a privilege separation framework called AdDroid and introduce a new advertisement API to limit the scope of such libraries. Although Nativeguard [47] is not constrained by the type of libraries and provides a library-level enforcement with app modification, it cannot provide fine-grained policies at the level of distinct functionalities that are tied to specific methods. Our work offers expressive policies and can deal with cases where a single library is used for legitimate and illegitimate purposes at the same time. Other recent works have proposed new systems that extend the granularity of previous approaches [48], [49]. A number of techniques require OS support and solve the problem on privilege and permission levels. Adsplit [50] isolates ad processes from user activities with distinct UID to serve different permissions to the processes, Aframe [51] isolates processes with iframe displays and compartmentalizes ad permissions and Swirls [52] focuses on data protection through encapsulation across different app contexts.

On-network enforcement: Early efforts aiming at enforcing policies on the network rely on solutions that are transparent to both the protected and remote endpoints [53]. Later works propose to use mechanisms for tagging data as it flows through the network stack [54]. This allows to add more semantics to the packages originating the communication and enable more complex enforcement on the network. Dymo [55] injects a process' identity label to network packets for enforcing which software packages to be permitted on users' machines. Hond et al. [32] proposed a SDN-based programmable BYOD system that can provide app-specific policy enforcement on the enterprise network. One major problem common to all current works performing on-network enforcement is that they lack on expressiveness to build fine-grained enforcement systems, which relates to the granularity of the information embedded in the network package (i.e., embedding either the process or app identifier). As opposed to our work, we embed contextual information from the software component operating the network connection. For finer granularity in inspection phase, Shebaro's work [56] inspects uses device location as context, however it revokes/grants app permissions as a policy enforcement mechanism and lacks enforcement on app functions. While Hong et al. [32] provide larger expressiveness than previous works, their approach requires important changes in the network architecture of a corporation such as extending already existing SDN frameworks. Similarly, Poise [57] also rely on SDN controllers while inspecting context for network-enforcement while periodically broadcasting context. Our work addresses these issues by augmenting the resulting network traffic with key contextual information on every packet header, which is used to build expressive policies that are enforced at the corporate network level without relying on SDN. Backes et al. [58] leverages the well structured class hierarchy and method signatures of ad libraries for fingerprinting, but modifies the class structures during ad blocking efforts. PrivacyGuard [59] uses a localized VPN-based platform to intercept network traffic of apps and filter them based on taint analysis of data leaking apps.

Previous studies have also characterized applications' behavior on requesting dangerous permissions, accessing and sending sensitive information over the network [60] and identified a set of commonly used libraries with keyword matching and ad component detection [22], which provided us with the insights to determine high level policies in BORDERPATROL.

IX. CONCLUSIONS

In this paper, we presented a novel fine-grained policy enforcement system for Bring Your Own Device (BYOD) enabled corporate networks. Our approach distinguishes itself with the feature of blocking packets originating from undesirable application functionalities while leaving remaining functions operational. We then built BORDERPATROL, a prototype system that implements this approach, presented realistic use cases in a BYOD context, and analyzed 2,000 apps from Google PlayStore. Finally, we evaluated the performance overhead of our prototype. Our results show that BORDERPATROL is effective in enforcing policies with negligible overhead.

ACKNOWLEDGEMENTS

This work was partially funded by ONR under grants N00014-17-1-2541 and N00014-17-1-2011. We would like to thank the anonymous reviewers and our shepherd Kaustubh Joshi for their insightful feedback and help in improving the final version of our paper.

REFERENCES

[1] Samsung, "Samsung Knox for Android." https://www.samsungknox.com/en/knox-features/android, 2018.

[2] Google, "Android Device Management for Enterprises." https://www.android.com/enterprise/management/, 2018.

[3] J. Chen, "Apple tablet prototypes possibly identified by web analytics, running iphone-like os 3.2." https://gizmodo.com/5456004/apple-tablet-prototypes-possibly-identified-by-web-analytics\\-running-iphone-like-os-32, 2010.

[4] Yahoo, "Flurry Analytics." https://developer.yahoo.com/flurry/, 2018.

[5] Dataviz, "Docs To Go Android App." http://www.dataviz.com/dtg-android, 2018.

[6] Dropbox, "Dropbox Inc. App." https://play.google.com/store/apps/details?id=com.dropbox.android, 2018.

[7] EU, "General Data Protection Regulation." https://ec.europa.eu/info/law/law-topic/data-protection/reform/rules-business-and-organisations_en, 2018.

[8] IETF, "RFC 791: Internet Program Protocol Specification." https://tools.ietf.org/html/rfc791, 1981.

[9] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 90–102, ACM, 2014.

[10] Samsung, "KNOX Root of Trust." https://kp-cdn.samsungknox.com/bb91024cad9080904523821f727b9593.pdf, 2016.

[11] IETF, "RFC 7126: Recommendations on Filtering of IPv4 Packets Containing IPv4 Options." https://tools.ietf.org/html/rfc7126, 2014.

[12] JuniperNetworks, "Understanding Network Reconnaissance Using IP Options." https://www.juniper.net/documentation/en_US/junos/topics/concept/reconnaissance-deterrence-network-reconnaissance-understanding.html, 2017.

[13] B. Gruver, "dexlib2 library." https://github.com/JesusFreke/smali/tree/master/dexlib2, 2017.

[14] rovo89, "Xposed Framework API." http://api.xposed.info/reference/packages.html, 2017.

[15] PyPI, "NetfilterQueue." https://pypi.python.org/pypi/NetfilterQueue, 2017.

[16] SecDev, "Scapy." http://www.secdev.org/projects/scapy/, 2017.

[17] RabbitMQ, "RabbitMQ Messaging Protocol." https://www.rabbitmq.com/, 2018.

[18] QEMU, "QEMU." https://www.qemu.org/, 2018.

[19] Android, "ADB Monkey UI Exerciser." https://developer.android.com/studio/test/monkey.html, 2017.

[20] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of Google Play," vol. 42, no. 1, pp. 221–233, 2014.

[21] Apache, "Apache HTTP client library `org.apache.http.client`." https://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/, 2018.

[22] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "An investigation into the use of common libraries in Android apps," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1, pp. 403–414, IEEE, 2016.

[23] Facebook, "Graph API." https://developers.facebook.com/docs/graph-api/, 2018.

[24] SolCalendar, "SolCalendar app." https://play.google.com/store/apps/details?id=net.daum.android.solcalendar, 2016.

[25] Box, "Box App." https://play.google.com/store/apps/details?id=com.box.android, 2018.

[26] SimpleHTTPServer, "Python Simple HTTP Server package." https://docs.python.org/2/library/simplehttpserver.html, 2018.

[27] NIST, "CVE-2016-2402: OkHttp certificate pinner bypass." https://nvd.nist.gov/vuln/detail/CVE-2016-2402, 2016.

[28] M. Conti, V. T. N. Nguyen, and B. Crispo, "Crepe: Context-related policy enforcement for android.," in *ISC*, vol. 10, pp. 331–345, Springer, 2010.

[29] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in Android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pp. 71–72, Acm, 2012.

[30] US-CERT, "Complete Mediation." https://www.us-cert.gov/bsi/articles/knowledge/principles/complete-mediation, 2005.

[31] Samsung, "Samsung KNOX container-wide VPN." https://docs.samsungknox.com/knox-platform-for-enterprise/admin-guide/knox-workspace-vpn.htm, 2018.

[32] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu, "Towards SDN-defined programmable BYOD (Bring Your Own Device) security.," in *Network and Distributed System Security Symposium*, 2016.

[33] Statista, "Number of available applications in the Google Play Store." https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, 2018.

[34] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, pp. 1–16, 2017.

[35] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez, "Bug fixes, improvements,... and privacy leaks," in *Network and Distributed System Security Symposium*, 2018.

[36] G. Russello, A. B. Jimenez, H. Naderi, and W. van der Mark, "Firedroid: Hardening security in almost-stock Android," in *Proceedings of the 29th Annual Computer Security Applications Conference*, pp. 319–328, ACM, 2013.

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

[38] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, "Njas: Sandboxing unmodified applications in non-rooted devices running stock android," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pp. 27–38, ACM, 2015.

[39] M. Zhang and H. Yin, "Efficient, context-aware privacy leakage confinement for Android applications without firmware modding," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pp. 259–270, ACM, 2014.

[40] Samsung, "KNOX supported devices." https://www.samsungknox.com/en/knox-platform/supported-devices, 2018.

[41] SEAP, "Android custom ROM compatibility." https://seap.samsung.com/forum-topic/once-i-install-knox-customization-rom-\\-it-possible-upgrade-newer-versions-android-ota, 2015.

[42] Frida, "Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.." https://www.frida.re/, 2018.

[43] V. F. Taylor and I. Martinovic, "To update or not to update: Insights from a two-year study of Android app evolution," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 45–57, ACM, 2017.

[44] Google, "ADM Logging." https://developer.android.com/work/dpc/logging, 2018.

[45] Samsung, "KNOX: Network platform analytics." https://seap.samsung.com/html-docs/android/Content/network-platform-analytics-reference-isv.htm, 2018.

[46] R. Xu, H. Saïdi, and R. J. Anderson, "Aurasium: Practical policy enforcement for android applications.," in *USENIX Security Symposium*, vol. 2012, 2012.

[47] M. Sun and G. Tan, "Nativeguard: Protecting android applications from third-party native libraries," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pp. 165–176, ACM, 2014.

[48] J. Zhan, Q. Zhou, X. Gu, Y. Wang, and Y. Niu, "Splitting third-party libraries privileges from android apps," in *Australasian Conference on Information Security and Privacy*, pp. 80–94, Springer, 2017.

[49] F. Wang, Y. Zhang, K. Wang, P. Liu, and W. Wang, "Stay in your cage! a sound sandbox for third-party libraries on android," in *European Symposium on Research in Computer Security*, pp. 458–476, Springer, 2016.

[50] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications.," in *USENIX Security Symposium*, vol. 2012, 2012.

[51] X. Zhang, A. Ahlawat, and W. Du, "Aframe: Isolating advertisements from mobile applications in android," in *Proceedings of the 29th Annual Computer Security Applications Conference*, pp. 9–18, ACM, 2013.

[52] G. Salles-Loustau, L. Garcia, K. Joshi, and S. Zonouz, "Don't just BYOD, bring-your-own-app too! Protection via virtual micro security perimeters," in *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pp. 526–537, IEEE, 2016.

[53] A. D. Keromytis and J. L. Wright, "Transparent network security policy enforcement.," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 215–226, 2000.
[54] A. D. Keromytis, "Tagging data in the network stack: mbuf_tags," 2003.
[55] B. Gilbert, R. Kemmerer, C. Kruegel, and G. Vigna, "Dymo: Tracking dynamic code identity," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 21–40, Springer, 2011.
[56] B. Shebaro, O. Oluwatimi, and E. Bertino, "Context-based access control systems for mobile devices," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 2, pp. 150–163, 2015.
[57] A. Morrison, L. Xue, A. Chen, and X. Luo, "Enforcing context-aware BYOD policies with in-network security," in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, 2018.
[58] "Seamless in-app ad blocking on stock A,"
[59] Y. Song and U. Hengartner, "Privacyguard: A VPN-based platform to detect information leakage on Android devices," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pp. 15–26, ACM, 2015.
[60] AppCensus, "AppCensus Analysis Database." https://appcensus.mobi/, 2017.