What's in Your Dongle and Bank Account? Mandatory and Discretionary Protection of Android External Resources

Soteris Demetriou[†*] , Xiaoyong Zhou[‡*],
Muhammad Naveed[†], Yeonjoon Lee[‡], Kan Yuan[‡], XiaoFeng Wang[‡], Carl A Gunter[†]
[†]Department of Computer Science, University of Illinois at Urbana-Champaign
[‡]School of Informatics and Computing, Indiana University, Bloomington
{sdemetr2, naveed2, cgunter}@illinois.edu, {zhou, yl52, kanyuan, xw7}@indiana.edu

***Abstract*—The pervasiveness of security-critical external resources (e.g accessories, online services) poses new challenges to Android security. In prior research we revealed that given the `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions, a malicious app on an authorized phone gains unfettered access to any Bluetooth device (e.g., Blood Glucose meter, etc.). Here we further show that sensitive text messages from online banking services and social networks (account balance, password reset links, etc.) are completely exposed to any app with either the `RECEIVE_SMS` or the `READ_SMS` permission. Similar security risks are present in other *channels* (Internet, Audio and NFC) extensively used to connect the phone to assorted external devices or services. Fundamentally, the current permission-based Discretionary Access Control (DAC) and SEAndroid-based Mandatory Access Control (MAC) are too coarse-grained to protect those resources: whoever gets the permission to use a channel is automatically allowed to access all resources attached to it.**

To address this challenge, we present in this paper *SEACAT*, a new security system for fine-grained, flexible protection on external resources. SEACAT supports both MAC and DAC, and integrates their enforcement mechanisms across the Android middleware and the Linux kernel. It extends SEAndroid for specifying policies on external resources, and also hosts a DAC policy base. Both sets of policies are managed under the same policy engine and Access Vector Cache that support policy checks within the security hooks distributed across the framework and the Linux kernel layers, over different channels. This integrated security model was carefully designed to ensure that misconfigured DAC policies will not affect the enforcement of MAC policies, which manufacturers and system administrators can leverage to define their security rules. In the meantime, a policy management service is offered to the ordinary Android users for setting policies that protect the resources provided by the third party. This service translates simple user selections into SELinux-compatible policies in the background. Our implementation is capable of thwarting all known attacks on external resources at a negligible performance cost.

* The two lead authors are ordered alphabetically.

NDSS '15, 8-11 February 2015, San Diego, CA, USA
Copyright 2015 Internet Society, ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23098

I. Introduction

The prosperity of the Android ecosystem brings in a broad spectrum of external resources (accessories, web services, etc.), which vastly enrich Android devices' functionalities. Nowadays, people use smartphone accessories not only for convenience and entertainment (e.g., Bluetooth earpieces, USB travel chargers, etc.), but for performing critical tasks related with domains such as healthcare and fitness (e.g., diabetes self-management [34]), finance (e.g., creditcard payments [8]) and even home security [32]. Furthermore, web resources are extensively utilized to support Android applications (apps for short), providing sensitive services like mobile banking, monetary transactions and investment management [11], [3]. Those external resources carry private user information (health, finance, etc.) and are responsible for security-critical operations (i.e., home security). However, it is not clear whether they are sufficiently protected by mobile operating systems (OS).

External resource protection on Android. In a previous study we showed that an unauthorized app with the `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions can acquire unfettered access to Android's Bluetooth healthcare accessories, and download sensitive medical data such as a patient's blood sugar level from them [28]. Also discovered in prior research is that network sockets opened by screenshot services are exposed to any apps with the `INTERNET` permission, allowing them to capture the screen of an Android phone at any given point [24]. Note that this lack of control on the network channel can also have other consequences: for example, given the `INTERNET` permission, an untrusted game app might be able to directly communicate with a corporate internal server, as an authorized app does. Even popular mobile credit-card payment systems were known to be vulnerable [26]: it is reported that credit-card information transmitted by the Square dongle to its mobile app through the Audio jack was not encrypted and could be easily picked up by any app with the `AUDIO` permission.

Although the problem with Square was later fixed with an AES encryption scheme built into its dongle (which increases the cost of the device), such accessory/app side solutions are rather ad hoc, whose security qualities are hard to control. Actually, most external resources today are completely unprotected, for reasons such as the desire to make things easy for users, limited capabilities of accessories, cost constraints, etc. Indeed, in our research, we successfully exploited the popular

Jawbone UP wristband [6] (an activity tracker recording a user's sleep, eating habits and other daily activities) through the Audio channel, and downloaded all its data using an unauthorized app (Section III-B). This lack of protection is also ubiquitous in apps receiving sensitive information from online resources through *Short Message Service* (SMS) and those connected to external devices using *Near-Field Communication* (NFC). More specifically, we analyzed high-profile online financial services (Bank of America, Chase, PayPal, etc.) and social networks (Facebook, Twitter, etc.) that deliver messages to their customers' devices (which should be received by the system app `com.android.sms` or the official apps of those services), and popular apps that have the NFC capability, and concluded that they are all vulnerable. Again, we found that unauthorized apps could get the user's messages once they are granted the `RECEIVE_SMS` or `READ_SMS` permission, and read from the NFC devices they are not supposed to touch when they possess the `NFC` permission. Of particular concern here are the short messages from banks, which often contain sensitive information such as a password for two-factor authentication, account balances, etc., and therefore should only be seen by their customers through `com.android.sms` or other official apps provided by the vendor. In addition, messages from Twitter and Facebook even carry links for resetting account passwords. Such information turns out to be completely unprotected from unauthorized apps. Demos for the attacks are posted on a private website [7].

Such threats to external resources are both realistic and serious, given the fact that indeed a lot of not-so-trustworthy apps do ask for related permissions (with a good reason for doing so sometimes) and have already been used by hundreds of millions of Android users. Take `RECEIVE_SMS` as an example. Popular third-party apps like Go Locker (50,000,000 to 100,000,000 installations) use it to receive messages (in this case, displaying the message on the lock screen). Our study on 13,500 highly-ranked apps (500 top apps from each of the 27 Google Play categories) from Google Play shows that altogether 560 apps require the `RECEIVE_SMS` or the `READ_SMS` permission, gleaning totally over 3 billion installations (Section III-B). The problem is that once those apps get the permission, they are also granted the privilege to read *any* messages, including those from Chase with one's account details, from Facebook with the link for resetting the password and from Life360 with the information about the family members' locations.

Fundamentally, *Android is not designed to protect its external resources*. Specifically, the *Discretionary Access Control* (DAC) mechanism Android provides to its user is based upon *permissions*, which are meant for authorizing access to an Android device's local resources such as camera, SD card, etc. When it comes to external resources, all permissions can do is to merely control individual *channels* through which the phone talks to external resources, such as Bluetooth, NFC, Internet, SMS and Audio. This access control is too coarse-grained to safeguard external resources of critical importance to the user, as it cannot differentiate those attached to the same channel, not to mention implementation of different access policies to protect them. As a result, whoever gets the permission to the channel (e.g., `BLUETOOTH`, `AUDIO`) is always given full access to *any* resources associated with the channel. Even for the emerging *SEAndroid* [30] powered kernel, a *Mandatory Access Control* (MAC) mechanism incorporated into Android to enable manufacturers or organizational administrators to specify and enforce finer-grained security policies, it just covers local resources (e.g., files) and cannot even assign a security tag to an external resource.

Security-enhanced channel control. Given the ongoing trend of using Android devices to support *Internet of Things* (IoT) for security-critical applications (e.g., home security), it becomes imperative to extend the Android security model to protect its external resources. This needs to be done on both the MAC and DAC layers. On one hand, device manufacturers and organizational administrators should be given the means to dictate the way their accessories and online resources should be accessed by apps: for example, only an official Samsung app is allowed to talk to the Samsung smart watch through Bluetooth. On the other hand, flexibility needs to be granted to ordinary users, who utilize third-party accessories (e.g., activity tracking wristband) and interact with third-party online services to manage their private information. For example, the user may hope to install her favorite apps like Go Locker but wants to ensure that they cannot read her bank's messages. Development of such protection mechanisms needs to be well thought-out, to avoid two separate mechanisms with duplicated functionalities, which complicates both the implementation and operations of the security model.

To tackle this prevalent problem, we developed in our research a suite of new techniques that protect Android external resources through mediating the channels they use to interact with the phone. Our approach, called *SEACAT* (Security-Enhanced Android Channel Control), integrates both MAC and DAC in a way that their policy compliance checks and enforcement go through the same mechanism. This integration simplifies the design of SEACAT and reduces its operational overheads. In the meantime, it is warily constructed, to avoid any interference between these two security models, ensuring that MAC policies are always followed even when DAC has been misconfigured by the user. More specifically, we extended SEAndroid's implementation on AOSP to describe the external resources over different channels. This is achieved by defining new SEAndroid *types* to represent the resources based upon their identities observed from their channels, including the `MAC` address of the Bluetooth accessory, the serial number of an NFC device, the IP address of a socket and the ID of an SMS sender. These types allow a system administrator to specify a set of mandatory security policies, which are enforced by the security hooks we placed at system functions related to those channels within both Android's framework/library layer and the Linux kernel. Whenever a call is generated, the hook checks its policy compliance through an SEAndroid function, and then determines whether to let the call go through in accordance with the outcome of the check.

Such operations are always applied to MAC policies first. For the system calls cleared of the MAC policies (that is, the calling processes are not touching any resources specified by the policy administrator), the hook further checks their compliance with a set of DAC policies, using the same function. These policies are defined in the same format as their MAC counterparts. They are maintained by a policy management service, through which an Android user and app developers can specify how an external device or an Internet service should be accessed by different apps, when those external resources are not included in any MAC policy. For example, the official app of the Chase bank can specify within its manifest file a DAC rule that only itself and the system app (`com.android.sms`) are allowed to receive text messages from the bank. Once this rule is approved by the phone user, a malicious app running on the phone will no longer be able to read messages from Chase, even when it has either the `RECEIVE_SMS` or the `READ_SMS` permission. Furthermore, this hybrid MAC/DAC approach enables SEACAT to protect even resources with no apparent identifiers: a user can leverage SEACAT's DAC component to restrict access to a channel when that is being used to communicate sensitive information.

We implemented SEACAT on Android 4.4 with the SEAndroid enhanced kernel 3.4.0 (AOSP 4.4.2_r12). Our prototype was run against all known attacks on different external resources, including the new ones we discovered in this work and those reported in prior research [28], [24]. Our study shows that SEACAT easily defeated all those attacks, at a very low performance overhead.

Contributions. We outline the paper's contributions below:

• *New understanding*. We investigated a set of channels that have not been systematically studied by prior research [28], [24], including SMS, Audio and NFC. Our findings provide further evidence pertaining Android's limitations in securing its external resources and highlight the need for finer-grained access control techniques to protect them.

• *New techniques*. We designed the first mechanism that provides a centralized and comprehensive protection of different kinds of Android external resources over their channels. Our approach supports both MAC and DAC in an integrated, highly efficient way, without undermining their security guarantees. These new techniques allow both system administrators and ordinary Android users to specify their policies and safeguard their accessories and other external resources.

• *Implementation and evaluation*. We implemented our design and evaluated our system against all known threats and also measured its performance. Our prototype successfully addresses all known security risks and can be swiftly extended to protect new channels.

II. Background

Android external resources and channels. Android and other mobile systems are routinely employed by their owners for managing their external resources. Particularly, almost every app running on these systems is supported by a remote service, which interacts with the app through the Internet or the telephone network (using short text messages). Such services are increasingly being utilized to store and process private user information, particularly the data related to online banking, social networking, investment, healthcare, etc. Moreover, the trend of leveraging smartphones to support the Internet of Things, brings in a whole new set of external devices, which carry much more sensitive data than conventional accessories (e.g., earpieces, game stations). Examples include health and fitness systems (e.g., blood pressure monitors [31], electrocardiography sensors [33], glucose meters [23]), remote vehicle controllers (e.g., Viper SmartStart [10]), home automation and security systems [32] and others. Those external devices and Internet resources are connected to smartphones through a variety of *channels*, which are essentially a set of hardware and software through which an app accesses the external resources. The most popular channels include Bluetooth, NFC, Internet, SMS and Audio.

Android security model. Android comes with a discretionary access control system characterized by its application sandboxing and permission model. Naturally, all third-party apps are considered untrusted by the system. Each Android app is confined within its own sandbox, which is enforced through the Linux-kernel level protection: every app runs as a separate Linux user whose data and process are isolated from those of other apps. To access the resources outside its sandbox, the app can get *permissions* from the system if it is signed by the manufacturers or other authorized parties or directly from the device owner when the app is installed. Those permissions enable the app to use sensitive resources such as GPS, camera, etc. This security model has been implemented across different Android layers, including the application-framework/library layer (also called *middleware*) and the Linux kernel layer. A problem of this security model is its coarse granularity. Specifically, for each channel an app needs to go through to touch external resources, the permission-based DAC is binary: the app is either granted unrestricted use of the channel to communicate with any resources attached to it, or denied the channel in its entirety. This is the root cause for the Bluetooth mis-bonding problem we have reported in prior research [28], in which any app with the `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions becomes entitled to access sensitive patient data collected by a health-care device.

SEAndroid. Security-Enhanced Android is a mandatory access control system built on top of Android [30]. It is designed to mediate all interactions of an app with the Linux kernel and other system resources. Furthermore, SEAndroid confines even system daemons to limit the damage they can cause once they are compromised. It also provides a centralized policy configuration for system administrators and device manufacturers to specify their policies.

More specifically, SEAndroid [30] associates with each subject (e.g., process) and object (e.g., file) a *security context* represented as a sequence `user: role: domain or type[: level]` and indexed by a *Security Identifier* (SID). The most important component here is the `type`[1]. Under a *type enforcement* (TE) architecture, a security policy dictates whether a process running within a `domain` is allowed to access an object labeled with a certain `type`. Following is a policy specified for all third-party apps: `allow untrusted_app shell_data_file:file rw_file_perms`. This policy states that all the apps associated with the domain `untrusted_app` are allowed to perform "`rw_file_perms`" operations on the objects associated with the type `shell_data_ file` which is of *class*[2] `file`.

The SEAndroid module currently incorporated into the

[1] `role` is for role-based access and `level` for multi-level security.

[2] A `class` defines a set of operations that can be performed on all objects associated with a type.

AOSP (Android Open-Source Project) 4.3 and 4.4 defines five domains within its policy files: `platform_app`; `shared_app`; `media_app`; `release_app` and `untrusted_app`. The last one is the domain assigned to all third-party applications installed by the user, in accord with Android's adversary model. These policy files are ready-only and compiled into the Android kernel code. They are enforced by security hooks placed at different system functions at the kernel layer. For example, the function `open` is instrumented to check the compliance of each call with the policies: it gets the type of the file to be opened and the domain of the caller, and then runs `avc_has_perm` with the SIDs of both the subject and object to find out whether this operation is allowed by the policies. Here `avc_has_perm` first searches an Access Vector Cache (AVC) that caches the policies enforced recently and then the whole policy file. In addition to the components built into the kernel, SEAndroid also includes a separate middleware MAC (MMAC) that works on the application-framework/library layer. The current implementation of MMAC on AOSP is limited to just assigning a security tag (`seinfo`) to a newly installed application. When Zygote forks a process for an app to be launched, it uses that tag in tandem with a policy file (`seapp_contexts`) to decide which SELinux domain should be assigned to it.

The current design of SEAndroid still cannot achieve the granularity for controlling external resources. It does not have types defined for the address of a Bluetooth device, the serial number for an NFC device, the SMS ID and the Audio port, nor does it place security hooks at the channels related to such resources. To control access to the resources, system functions at both the kernel layer and the framework/library layer need to be instrumented. SEAndroid does include a mechanism to mediate a range of IP addresses a process can connect to. However, the policy is hard-coded within the Linux kernel and its enforcement has not been exposed to Android's DAC. As a result, an ordinary user cannot specify rules to protect her Internet resources. In our research, we extended both the MAC and MMAC layers that are currently integrated into AOSP (Section IV), to protect those channels and further leveraged the enforcement mechanism to support a DAC system that guards a wide spectrum of external resources.

III. Understanding the Threats

To understand the security threats to Android external resources, we analyzed a set of prominent accessories and online services that utilize popular channels. Our findings echo the prior studies on Bluetooth and the Internet (local socket connections) channels [28], [24]. In particular, we previously found that security-critical Bluetooth devices are under the threats of information stealing and data injection attacks from an unauthorized app with the `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions [28]. In addition, others illustrated that all no-root third-party screenshot services can be exploited by a malicious app connecting to them through the Internet channel, which can also be abused to break other security policies, e.g., unauthorized access to enterprise internal servers. This work further shows that the SMS, Audio and NFC channels are equally under-protected, exposing private user information like bank account balances, password reset links etc. Those findings point to the security challenges posed by the widening gap between the coarse-grained Android protection and the current way of using external resources.

A. Methodology

Apps and external resources. In our study, we first looked at the permissions for accessing main channels that apps use to communicate with their external resources, including Audio, NFC, SMS, Bluetooth and Internet. Our purpose is to find out how pervasive these permissions are among third-party apps and how popular those apps are among Android users (indicating their willingness to grant the permissions to the apps). To this end, we studied 13,500 top-ranking apps collected from Google Play, whose total and average number of installations are presented in Table I. Some of these apps indeed need the permissions to provide services. An example is GoLocker [5], which uses the `RECEIVE_SMS` permission to record and process incoming messages, and notify the user of the event on her lock screen. Others are just over-privileged [20], asking for permissions they never use. Either way, Android users have to grant those apps what they want in order to install them, and many of them do, as indicated by billions of installations reported by Google Play.

We further investigated whether the apps using these permissions (particularly for Audio, NFC and SMS) to exchange sensitive data with their external resources do it in a secure manner. This is important because if this resource-app protection is not in place, other apps with the same permissions can get their hands on the data, due to Android's coarse-grained control on the channels. For this purpose, we chose from our collection a small set of top-ranking apps handling private information, including 13 Audio and 17 NFC apps. For SMS, we looked into 14 popular online services, including those provided by leading financial institutes (Bank of America, Chase, Wells Fargo, PayPal) and social networks (Facebook, Twitter, WhatsApp, WeChat, Naver Line, etc.), and a web mail (Gmail). Those services communicate with `com.android.sms` and sometimes, their own apps using short text messages. For Bluetooth and Internet, privacy threats to these channels have already been reported by prior research [28], [24].

Table II provides examples for the apps and services used in our study. All the services we analyzed clearly involve private user data. Such sensitive information is also handled by all five apps subjected to our analysis, i.e two credit-card-related NFC apps, one credit-card-related Audio app, one fitness Audio app and one app using SMS for two-step authentication. Some other payment related apps using the Audio jack, are heavily obfuscated and we were not able to decompile them using popular de-compilation tools (dex2jar, apktool). Most of the other apps in the Audio category are remote controllers or sensors that work with a dongle attached to the phone's Audio jack. Although those devices do not appear to be particularly sensitive (e.g., the camera that can be commanded remotely to take pictures), such functionalities (e.g., remote control) could have security implications when they are applied to control more sensitive devices. Our study also reveals that the most prevalent use of NFC apps is for reading and writing NFC tags (tags with microchips for short-range radio communication), which can be used to keep sensitive user data (e.g., a password for connecting to one's Wi-Fi access point) or trigger operations

TABLE I: Environment Study: 13,500 top apps (500 top apps for each of 27 Google Play categories) from Google Play. #downloads (total and average) and #apps per sensitive permission

No	Permission(s)	Total Downloads	Average Downloads	Number of apps
1	READ_SMS (and not 2)	1,519,670,000	11,965,906	127
2	RECEIVE_SMS (and not 1)	641,104,000	3,727,349	172
3	1 AND 2	1,220,503,000	4,676,257	261
4	BLUETOOTH (and not 5)	1,968,116,000	9,283,566	212
5	BLUETOOTH_ADMIN (and not 4)	0	0	0
6	4 AND 5	1,215,007,600	3,310,647	367
7	RECORD_AUDIO (and not 8)	1,960,964,950	2,689,938	729
8	MODIFY_AUDIO_SETTINGS (and not 7)	417,355,500	1,662,771	251
9	7 AND 8	3,164,060,000	8,218,338	385
10	NFC	2,583,934,500	14,850,198	174
11	INTERNET	20,153,137,630	1,694,965	11890

TABLE II: Critical Examples

Channel	App	Usage	# of downloads	Details
AUDIO	EMS+	Credit card reader	5,000 - 10,000	Decrypt : Creates a private key of RSA with hardcoded modulus and private exponent. Uses it to load session key which is used in AES to process messages from credit card dongle.
AUDIO	UP	Tracks sleep, physical activity and nutritional info	100,000 - 500,000	Doesn't include any authentication features. A repackaged app with different credential is able to read existing data from the band.
SMS	All bank services	Alert messages and Text banking	NA	Both SMS can be read by any app with SMS permission.Alert messages: sensitive financial activity and amount info. Text banking: receive, send money and check balance.
SMS	Chat and SNS	Authentication	100,000,000 - 1,000,000,000	2 step authentication; verification code sent via SMS.
NFC	SquareLess	Credit card reader	10,000 - 50,000	Reads credit card information. Malicious apps may also read credit card data as this app does.
NFC	Electronic Pickpocket RFID	Credit card reader	10,000 - 50,000	Reads credit card information. Malicious apps may also read credit card data as this app does.

(e.g., Wi-Fi connection). A more sensitive application of NFC is payment through a digital wallet. However, related NFC equipment is hard to come by.

Security analysis. Over those apps and services, we conducted both dynamic and static analyses to determine whether there is any protection in place when they use those channels. For SMS, we simply built an app with the `RECEIVE_SMS` permission to find out what it can get. All NFC apps were studied using NFC tags, in the presence of an unauthorized app with the `NFC` permission. For those in the Audio category, we analyzed a Jawbone UP wristband, a popular fitness device whose app (`com.jawbone.up`) has 100,000 to 500,000 downloads on Google Play, to understand its security weakness. In the absence of other Audio dongles, relevant apps were decompiled for a static code inspection to find out whether there is any authentication and encryption protection during those apps' communication with their external devices. Specifically, we looked for standard or home-grown cryptographic libraries (e.g., javax.crypto, BouncyCastle, SpongyCastle) within the code, which are needed for establishing a secret with the dongles. Also, the apps are expected to process the data collected from their dongles locally, instead of just relaying it to online servers, as a few payment apps do. This forces them to decrypt the data if it has been encrypted. Finally, we ran those apps to check whether a password or other secrets are needed to establish a connection with their dongles. Our analysis was performed on a Nexus 4 with Android 4.4.

B. Results

SMS. The SMS channel turns out to be intricate. Whenever the `Telephony` service on the phone receives a text message from the radio layer, the `InboundSmsHandler` puts it in an Intent, and then calls `SMSDispatcher` to broadcast it to all the apps that register with the event (`SMS_RECEIVED_ACTION` or `SMS_DELIVER_ACTION`) and have the `RECEIVE_SMS` permission. Also the `InboundSmsHandler` stores the message to the content provider of SMS. Such a message is limited to text content with up to 160 characters. To overcome this constraint, the message delivered today mainly goes through the *Multimedia Messaging Service* (MMS), which supports larger message length and non-text content such as pictures. What really happens when sending such a message (which can include multimedia content) is that a simple text message is first constructed and transmitted through SMS to the MMS on the phone, which provides a URI for downloading the actual message. Then, MMS broadcasts the message through the Intent to recipients with the `RECEIVE_MMS` permission and also saves the message locally through its content provider. An app with the `READ_SMS` permission can query both the SMS and MMS content providers for their contents. Our study shows that this mechanism can leak sensitive information.

As expected, all short messages from leading online services delivered to our Nexus 4 phone were fully exposed to the unauthorized app with the `READ_SMS` or the `RECEIVE_SMS` permission. Note that such messages should only be received and read by `com.android.sms` to display their content to the owner of the phone, as well as those services' official apps: for example, Facebook, Naver Line, WeChat and WhatsApp, directly extract a verification code from their servers' messages to complete a two-step authentication on the owner's behalf.

Information leaks through this under-regulated channel are serious and in some cases, catastrophic. A malicious app can easily get such sensitive information as account balances, incoming/outgoing wire transfers, debit card transactions, ATM withdrawals, a transaction's history, etc. from Chase, Bank of America and Wells Fargo, authorized amount for a transaction, available credit, etc. from Chase Credit Card and Wells Fargo Visa, and notifications for receiving money and others from

PayPal. It can also receive authentication secrets from Facebook, Gmail, WhatsApp, WeChat, Naver Line and KakaoTalk, and even locations of family members from Life360, the most prominent family safety online service. An adversary who controls the app can also readily get into the device owner's Facebook and Twitter accounts: all she needs to do is to generate an account reset request, which will cause those services to send the owner a message with a reset link and confirmation code. With such information, even the app itself can automatically reset the owner's passwords, by simply sending requests through the link using the mobile browser. A video demo of those attacks is posted online [7]. Note that almost all banks provide mobile banking, which allows enrolled customers to check their account and transaction status through SMS messages. Given the fact that even among our collection of 13,500 apps, already hundreds of third-party apps with the `READ_SMS` or `RECEIVE_SMS` permission have been installed billions of times (see Table I), for millions of users, their confidential information (account details, authentication secret, etc.) has already been exposed to those apps.

Audio. We analyzed the Jawbone UP wristband [6], one of the most popular fitness devices that utilize the low-cost Audio channel. The device tracks its user's daily activities, when she moves, sleeps and eats, and provides summary information to help the user manage her lifestyle. Such information can be private. However, we found that it is completely unprotected. We ran an unauthorized app that dumped such data from the device when it was connected to the phone's Audio jack.

For all other apps in the Audio category, we did not have their hardware pieces and therefore could only analyze their code statically. Specifically, among all 5 credit-card reading apps, PayPal, Square and Intuit are all heavily obfuscated, which prevented us from decompiling them. Those devices are known to have cryptographic protection and designed to send encrypted credit-card information from their card readers directly to the corresponding web services [9], [17]. The other two apps, EMS+ and Payment Jack, were decompiled in our research. Our analysis shows that both of them also receive ciphertext from their card-reader dongles. However, they decrypt the data on the phone using a hard-coded secret key. Since all the instances of these apps share the same key, an adversary can easily extract it and use it to decrypt a user's credit-card information downloaded from the app's payment dongle. Furthermore, all other apps, which either support sensors (e.g, wind meter) or remote controllers (e.g., remote picture taking), are unprotected, without authentication and encryption at all. This demonstrates the challenge for the device manufacturer and app developer to come up with a practical resource-device protection mechanism, highlighting the need for an OS-level solution.

NFC. Android employs a dispatcher mechanism to decide which app can access an NFC device or tag. The dispatcher will choose an app to get NDEF data and the device/tag's serial number, according to the priorities that the apps register with through Intent-filters. These priorities from the highest to the lowest are: `NDEF_DISCOVERED`, `TECH_DISCOVERED` and `TAG_DISCOVERED`. The system app (`com.google.android.tag`) runs with the lowest priority. According to its priority, an app receives the Intent that carries NDEF (NFC Data Exchange Format) data scanned by the phone. When the NFC device/tag has no NDEF data (but data in other formats) on it, the Intent dispatched to the app just contains the serial number of the device/tag, not the data, and the recipient is supposed to directly communicate with the device/tag using the number to get the data. Also, when multiple apps have the same priority, an "Activity Chooser" dialogue will be presented to the user for selecting the receiving app. This process negatively affects users' experience as every single time that a tag is discovered a pop-up box will appear, even for the tag that has been used before [1], [2].

In our research, 5 out of 17 popular NFC apps (e.g., NFC Tools) we found are used to read and write NFC tags [3]. These apps allow users to store any data on tags, including sensitive information (e.g., a password for one-touch connection to a Wi-Fi access point). However, there is no authentication and encryption protection at all[4]. We ran an unauthorized app with the `NFC` permission to collect data from the tag whenever our Nexus phone touched it. The "Activity Chooser" mechanism could offer some protection, but only in the case a malicious app does not have a higher registered priority than the legitimate one. This can be a problem, for example, when one only uses the system NFC app, which has the lowest priority. Also the approach cannot be used by system administrators to enforce any mandatory policies. Android is also vulnerable in the case that a malicious app is in the foreground with `foregroundDispatch` enabled. When this happens, the OS will send the Intent to that app allowing it data access.

Among the rest of the apps, NFC ReTag FREE utilizes the serial number of an NFC tag to trigger operations. Again, since the communication through the NFC channel is unprotected, a malicious app can also acquire the serial number, which leaks out the operation that the legitimate app is about to perform. The only NFC app with protection is the NFC Passport Reader. What it does is to use one's birth date, passport number and expiration date to generate a secret key for encrypting other passport information. The problem is, once those parameters are exposed, the adversary can recover the key to decrypt the data collected from the NFC channel.

Discussion. From the 13,500 apps collected, we further note that any app with the Internet permission (for 93% of them, each has been installed 1,694,965 times on average, as shown in Table I) can access any domain. With advertising already a ubiquitous way for apps to profit, more and more apps request the `INTERNET` permission to allow the ad component to work. This creates privacy risks when such apps are being used in a business or private network, as they can freely connect to any internal servers, if proper protection is not in place. Therefore, we believe that the capability to let users control access to IP/domains is important. For example, an organization can require its employees to set policies on their phones to ensure that internal IPs are only accessed by its enterprise apps, not Angry Birds.

Also we see nearly 600 third-party apps asking for the `BLUETOOTH` or `BLUETOOTH_ADMIN` permissions, and having been installed over 3 billion times (Table I). The presence of

[3]This is expected as this is one of the major use cases of NFC on Android [4]

[4]There are more expensive tags such as MIFARE that support encryption and authentication. The app using those tags needs the user to manually enter a secret. Clearly, they are not used for protecting the information like Wi-Fi passwords, which should be passed to one's device conveniently.

these apps, which most likely are not fully trusted, constitutes a serious threat to private user data stored on different Bluetooth accessories (e.g., glucose meters [23]), as we have reported in prior research [28]. Note that so far, there is no effective way to address this issue. Although a framework-layer defense mechanism (called *Dabinder*) has been proposed in our previous work [28], that comes with inherent limitations, as it can actually be *bypassed* by a malicious app with native code. This is because the protection was implemented within the Bluetooth service (Section IV-C, Figure 3), while native code with the `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions can directly talk to the Bluetooth stack to establish a connection with the external device. In Section IV, we describe a new technique that provides comprehensive protection and supports Mandatory Access Control on this channel.

IV. External Channel Control

Our study presented on Section III emphasizes the need for a more fine-grained control over the channels of communication with Android external resources, with strong security guarantees. Ad-hoc solutions on each channel fall short of providing such guarantees and further suffer from the lack of backward-compatibility, flexibility, extensibility to future channels and maintainability. In this section, we present the first design for protecting Android's external resources. Our system, called *SEACAT* employs a flexible hybrid MAC/DAC approach. It extends SEAndroid's MAC to safeguard resources with distinct resource identifiers such as SMS, NFC, Bluetooth and Internet, and also adds in a DAC module to allow the user and app developers to specify rules through simple and straightforward user interaction for all these channels. In addition, its DAC component allows control of channels even in the absence of resource identifiers. We illustrate this on the Audio channel. We implemented SEACAT on AOSP 4.4_r12 with an SEAndroid-powered kernel 3.4.0.

A. Design Overview

Challenges. Our objective is to develop a simple security mechanism that supports flexible fine-grained mandatory and discretionary protection of various external resources through controlling their channels of communication. Our solution should also be extensible as potential channels, app functionalities and developer practices are hard to predict. Furthermore the system has to be maintainable and easily manageable. Lastly, our solution should be efficient, backward-compatible and effective.

However, achieving this goal is by no means a smooth sail. Here are a few technical challenges that need to be overcome in our design and implementation.

• *Limitations of SEAndroid*. Today's SEAndroid does not model external resources. Even after it is extended to describe them, new enforcement hooks need to be added to system functions scattered across the framework/library layer and the Linux kernel. For example, the Bluetooth channel on Android 4.4 is better protected on the framework layer, which has more semantic information, while the control on the Internet should still happen within the kernel. Supporting these hooks requires a well though-out design that organizes them cross-layer under a unified policy engine and management mechanism for both MAC and DAC.

• *Complexity in integration*. The current Android already has the permission-based DAC and SEAndroid-based MAC. An additional layer of DAC protection for external resources could complicate the system and affect its performance[5]. How to integrate SEACAT into the current Android in the most efficient way is challenging.

Design. To address these challenges and meet our objectives, we have come up with a centralized design that integrates policy compliance checks from both the framework and the kernel layer, and enforces MAC and DAC policies within the same security hooks (Figure 1). It safeguards all known external resources in a unified way allowing its easy extension to new channels. More specifically, the architecture of SEACAT includes a policy module, a policy enforcement mechanism and a DAC policy management service. At the center of the design is the policy module, which stores security policies and provides an efficient compliance-check service to both the framework and the kernel layers. It maintains two policy bases, one for MAC and the other for DAC. The MAC base is static, which has been compiled into the Linux kernel in the current SEAndroid implementation. The DAC base can be dynamically updated during the system's runtime. Both of them are operated by a policy engine that performs compliance checks. The engine is further supported by two Access Vector Caches (AVCs), one for the kernel and the other for the framework layer. Each AVC caches the policies recently enforced using a hash map. Due to the locality of policy queries, this approach can improve the performance of compliance checks. Since DAC policies are in the same format as MAC rules, they are all served by the same AVC and policy engine.

The enforcement mechanism comprises a set of security hooks and two pairs of mapping tables. These hooks are placed within the system functions responsible for the operations on different channels over the framework layer and the kernel layer. Whenever a call is made to such a function, its hook first looks for the security contexts of the caller (i.e., app) and the object (e.g., a Bluetooth address, the Sender ID for a text message, etc.) by searching a MAC mapping table first and then a DAC table. The contexts retrieved thereby, together with the operation being performed, are used to query the AVC and the policy engine. Based upon the outcome, the hook decides whether to let the call go through. Just like the AVC, each mapping table has two copies, one for the framework layer and the other for the kernel. Also, the MAC table is made read-only while the DAC table can be updated during runtime.

Both the DAC policy base and DAC mapping table are maintained by the policy management service, which provides the user an interface to identify important external resources (from their addresses, IDs, etc.) and the apps allowed to access them. Also it can check manifest files of newly installed apps to extract rules embedded there by the developer (e.g., only the official Chase app can get the text message from Chase) to ask for the user's approval. Those policies and the security

[5]Note that this new DAC cannot be easily integrated into the permission mechanism, since the objects there (different Bluetooth devices, web services, etc.) can be added into the system during runtime.

contexts of the labeled resources are uploaded to the DAC base and the mapping tables respectively.

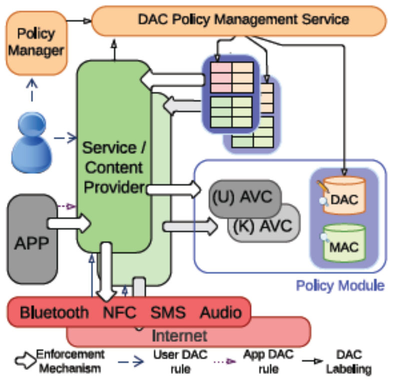

Fig. 1: *SEACAT* architecture

Adversary model. Like SEAndroid, the security guarantee of SEACAT depends on the integrity of the kernel. We have to assume that the adversary has not compromised the kernel to make the approach work. In the meantime, SEACAT can tolerate corrupted system apps, as long as they are confined by SEAndroid. Furthermore, the DAC mechanism is configured by the user and therefore could become vulnerable. However, our design makes sure that even when it is misconfigured, the adversary still cannot bypass the MAC protection in place. Finally, we regard all third-party apps as untrusted and thus we assume the presence of malicious apps on the user's device, with proper permissions to access all aforementioned channels.

B. Policy Specification and Management

To control external resources, we first need to specify the right policies and identify the subjects (i.e., apps) and objects (e.g., Bluetooth glucose meter, the Chase bank, etc.) to apply them. This is done within the policy module and our policy management service.

Policy specification. As discussed before (Section II), an SEAndroid rule determines which domain is allowed to access which resources, and how this access should happen. To specify such a rule for external resources, both relevant domains (for apps) and types (for external resources) need to be defined. The `domain` part has already been taken care of by SEAndroid: we can directly declare ones for any new apps whose access rights, with regard to external resources, need to be clarified. When it comes to `types`, those within the AOSP Android have been marked as `file_type`, `node_type` (for sockets and further used to specify IP range), `dev_type`, etc. In our research, we further specified new categories of types (or `attributes`), including `BT_type` for MAC addresses of Bluetooth devices, `NFC_type` for NFC serial numbers and `SMS_type` for SMS Sender ID (originating addresses). Here is an example policy based upon these domains and types:

```
allow trusted_app bt_dev:btacc rw_perms
```

where `bt_dev` is a type for Bluetooth devices (identified by their MAC addresses) and `btacc` includes all the operations that can be performed on the type. This policy allows the apps in the domain `trusted_app` to read from and write to the MAC addresses in the type `bt_dev`. Later we describe how to associate such a domain with authorized apps, and the type with external resources.

The DAC Policy Management Service, uses the user's input to construct in the background the DAC policies. The DAC policies used in SEACAT are specified in the same way as the MAC policies, using the same format, which enables them to be processed by the policy engine and AVC also serving MAC policies. The DAC policy base, includes a set of types defined for the Audio channel. Audio has not been included in the MAC policies since the device attached to it cannot be uniquely identified: all we know is just whether the device is an input (headset) or output (speaker) device or the one with both capabilities. Even in this type of channels where there is a lack of identifiers for the external resources, SEACAT's hybrid approach allows protection through its DAC component. Specifically, for user-defined DAC policies, we provide a mechanism to lock the whole channel when necessary, a process elaborated later for audio. Moreover, although the DAC base is supposed to be updated at runtime, to avoid the overheads that come with such updates, we predefined a set of "template" policies that connect a set of domains to a set of types in different categories (Bluetooth, NFC, SMS, Internet and Audio) with read and write permissions. The domains and types of those policies are dynamically attached to the apps and resources specified by the user during runtime. In this way, SEACAT only needs to maintain a mapping table from resources to their security contexts (`user_seres_contexts`) before the template rules run out.

App labeling. For the domains defined for MAC, how they are assigned to apps is also specified in the policies. Our implementation allows the administrator to grant trusted apps, permissions to use restrictive external resources. Such apps are identified from the parties who sign them. Specifically, when an app is being installed, SEAndroid assigns it an `seinfo` tag according to its signature. The mapping between this tag and the app's domain is maintained in the file `seapp_contexts`, which *Zygote*, the Android core process that spawns other processes, reads when determining the app's security context during its runtime.

Labeling apps for DAC is handled by the policy management service, which includes a set of hooks within the `PackageManager` and `installd`. Before an app is installed, these hooks present to the user a "dialogue box". This allows the user to indicate whether the app should be given a domain associated with certain channels (Bluetooth, NFC, SMS, Internet and Audio), so that it can later be given the privilege to access protected external resources. The user never deals with policies directly. She only answers to simple questions such as "Does this app come with an accessory?". For example, if the user answers positively to the previous question, the policy management service will assign a domain to the app. For an app assigned a domain, the `PackageManager` labels it with an `seinfo` tag different from the default one (for untrusted, unprivileged apps) and stores the tag alongside its related domain within a dynamic mapping file `user_seapp_contexts`. Note that this action will only be taken, in the absence of a MAC rule already dictating the domain assignment for this app.

We further modified `libselinux`, which is used by Zygote, to assign the appropriate security context to the process forked for an app. Our instrumentation within `libselinux` enables loading `user_seapp_contexts` for retrieving the security context associated with a user-defined policy. Note that again, when an `seinfo` tag is found within both `seapp_contexts` and `user_seapp_contexts`, its context is always determined by the former, as the MAC policies always take precedence. In fact the system will never create a DAC policy for an external resource that conflicts with a MAC policy. Nevertheless, if a compromised system app manages to inject erroneous DAC policies, they will never affect or overwrite MAC policies.

The design of SEACAT also allows the app developer to declare within an app's manifest the external resource the app needs exclusive access to. With the user's consent, the app will get a domain and the resource will be assigned a type to protect their interactions through a DAC rule. This approach makes declaration of DAC policies convenient: for example, the official app of Chase can state that only itself and Android system apps are allowed to receive the text messages from Chase; a screenshot app using an ADB service can make the IP address of the local socket together with the port number of the service off limit to other third-party apps.

External resource labeling. For standard local resources, such as files, SEAndroid includes policies that guide the OS to find them and label them properly. For example, the administrator can associate a directory path name with a type, so that every file stored under the directory is assigned that type. The security context of each file (which includes its type) is always kept within its extension, making it convenient to retrieve the context during policy enforcement. When it comes to external resources, however, we need to find a new way to label their identifiers and store their tags. This is done in our research using a new MAC policy file `seres_contexts`, which links each resource (the MAC address for Bluetooth, the serial number for NFC, the Sender ID for SMS and the IP/port pair of a service) to its security context. The content of the file is pre-specified by the system administrator and is maintained as read-only throughout the system's runtime. It is loaded into memory buffers within the framework layer and the Linux kernel respectively, and utilized by the security hooks there for policy compliance checks (Section IV-C).

Labeling external resources for the DAC policies is much more complicated, as new resources come and go, and the user should be able to dynamically enable protection on them during the system's runtime. SEACAT provides three mechanisms for this purpose: 1) connection-time labeling, 2) app declaration and 3) manual setting. Specifically, connection-time labeling happens the first time an external resource is discovered by the OS, for example, when a new Bluetooth device is paired with the phone. Also, as discussed before, an app can define the external resource that should not be exposed to the public (e.g., only system apps and the official Facebook app can get messages from the Sender ID "FACEBOOK"). Finally, the user is always able to manually enter new DAC policies or edit existing ones through an interface provided by the system. Note that, the user never actually deals with SELinux-like policies. Those are automatically constructed when the user answers simple questions such as "Please select the app you downloaded for this accessory.", or when she maps the ID "FACEBOOK" to the Facebook app.

For different channels, some labeling mechanisms work better than others. Bluetooth and NFC resources are marked mainly when they are connected to the phone: whenever there are apps assigned domains but not associated with any Bluetooth or NFC resources, SEACAT notifies the user once a new Bluetooth device is paired with the phone or an NFC device is detected; if such a new device has not been protected by the MAC policies, the user is asked to select, through an interface, all apps (those assigned domains) that should be allowed to access it (while other third-party apps' access requests should be denied). After this is done, a DAC rule is in place to mediate the use of the device. Note that once all such apps have been linked to external resources, SEACAT will no longer interrupt the user for device labeling, though she can still use the policy manager to manually add or modify security rules.

In our implementation, we modified a few system apps and services to accommodate this mechanism. For Bluetooth, we changed `Settings`, the Bluetooth system app and service. When the `Settings` app helps the user connect to a newly discovered Bluetooth device, it checks the device's MAC address against a list of mandatory rules. If the address is not on the list, the Bluetooth service pops an interface to let the user choose from the existing apps assigned domains but not paired with any resources. This is done through extending the `RemoteDevices` class. The MAC address labeled is kept in the file `user_seres_contexts`, together with its security context. This file is uploaded into memory buffers (for both the kernel and the framework layer) for compliance checks. For NFC, whenever a new device is found, Android sends an Intent to the app that registers with the channel III-B. In our implementation, we instrumented the NFC Intent dispatcher to let the user label the device and specify the apps allowed to use it when the dispatcher is working on such an Intent. This is important when the NFC device is security critical, as now the control is taken away from the potentially untrusted apps and delegated to the user (if no MAC mechanism is in place) during runtime. Furthermore, by providing this mechanism, the system can protect itself, and it is deprived of any dependency on end-to-end authentication between apps and external devices. Lastly, by utilizing the association of apps with resources specified in MAC and DAC policies, the user can read already labeled tags directly, avoiding unnecessary interaction with the "Activity Chooser" mechanism every single time an NFC device is discovered, which immensely improves the usability of the reading-an-NFC-device task. Again, the result of the DAC labeling is kept in `user_seres_contexts`.

External resources associated with SMS and Internet are more convenient to label through app declaration and manual setting. As discussed before, an app can request exclusive access to the text messages from a certain SMS ID. The user can also identify within the interface of our policy manager a set of SMS IDs (32665 for "FACEBOOK", 24273 for "Chase", etc.) to make sure that only `com.android.sms` can get their messages[6]. Also, there are cases where manual setting is needed for Internet. For example an organization can require its employees to set policies on their phones to ensure that internal

[6]The SMS IDs for services are public. It is easy to provide a list of well-known financial, social-networking services to let the user choose from.

IPs are only accessed by its enterprise apps. Other Internet resources should be specified by the app. For example, those using ADB-level services [24] can state the local IP address and services' port numbers to let our system label them.

Pertaining Audio, we label the whole channel at the right moment. Specifically, the DAC rule for the channel is expected to come with the app requiring it or set manually by the user through the policy manager. Whenever the system observes the Audio jack is connected to a device that fits the profile (input, output or mixed), SEACAT just pops up a "dialogue box" asking the user whether the device needs protection, if a DAC rule has already been required by either an app or the user. We can avoid this window popup when the app (the one expected to have exclusive access to the Audio channel) is found to run in the foreground. In either case, the whole Audio channel is labeled with a type, which can only be utilized by that app, system apps and services. This information is again stored in `user_seres_contexts` for policy enforcement. Notably, as soon as the device is detached from the Audio jack, the type is dropped from the file, which releases the entire channel for other third-party apps. To completely remove the pop-ups, the user can set the system to an "auto" mode in which the Audio is only labeled (automatically) when the authorized app is running. In this case, the user needs to follow a procedure to first start the app and then plug in the device to avoid any information leak.

C. Policy Compliance Check and Enforcement

Compliance check. To perform a compliance check, a hook needs to obtain the security contexts of the subject (the app), the object (MAC address, NFC serial number, etc.) and the operation to be performed (e.g., read, write, etc.) to construct a query for the policy engine (see Figure 2). Here the subject's context can be easily found out: on the framework layer, this is done through the SEAndroid function `getPidContext`, which utilizes the PID of a process to return its context information. Although the same approach also works within the Linux kernel, a shortcut is used in controlling Internet connections through sockets. Specifically, within the socket's structure, SEAndroid already adds a field `sk_security` to keep the security context of the process creating the socket. The field is used by the existing hooks to mediate the access to IP/port types. In our research, we put the enforcement of DAC policies there, which involves finding the security contexts of an IP-port pair from a DAC table within the kernel.

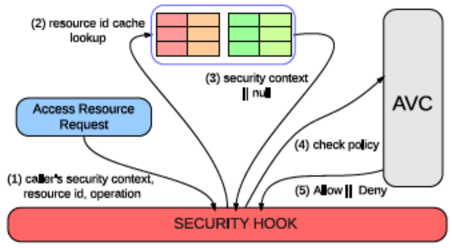
(2) resource id cache lookup
(3) security context || null
AVC
Access Resource Request
(4) check policy
(1) caller's security context, resource id, operation
(5) Allow || Deny
SECURITY HOOK

Fig. 2: *SEACAT* Policy Compliance Check

The object's context is kept within the MAC policy file `seres _contexts` and the DAC file `user_seres_contexts`. To avoid frequently reading from those files during the system's runtime, SEACAT uploads

TABLE III: A list of operations *SEACAT* offers to system apps and services.

FUNCTION	DESCRIPTION
loadPDPolicy	Loads the MAC (seres_res_contexts) and DAC (user_seres_contexts) policy bases containing the resource with security context associations, into the SEACAT memory buffers.
getResourceSecContext	Performs a lookup in the SEACAT memory buffers for a security type assigned to a resource.
getResourceChannel	Performs a lookup in the SEACAT memory buffers for the channel that a resource belongs to.
isResourceMAC	Returns 1 if the resource is present in SEACAT memory buffers and was loaded from the MAC policy base, 0 if it was loaded from the DAC policy base, or NULL otherwise.
insertDACRes	Stores the security context of a resource in the appropriate memory buffer and the corresponding policy base.
getDomain	Returns the security context assigned to a third-party app.

their content to a pair of buffers in the memory both in the framework layer and the kernel. These buffers are organized as hash maps, serving as the mapping tables to help a security hook retrieve objects' security contexts. Specifically, we implemented a function for searching the mapping tables within `libselinux`, and exposed this interface to the framework so that the security hooks can access it through Java or native code. Within the kernel, we built another mapping table for the DAC policy[7]. This table is synchronized automatically with the one for the framework layer to make sure that the same set of DAC policies are enforced on both layers. The set of operations we created for manipulation and retrieval of information from the memory buffers and exposed through libselinux to the rest of the system, are listed within Table III.

Given the security contexts for a subject (the app) and an object (e.g., an SMS ID), a security hook is ready to query the AVC and policy engine to find out whether an operation (i.e., system call) is allowed to proceed. On the framework layer, this policy compliance check can be done through `selinux_check_access`. In our research, we wrapped this function, adding program logic for retrieving an object's security context from the mapping table. The new function `seacat_check_access` takes as its input a resource's identifier (Bluetooth MAC, SMS ID, etc.), the caller's security context and the action to be performed, and further identifies the resource's security context before running the AVC and the policy engine on those parameters. Note that for the resource appearing within both MAC and DAC tables, its security context is only determined by the MAC policy. Also, the resource not within either table is considered to be public and can be accessed by any app. Again, this new function is made available to both Java and native code. The same mechanism was also implemented within the kernel, through wrapping the compliance check function `avc_has_perm`. The AVC and the policy engine are largely intact here, as our system was carefully designed to make sure that the DAC rules are in the same format as their MAC counterparts and therefore can be directly processed by SEAndroid.

Security hooks. Here we elaborate how security hooks were implemented to enforce policies.

[7]Note that we did not build the table for MAC here, since SELinux already has a table for enforcing MAC policies on IPs. Also, all other channels are enforced on the framework layer.

• *Bluetooth*. To fully control the Bluetooth channel, all its functions need to be instrumented. A prominent example here is `Bluetooth Socket.connect` within the Bluetooth service, which needs to be invoked for establishing a connection with an external device. In our implementation, we inserted a security hook at the beginning of the function to mediate when it can be properly executed. A problem is how to get the process ID (PID) of the caller process for retrieving its security context through `getPidContext`. Certainly we cannot use the PID of the party that directly invokes the function, which is actually the Bluetooth service. What we did is to turn to Binder, which proxies the inter-process call (IPC) from the real caller app. Specifically, our hook calls `getCallingPid` (provided by Binder) to find out the app's PID and then its security context, and passes the information to the Bluetooth stack. Inside the stack (see Figure 3) we instrumented the actual connection attempt, which uses the app's security context, the Bluetooth MAC address to be connected and the "`open`" operation as inputs to query `seacat_check_access`. What is returned by the function causes the connection attempt to either proceed or immediately stop. The Bluetooth service is notified accordingly regarding the success or failure of the connection attempt. In the same manner, we can instrument other functions in the Bluetooth stack.

Figure 3 illustrates how our policy enforcement is incorporated into the Android Bluetooth architecture. Our hooks are placed within the Bluetooth stack, which cannot be circumvented by a malicious app without the system privilege. This is in contrast to Dabinder, a framework-layer protection mechanism we proposed in prior work [28]. Dabinder just works within the Bluetooth service and can be bypassed by native code that directly talks to the Bluetooth stack (also illustrated in the figure), when it has the Bluetooth permissions. Further, our current protection is integrated into SEAndroid and can be used to enforce MAC policies, which Dabinder cannot.

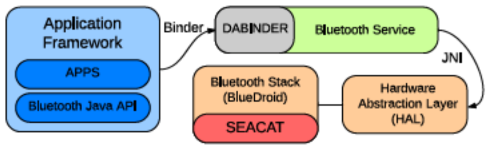

Fig. 3: Dabinder [28] and SEACAT enforcement for Android Bluetooth (BlueDroid).

• *NFC*. For the broadcomm chip on Google Nexus 4 devices, the NFC stack has been implemented on the framework/library layer through `libnfc-nci`. As a result, all our security hooks are placed on this layer, within major NFC functions `readNdef`, `writeNdef` and `connect`, for mediating a caller process's operations on an NFC device with a particular serial number (which is treated as the device's identifier).

A tricky part is that when a new NFC device is found to be in proximity, NFC runs a dispatcher to identify which apps have registered for that device through Intent-filters. When the tag/device contains NDEF data, and there exists at least one app with the `NDEF_DISCOVERED` priority, the dispatcher will deliver an Intent encapsulating this data to the identified app. This target app can access the data directly through the Intent. In cases where multiple apps request access to that NFC device, an "Activity Chooser" box will be presented to the user so she can choose which activity should be launched. This operation impedes user experience as it happens every single time the tag/device is discovered. Further there is no guarantee that the "Activity Chooser" will be presented to the user (see Section III-B). In our research, we instrumented the NFC dispatcher mechanism to execute the MAC and DAC policy compliance check against all apps requesting the Intent (whether that is one app or more), with regards to a specific device serial number. For those that fail the check, the dispatcher simply ignores them and therefore the Intent with the NFC device's contents will never reach them.

Our approach not only provides stronger security guarantees than the current system, as it controls all cases, but in contrast with a lot of security solutions, it also ameliorates users' experience. Once a rule is in place for an app-device interaction, then the "Activity Chooser" will not be presented again to the user every time the device is discovered.

• *Internet*. The Internet channel has been controlled inside the kernel, with security hooks placed within the functions for different socket operations. As discussed before, SEAndroid has already hooked those functions for enforcing mandatory policies on IP addresses, port numbers and others. In our research, we extended those existing hooks to add enforcement mechanisms for DAC policies. Specifically, we changed `selinux_inet_sys_rcv_skb` and `selinux_sock_rcv_skb_compat` to enable those wrapper functions to search the DAC mapping table within the kernel for the security contexts of IP-port pairs specified by the user and use such information to call `avc_has_perm`. Note that this enforcement happens to the objects (IP and port numbers) that have already passed the MAC compliance check: that is, those IP and port numbers are considered to be public by the administrator, while the user can still add her additional constraints on which party should be allowed to access them.

• *SMS*. To mediate this complicated channel, we instrumented both SMS and MMS to track the entire work flow and enforce MAC and DAC policies right before a message is being handed over to apps (Figure 4). Specifically, we hooked the function `processMessagePart` within the `SMSDispatcher` (see Section III-B) to get the ID of the message sender (i.e., the originating address) through `SmsMessageBase.getOriginatingAddress()`. This sender ID serves as an input for searching the mapping tables. The security context identified this way is then attached to the Intent delivered to MMS as an extra attribute `SEC_CON`. On the MMS front, a security hook inspects the attribute and further propagates the security context to another attribute within a new Intent used to transmit the real message once it is downloaded. We also modified the function `deliverToRegisteredReceiverLocked` within `BroadcastQueue` to obtain the security context of each app recipient involved in the broadcast and runs `seacat_check_access` to check whether the app should be allowed to get the message before adding the message to its process message queue.

Besides getting SMS message from Intent receiver for `SMS_RECEIVED_ACTION` or `SMS_DELIVER_ACTION`[8], an app can also directly read from the SMS or MMS content provider given the `SMS_READ` permission. To mediate such

[8]On Android 4.4, only the default sms app gets this Intent

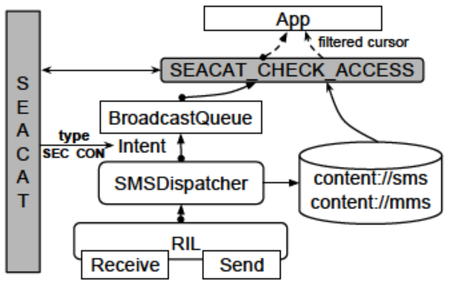

Fig. 4: *SEACAT* SMS enforcement

accesses, we further instrumented the content provider of `SMSProvider` and `MMSProvider` to perform the policy compliance check whenever an app attempts to read from its database: based on the app's security context and each message's address, our hooks sanitize the cursor returned to the app, removing the message it is not allowed to read.

• *Audio.* Like SMS, the Audio channel is also mediated on the framework layer. Whenever a device is connected to the Audio jack, `WiredAccessoryManger` detects the device and calls `setDeviceStateLocked`. Within the function, we placed a hook that identifies the type of the device (input/output/mixed) and checks the presence of a policy that controls the access to such a device. If so, it directly calls the SEACAT function `SensChannel.assignType` to assign the object type in the policy to the Audio channel (which prevents the channel from being used by unauthorized third-party apps) when an authorized app is running in the foreground. Otherwise, it pops up a "dialogue box" to let the user decide whether the device is the object within the policy and therefore needs to be protected. In either case, as soon as the device is unplugged from the Audio jack, the hook immediately removes from the DAC mapping table the entry for the Audio channel, thereby releasing it to other third-party apps.

Policy enforcement happens within the functions for collecting data from the Audio channel. Particularly, SEACAT has a hook inside the `startRecording` method of `android.media.AudioRecord`. Once the method is invoked, it looks for the security contexts for the calling process (through `getContext`) and the Audio channel (using `getResourceSecContext`) to check polices and determine whether the call can go through.

V. Evaluation

In our research, we evaluated the effectiveness of SEACAT against all existing threats to Android external resources and measured the overheads it introduces. Our study was performed on a pair of Nexus 4 phones with Android 4.4 (android-4.4.2_r12), kernel KRT16S, with the 3.4 kernel (android-msmmako3.4kitkatmr0): one installed with an unmodified OS (AOSP) to serve as a baseline, and the other with the SEACAT-enhanced kernel and framework. Following we report what we found. The video demos for this study can be found online [7].

A. Effectiveness

Known threats. Table IV presents 5 known threats to external resources used in our research, which include collection of data from iThermometer through Bluetooth misbonding [28], unauthorized use of an ADB proxy based screenshot service through local socket connections [24], as well as our attacks on SMS (stealing text messages from Chase and Facebook), Audio (gathering activity data from the UP wristband) and NFC (reading sensitive information from NFC tags). In our study, we ran those attacks on the unprotected Nexus 4, which turned out to be all successful: the malicious app acquired sensitive information from the external resources through the channels (Bluetooth, SMS, Internet, Audio and NFC), exactly as reported in prior research [28], [24] and Section III.

TABLE IV: Threats to Android external resources. (∗) attacks demonstrated here.

No	KNOWN THREATS
1	Bluetooth misbonding attack
2	unauthorized adb-based screenshots
3*	unauthorized read of an SMS message
4*	unauthorized access to audio device
5*	unauthorized read of an NFC device's contents

Preventing unauthorized resource access. All such attacks, however, stopped working on the SEACAT-enhanced Nexus 4. Specifically, after assigning a type to the MAC address of the iThermometer device through our policy management service, we found that only the official app of iThermometer, which was assigned to an authorized domain, was able to get data from the device [7]. The malicious app running in the `untrusted_app` domain could no longer obtain body temperature readings from the thermometer. For SMS, once we labeled the Sender IDs of Chase and Facebook with a type that can only be accessed by the apps within the system domain, the third-party app could not find out when messages from those services came, nor was it able to read them from the SMS content provider `content://sms`. On the other hand, the user could still see the messages from `com.android.sms` [7]. Similarly, the screenshot attack reported in prior research [24] was completely thwarted when the local IP address and port number was labeled. Also the security type given to the serial number of an NFC tag successfully prevented the malicious app from reading its content. In the presence of both authorized and unauthorized apps, the protected Nexus directly ran the authorized app, without even asking the user to make a choice, as the unprotected one did. For Audio, after the user identified the presence of the Jawbone wristband or the official app of the device was triggered, the channel could not be accessed by the malicious app. It was released only after the wristband was unplugged from the Audio jack.

The effectiveness of our protection was evaluated under both MAC and DAC policies for all those attack cases, except the one on the Audio channel, which we only implemented the DAC protection (Section IV-B). Also, we tried to assign a resource specified by a MAC policy to a DAC type using our policy manager and found that the attempt could not go through. Even after we manually injected such a policy into our DAC database and mapping table (which cannot happen in practice without compromising the policy manager), all the security hooks ignored the conflicting policy and protected the resources in accordance with the MAC rules.

B. Performance

Experimental setting. To evaluate the performance impact of SEACAT, we measured the execution time for the operations that involve our instrumentations, and compared it with the delay observed from the baseline (i.e., the unprotected Nexus 4). Table V shows examples of the operations used in our research. In the experiments, we conducted 10 trials for each operation to compute its average duration. Note that comparison with SEAndroid [30] is moot, as the hooks we placed to enforce control over external resources are not present there. Thus the operations we measured will provide the same result whether on AOSP 4.4 or SEAndroid.

Specifically, we recorded the installation time for a new app, which involves assignment of domains. The time interval measured in our experiment is that between the moment the `PackageManager` identifies the user's "install" click and when the `BackupManagerService` gets the Intent for the completion of installing an app with 3.06 MB. For Bluetooth, both the pairing and connection operations were timed. Among them, the pairing operation recorded starts from the moment it was triggered manually and ends when the `OnBondStateChanged` callback was invoked by the OS. For connection, we just looked at the execution time of `BluetoothSocket.connect`. Regarding SMS, we measure the time from when a SMS message is received (`processMessagePart`) to when the message is delivered to all the interested receivers and the process of querying the SMS content provider. The Internet-related overhead was simply found out from the network connection time.

The amount of time it takes to dispatch an NFC message is related to the status of the target app: when it was in the foreground, we measured the interval between `dispatchTag` and the completion of the `NfcRootActivity`; otherwise, our timer was stopped when `setForegroundDispatch` was called. For the Audio channel, we recorded the time for the call `AudioRecord.startRecording` to go through.

Results. The results of this evaluation are presented in Table V. As we can see from the table, the delays introduced by SEACAT are mostly negligible. Specifically, the overhead in the installation process caused by assigning domains to an app was found to be as low as 49.52 ms. Policy enforcement within different security hooks (with policy checks) happened almost instantly, with a delay sometimes even indistinguishable from the baseline. In particular, in the case of NFC, even when the unauthorized app with the `NFC` permission was running in the foreground, our implementation almost instantly found out its security context and denied its access request. The only operation that brings in a relatively high overhead is labeling an external device. It involves assigning a type to the resource, saving the label to `user_seres_contexts`, updating the DAC mapping table accordingly and even changing the DAC policy base to enable authorized apps' access to the resource when necessary. On average, those operations took 189.44 ms. Note that this is just a one-time cost, as long as the user does not change the type given to a resource. An exception is Audio, whose type is assigned whenever the dongle under protection is attached to the Audio jack. Note that the user only experiences this sub-second delay once per use of the accessory, which we believe is completely tolerable. In our results we report the absolute time needed to perform an operation, instead of providing the percentage difference with the baseline. Doing the latter is misleading in our case. Consider for example the operation `content://sms query()`. On the baseline it takes 2.7ms while the same operation costs 6.39ms on SEACAT. While this entails a 137% slowdown, it is way below a user perceivable delay [25], [15].

All the results presented here do not include the delay caused by human interventions: for example, the time the user takes to determine if an app or resource should be protected. Such a delay depends on human reaction and therefore is hard to measure. Also they only bring in a one-time cost, as subjects (apps) and objects (resources) only need to be labeled once. Actually, for NFC, our implementation could even remove the need for human intervention during policy enforcement: in the presence of two apps with the same `NFC` priority, the user could be asked to choose one of them to handle an NFC event whenever it happens, while under SEACAT, this interaction is avoided if one of the apps is assigned in the domain authorized to access the related NFC device and the other is not.

VI. Related Work

SEAndroid. Our approach is built on AOSP, on top of the partially integrated SEAndroid [30]. SEACAT leverages the existing AVC and policy engine for compliance checks over both MAC and DAC databases. By comparison, the current implementation of SEAndroid does not offer any protection for external resources: it neither can specify policies for this purpose, nor does it have the right hooks to enforce such policies. Particularly on the framework layer, the MMAC mechanism within SEAndroid can only achieve the control granularity at the permission level, a far cry from what is expected to mediate external resources.

An improvement on MMAC has been proposed recently [14], which, like SEACAT, also supports app-based policies and user-specified policies. Further, the way it controls content providers is similar to what we did when sanitizing the list of messages to let an app access only those it is allowed to read. Nevertheless, like SEAndroid, this prior work does not offer any means to control external resources either. It cannot label those devices, not to mention enforcing any policies. Also, the approach is designed as an alternative to SEAndroid, which comes with its own policy language and policy engine. By comparison, SEACAT is carefully designed to be a natural extension of AOSP to handle external resources.

External-device misbonding. This work is partially inspired by our prior research on Bluetooth misbonding problems [28] and work conducted by others on unauthorized screenshot taking [24]. Particularly, in prior work [28] we developed a security mechanism, called Dabinder, to offer a fine-grained control on Bluetooth devices. However, Dabinder is implemented on the framework layer, inside the Bluetooth service which could be bypassed by any app with native code. Native code can be used to talk directly to the Bluetooth stack and in general circumvent the framework protection. SEACAT works as an integrated part of SEAndroid in AOSP, which offers protection cross-layer, preventing unauthorized access to Linux devices. Specifically for Bluetooth, SEACAT enforces policies directly in the Bluetooth stack (see Figure 3), providing much stronger security guarantees. Also importantly, Dabinder is designed to be a DAC mechanism just for protecting Bluetooth devices, while SEACAT offers centralized protection that enforces both MAC and DAC policies, across multiple channels (Bluetooth,

TABLE V: Performance Measurements in milliseconds (ms). Confidence Interval (CI) given for confidence level=95%

AOSP				SEACAT				
Operation	**mean**	**stdev**	**CI**	**Operation**	**mean**	**stdev**	**CI**	**overhead (ms)**
install app	1415.6	40.61	±25.17	install app (label)	1465.2	76.07	±47.15	49.52
Bluetooth pairing	1136.5	351.65	±217.95	Bluetooth pairing (label)	1434.4	237.60	±147.26	279.9
BluetoothSocket.connect	1699.1	770.22	±477.38	BluetoothSocket.connect	1616	306.83	±190.17	-83.1
				BluetoothSocket.connect (block)	6	3	±1.86	-1693.1
dispatchTag	87.3	4.32	±2.68	dispatchTag(MAC:allow)	96.9	4.63	±2.87	9.6
				dispatchTag(MAC:block)	113.1	3.57	±2.21	25.8
				dispatchTag(label+allow)	358.28	40.47	±25.08	270.98
dispatchTag (foreground)	272	26.33	±16.32	dispatchTag(allow foreground)	269	41.53	±25.74	-3
				dispatchTag(deny foreground)	132.5	21.76	±13.49	-139.5
Ndef.writeNdefMessage(app A)	197.1	6.17	±3.82	Ndef. writeNdefMessage (DAC/MAC allow)	190.89	14.61	±9.06	-6.21
Ndef.writeNdefMessage(app B)	112.4	12.45	±7.72	Ndef. writeNdefMessage (un-labeled)	117.5	16.36	±10.14	5.1
SMS process message	94	7.3	±4.52	SMS process message(allow)	106.5	8.11	±5.03	12.5
				SMS process message (redi-rect)	154	12.11	±7.51	60
content://sms query() filter (10 messages)	2.7	1.1	±0.68	SMS query() filter	6.39	2.4	±1.49	3.69
Audio device connection	14.9	5.11	±3.17	Audio device connection (label+ connect)	177.6	21.92	±13.59	162.7
AudioRecord.startRecording (allow)	85.9	6.84	±4.24	AudioRecord.startRecording (allow)	95.6	16.75	±10.38	9.7
				AudioRecord.startRecording (block)	7.2	3.58	±2.22	-78.7

SMS, Internet, Audio and NFC) and its unified approach allows easy extension to new channels.

Enhancing Android security model. Android permission system has long been scrutinized and there is a line of research on enhancing this security model [27], [21], [18], [12], [16], [22], [19], [13]. Most related to our work is Porscha [29], which controls the content an app can access on a phone for digital rights management. For SMS messages, this has been done through sending an IBE encrypted message to a Porscha proxy on the phone, which further dispatches the message to authorized apps according to a set of policies. Porscha needs to make a substantial change to the SMS mechanism, adding the proxy to intercept incoming messages and a new field in MMS content provider for tagging messages. By comparison, SEACAT just places hooks within the existing mechanism, using SEAndroid for policy compliance check, and therefore is much easier to integrate into today's Android, and also offers both mandatory and discretionary protection across-layers.

VII. Conclusion and Future Work

In this paper, we present SEACAT, a new security system that enhances today's Android security model to protect external resources. SEACAT was designed to provide integrated security control through both MAC and DAC across different Android layers. More specifically, we utilize the same policy engine and AVC to support policy compliance checks on both MAC and DAC policy sets, which were extended for setting policies on external resources. Such checks are performed on the Android framework layer as well as the Linux kernel, within different security hooks placed there to control various channels (Bluetooth, SMS, Internet, Audio and NFC). DAC and MAC rules are enforced through the same security hooks. In the meantime, a misconfigured DAC policy will not cause the MAC rules to be circumvented. This new system provides phone manufacturers and system administrators means to define mandatory security policies. It also empowers ordinary Android users to specify their own rules to protect resources from third parties. SEACAT provides strong security guarantees, incurs a negligible performance overhead, is backward-compatible and in some cases it even improves users' experience.

The current design of SEACAT only manages the 5 most popular channels. A further step in this direction would extend our enforcement mechanism to other channels, such as Wireless and Infrared. SEACAT cannot provide MAC protection to Audio, due to the lack of identifiers for the devices attached to this channel. A solution could fingerprint different Audio devices through probing them to inspect their responses.

Acknowledgment

IU authors were supported in part by the NSF 1017782, 1117106, 1223477 and 1223495. This work was also partially supported by HHS 90TR0003-01 (SHARPS) and NSF 13-30491 (ThaW). The views expressed are those of the authors only.

References

[1] androidcentral.com set default nfc app. http://goo.gl/ZJj7Kf.

[2] androidforums.com making nfc app as default when scanning tag. http://goo.gl/jJtuUK.

[3] Chase official website. http://goo.gl/GHlt0Y.

[4] developers.android.com nfc basics. http://goo.gl/65ilBF.

[5] GooglePlay go locker app. http://goo.gl/s9CSTi.

[6] Jawbone official website. http://goo.gl/nt25Ub.

[7] SEACAT demos website. http://goo.gl/y61UBH.

[8] Square official website. http://goo.gl/oq3IqI.

[9] Square Security official website. http://goo.gl/Nij1v0.

[10] Viper official website. http://goo.gl/F7j0SE.

[11] Yahoo Mobile official website. http://goo.gl/qppn9v.

[12] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *HotMobile*, pages 49–54, New York, NY, USA, 2011. ACM.

[13] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on android. In *SPSM*, pages 51–62, New York, NY, USA, 2011. ACM.

[14] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *USENIX Security*, 2013.

[15] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *CHI*, pages 181–186, New York, NY, USA, 1991. ACM.

[16] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. X. Song. Contextual policy enforcement in android applications with permission event graphs. In *NDSS*. The Internet Society, 2013.

[17] B. Dwyer. Paypal here vs. square. http://goo.gl/wNGueP.

[18] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *NDSS*, Feb. 2011.

[19] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS*, pages 235–245, New York, NY, USA, 2009. ACM.

[20] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS*, pages 627–638, New York, NY, USA, 2011. ACM.

[21] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *CCS*, pages 639–652, New York, NY, USA, 2011. ACM.

[22] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-time enforcement of information-flow properties on Android (extended abstract). In *ESORICS*, pages 775–792. Springer, Sept. 2013.

[23] J. R. W. J. Joseph Tran, Rosanna Tran. Smartphone-based glucose monitors and applications in the management of diabetes: An overview of 10 salient "apps" and a novel smartphone-connected blood glucose monitor. http://goo.gl/GrvL6i, 2012.

[24] C.-C. Lin, H. Li, X. Zhou, and X. Wang. Screenmilker: How to milk your android screen for secrets. In *NDSS*, 2014.

[25] R. B. Miller. Response time in man-computer conversational transactions. In *AFIPS (Fall, part I)*, pages 267–277, New York, NY, USA, 1968. ACM.

[26] E. Mills. Researchers find avenues for fraud in square. http://goo.gl/Fq7CrR, 2011.

[27] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS*, pages 328–332, New York, NY, USA, 2010. ACM.

[28] M. Naveed, X. Zhou, S. Demetriou, X. Wang, and C. A. Gunter. Inside job: Understanding and mitigating the threat of external device mis-bonding on android. In *NDSS*, 2014.

[29] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy oriented secure content handling in android. In *ACSAC*, pages 221–230, New York, NY, USA, 2010. ACM.

[30] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, 2013.

[31] S. Stein. Withings wireless blood pressure monitor supports android/ios, now available. http://goo.gl/9ed0aP, 2014.

[32] K. Voss. Top 10 phone apps for home security. http://goo.gl/4MHK7A, 2014.

[33] N. Wanchoo. Fda approves mega electronic's android-based emotion ecg mobile monitor. http://goo.gl/wqU4oS, 2013.

[34] J. Wick. New interactive diabetes support tools manage mealtime insulin dosing. http://goo.gl/78RnDp, 2014.