# Using Component Isolation to Increase Trust in Mobile Devices

Temitope Oluwafemi

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2015

Reading Committee:

Tadayoshi Kohno, Co-Chair

Franziska Roesner, Co-Chair

Shwetak Patel

Howard Chizeck

Program Authorized to Offer Degree:

Electrical Engineering

University of Washington

## Abstract

Using Component Isolation to Increase Trust in Mobile Devices

Temitope Oluwafemi

Chair of the Supervisory Committee:
Associate Professor Tadayoshi Kohno, Co-Chair
Department of Computer Science and Engineering
Assistant Professor Franziska Roesner, Co-Chair
Department of Computer Science and Engineering

Mobile phones pervade virtually every realm of societal interactions ranging from the obvious, communications and mobile payments, to the less obvious use of health monitoring. They also vary in complexity and cost with various offerings including smart, feature or basic phones. But these ubiquitous devices come with security and privacy vulnerabilities that may leave users susceptible to data theft. These vulnerabilities include but are not limited to malicious third party applications, design flaws in operating systems, and risks unintentionally incurred by the device owner through ill-advised use. All of these threats put the user's data at risk of theft.

Motivated by the lack of trust in the way data is stored and processed on personal mobile devices and the potential for data to be mishandled by parties with whom they are shared, this dissertation examines:

- data leakage channels in modern mobile operating systems and the shortcomings in current approaches to address them, and
- the sharing of sensitive data via mobile devices and the shortcomings in current approaches to validate parties with whom data is shared.

Our work tackles the lack of trust in mobile devices and the systems that use them from two perspectives – (1) modifying phones by separating components to isolate data belonging to different entities and (2) adding new external hardware to separate trust among users.

To concretely explore (1), we study mechanisms to establish trust on mobile devices used for both work and personal purposes. The specific problem we consider relates to how proper isolation and security of both profiles on a single device can be achieved. We propose AppFork, an Android-based platform, which isolates and secures partitions belonging to work and personal profiles. We also address data leakage channels that were identified through the analysis of over 14,000 Android apps.

In exploring (2), we study how to establish trust as a prelude to sharing information with other parties over a phone call. We propose NoSSN, a solution that provides a privacy preserving two-way authentication scheme for parties communicating over the phone. NoSSN is a phone agnostic and portable solution that provides impersonation and replay resistance as security features. NoSSN is designed to be an additional piece of hardware (dongle) that can be attached to any mobile device through the audio jack, to provide secure and isolated profiles among users. We contribute two security protocols based on symmetric and public key crypto implementations in addition to designing and implementing an audio modem.

We describe the design, implementation and evaluation of both systems along with detailed discussions to highlight the application of this dissertation in practice. Both systems help lay the

foundation for the development of functional and marketable solutions to meet the demands of trust in mobile devices and associated systems. We envision these two solutions influencing the design of future mobile operating systems and usable accessory hardware that bolster trust within and external to the mobile device. Moreover, these works jointly contribute to the body of knowledge seeking to elevate the status of the mobile phone into a more trusted computing and communication platform.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGEMENTS

I would first like to thank God through Whom we live, move and have our being. This journey would never have been possible for me without Him.

I also would specially like to thank Prof. Tadayoshi Kohno for his wisdom and guidance through the various academic challenges I encountered. I would like to credit him for seeing potential in me and taking it upon himself to refine my very crude research skills. He has been patient, phenomenal, remarkable and such a blessing to me in my time of need, and I hope I can someday repay his faith in me.

Next, I would like to thank Assistant Prof. Franziska Roesner for her tenacious drive and insightful advice during the course of my doctorate degree. Her very welcoming attitude allowed me to settle in comfortably along with her very efficient and goal driven attitude which was at the very least, infectious.

I also would like to thank members of my committee for their direction and support through this journey. Additionally, I would like to thank several collaborators that contributed immensely to the success of this work. They include Oriana Riva (MSR), Suman Nath (MSR), Earlence Fernandes (University of Michigan) and Kiron Lebeck (University of Washington). I am also grateful to past and current members of the Security and Privacy Research Lab, who in various capacities have been instrumental to the success of this work. I also would like to thank Alexei Czeskis, Peter Ney, Ian Smith and Alex Takakuwa for their insightful feedback and advice in completing this work.

Lastly, I would like to thank my family and friends for their unrelenting support and prayers during this journey, which sometimes felt unending. They cheered me on and remained my number one supporters even at my lowest points. May God remember you for good.

# Chapter 1. INTRODUCTION

Over the past decade, we have seen a meteoric rise in the use of mobile devices in virtually every facet of life, ranging from the obvious – communications and mobile payments – to the less obvious – health monitoring. Mobile devices also come with varying levels of complexities and costs with offerings that include smart, feature or basic phones. Consumers use their mobile devices to store and utilize personal and work-related data, oftentimes carrying their most important and sensitive information in their pockets. With such ubiquity, mobile devices come with security and privacy vulnerabilities that may leave users susceptible to data theft. These include but are not limited to malicious third party applications, design flaws in operating systems, and risks unintentionally incurred by the device owner through ill-advised use. All of these threats put the user's data at risk of theft.

Despite mature mobile operating systems, leakage channels, i.e., avenues through which data may be leaked indiscriminately on a phone, remain a *serious* security concern. These systems are still striving to be trusted platforms but have not attained this due to such leakage channels. For example, some top free Android apps were shown to have leaked users' private contact lists – without their consent – to advertising companies [1]. Further, establishing trust with communicating parties via phone is a problem still not solved. *Even* with advancements in operating systems, there is still no sure way to establish trust with parties who you choose to share data with over a phone call. Examples of how the lack of trust with communicating parties can be detrimental is evident from [2] and [3] where unsuspecting international students were victims of phone scams due to the unverified identities of their scammers.

Thus, we argue that sensitive data can be misused by unverified communicating parties internal and external to the mobile device. Even if the operating system is trusted to handle sensitive data, we assert that data sharing with individuals who may be inadequately vetted, renders the security and privacy measures on such operating systems useless.

## 1.1 VISION

Our works seeks to

1. Isolate components on mobile operating systems as a way to enhance trust between different entities resident on a single device,
2. Find a more robust way to establish trust as a prelude to sharing data with parties we intend to communicate with.

In achieving this vision, we perform a detailed study of existing third-party applications on popular Android mobile operating system. This study serves as a precursor to understanding leakage channels in modern mobile operating systems. We use Android as a case study due to its open source implementation and its 78% share of the OS market share [4]. We will utilize these findings to inform the design of more robust operating systems which have more isolated components to deal with the lack of trust between different entities resident on the same device.

Moreover, we describe a novel way of using additional hardware, external to the device to separate and enhance trust between communicating users on a mobile device. We explore the use of such external hardware that is compatible with a wide variety of phones including landlines, smart, feature, basic and soft phones. Additionally, to achieve a robust and functional implementation, we examine the shortcomings of currently used authentication techniques in verifying the identities of communicating parties over the phone.

## 1.2 CONTRIBUTIONS

To concretely achieve (1), we designed AppFork, an Android-based platform which allows users to switch a single app from one active profile (e.g., work) to another without switching the active profile of all other apps. AppFork was motivated by the growing trend of employers allowing employees to use their personal mobile devices for work (Bring Your Own Device: BYOD) while simultaneously imposing strict security policies on these same devices. Such policies, e.g., a complete wipe of the device after a series of incorrect log-ins, protects the company's sensitive data, but can effect a user's privacy and lessen the user's control over his or her own data. In recent works in this area, we see partitioning of work and personal data through the use of virtualization techniques. However, virtualization has limitations, which include large overhead

on resource-constrained phones and a stipulation that all apps be in the same partition at any given time. AppFork as a solution overcomes these constraints and achieves the desired security goals of component isolation at a better level of performance and lower storage overhead when compared to state-of-the-art virtualization techniques currently in use.

To achieve (2), we designed NoSSN, a novel, privacy-preserving two-way authentication scheme that verifies the identity of both communicating parties on a phone call through a mobile device. NoSSN is a phone agnostic and portable solution that provides impersonation and replay resistance as security features. Through NoSSN, we contribute to two protocols optimized for low data rate communication via the audio channel on cellular networks. Our implemented work provides a prototype tested real-time over a major cellular network. The results present consumers and corporations with a secure and usable solution that ensures that identities of users can be validated without the exchange of personal sensitive data.

Together, AppFork (Chapter 3) and NoSSN (Chapter 4) help lay the foundation for the development of functional and marketable solutions to meet the demands of trust in mobile devices and associated systems. We envision that these two solutions will influence the design of future mobile operating systems and usable accessory hardware that bolster trust within and external to the mobile device. Moreover, these works jointly contribute to the body of knowledge seeking to elevate the status of the mobile phone into a more trusted computing and communication platform.

# Chapter 2. RELATED WORKS

In this Chapter, we give a high level overview of some related works in the contexts of component isolation within the mobile device and the use of additional hardware to establish trust between users interacting with devices or between devices interacting with each other.

## 2.1 COMPONENT ISOLATION IN MOBILE DEVICES

Here, we discuss some recent work in isolating components – specifically user profiles – in mobile devices to provide a trusted environment for different entities to co-exist. One such area where component isolation in mobile devices plays a dominant role is related to the "Bring Your Own Device" (BYOD) trend, where corporations allow their employees to access corporate data with their personal devices. In the following paragraphs, we study some related works in the BYOD application area.

MOSES [5] [6] which is a policy based framework that enforces software isolation of applications and data relies on taint tracking [7] at run time and only supports a global context switch from one domain to the other, i.e. an application cannot be switched independent of other applications from one domain to another. Other solutions like Divide [8] and Worklight [9] provide easy to deploy solutions but cannot support existing applications and are subject to privilege escalation attacks [10] [11] [12] [13].

Prior works, such as [14] [15] [16] [17], have explored the use of virtualization techniques on smartphones to provide isolated user spaces. However, these techniques come with significant performance and storage overheads and require the replication of application binaries across multiple virtual environments. Through replica installations of the same application across virtual phones, Cells [14], which is a lightweight OS-level virtualization architecture, provides support for multiple isolated environments running on the same physical smartphone. Foreground and background applications are also restricted to belong to the same domain or profile under virtualization techniques and user accounts, popular on tablet offerings, e.g., Microsoft Surface and Android 4.2 tablets.

Non-virtualization based techniques which include TLR [18] and TrustDroid [19], for instance, provide security frameworks for untrusted domain isolation. However, TrustDroid does not consider leaks through external storage and restricts an application to a single domain. To

avoid cross profile data leakages, TLR runs apps with sensitive data on Trusted Execution environments [20] [21] [22] [18] [23] using a Trusted Platform Module (TPM) and a trusted kernel.

## 2.2 Establishing Trust through Additional Hardware

We switch focus to exploring related works on establishing trust between communicating parties through the use of additional hardware. Parties could be users or devices and establishing trust in this context entails identity verification or authentication of the parties involved.

Authentication is an age-old problem that continues to be an active area of research. Early approaches for authentication have seen the use of fixed passwords that include Personal Identification Numbers (PINs). However, the use of fixed passwords are considered weak as they are time-invariant and once deciphered, can easily be circumvented. Additionally, due to the constraint levied on users to memorize fixed passwords, there is a tendency for users to reuse passwords across multiple services. Such reuse increases the chances of an attacker violating the authenticity of a user to a particular service through the compromise of another. Moreover, fixed password schemes are vulnerable to eavesdropping, replay, dictionary and brute-force attacks. Despite these shortcomings, various proposed alternatives struggle to replace passwords as the de-facto means for authentication as highlighted by Bonneau et al [24]. Some of the reasons for the lack of widespread adoption of password alternatives range from little to no improvement in security benefits to increased deployment costs [24].

Some password alternatives however provide significant improvement in security benefits which we highlight here. As examples, one-time password (OTP) schemes and more robust techniques based on challenge-response protocols have been proposed to bolster security. The use of challenge-response schemes, in particular, employs time-variant parameters to provide uniqueness guarantees during user authentication. Challenge-response protocols, however, reveal some partial information about the party being verified. This vulnerability affords adversarial verifiers the ability to probe for such information through the careful selection of challenges. Zero-knowledge protocols, on the other hand, address the issue with challenge-response protocols by allowing the claimant to provide proof of knowledge of a secret without revealing any information that can be used by an adversarial verifier. A detailed expose on how authentication is carried out is described in [25].

In the context of authentication and mobile devices, most literature have explored the use of mobile devices as a second factor during authentication as opposed to the use of additional, external hardware, to enhance trust on mobile devices. Hence, we claim that to the best of our knowledge, our work on the use of additional hardware to enhance trust on mobile devices presents a set of goals that are unique, and as a result, have not been explored in prior literature. However, we argue that we can still glean some learnings from works that have looked into the use of additional hardware to establish trust on any given device and in any given application area. We therefore discuss these works in the following paragraphs.

Similar to our goal of establishing trust as a prelude to sharing information between users on ubiquitous devices, Garriss et al [26] proposed a system that utilizes TPMs and support from x86 architecture to establish trust on public computing devices (kiosk). Their work focused on the use of personal mobile devices – as additional hardware – to address the lack of trust in using such kiosks before any sensitive task can be performed on them. Specifically, their proposed protocol determines the identity and integrity of all software installed on the public computing device and advises the user on the trustworthiness of the device.

Goodrich et al [27] proposed Loud and Clear (L&C), an approach that uses audio for human-assisted authentication for previously unassociated devices. L&C uses a text-to-speech (TTS) engine to vocalize an English-like sentence derived from the hash of a device's public key. Through their implementation, L&C provides a suitable and secure way to pair devices. In similar fashion, Silvester [28] invented the use of audio authentication initialization information as a backup method to pairing Bluetooth devices. Additionally, Czeskis et al [29] developed PhoneAuth, a system which provides second factor authentication for online services through a mobile device. The implementation of PhoneAuth is based on public key cryptography. Bauer et al in [30] described the design and implementation of Grey, a class of solutions that allows device owners to delegate authority to both physical and virtual resources through their smartphones.

The work of Li et al [31] demonstrates a method for resisting replay attacks via Dolphin, a proposed system which utilizes near field assertions through acoustic communications and sound power manipulation.

Lastly, we draw attention to the FIDO [32] Alliance whose goal is to reduce the reliance of user authentication via web pages on passwords. The FIDO Alliance has been instrumental in

developing the Universal Authentication Framework (UAF) protocol that provides a mechanism for a passwordless user experience on websites. Secondly, the FIDO alliance developed a Universal Second Factor (U2F) protocol where the security of online services is bolstered through the *use of additional hardware* for a strong second factor authentication of users. Both FIDO specifications are related to NoSSN in the use of additional hardware to increase trust during the authentication of communicating parties.

# Chapter 3. PER-APP PROFILES WITH APPFORK: THE SECURITY OF TWO PHONES WITH THE CONVENIENCE OF ONE[1]

## 3.1 INTRODUCTION

In this Chapter, we seek to address the lack of trust in mobile operating systems where data belonging to multiple entities – e.g., work and personal settings – co-exist. Specifically, we study how to utilize component isolation within the operating system to secure and isolate data and applications belonging to two different entities on the same phone.

As a case study for potential environments where such trust is needed, we will be looking specifically at the "Bring your own device" or BYOD paradigm where employers allow their employees to use their own personal devices, particularly smartphones and tablets, for work purposes [33] [34]. BYOD brings significant benefits to both the company and employees, including reduced equipment costs, improved employee engagement, and the convenience of carrying one dual-use device rather than a dedicated phone for each activity. Hence, the BYOD phenomenon appears to be here to stay.

Unfortunately, by using the same device for both work and personal activities, the user and the employer expose themselves to potential security and privacy risks [35] [36] [37] [38] [39]. A company's data is now stored and transmitted using devices and networks that the employer may not control. Applications (apps) on the phone may not all be controlled by the company and, in fact, could be untrustworthy or even malicious. Users may resist company-imposed policies, like data wipes after multiple unsuccessful attempts at unlocking the phone. Users may also worry about employers being able to mine personal data stored on their device, track their activities, and delete personal data unnecessarily [40]. In short, from a purely security and privacy perspective, there are important advantages – to both employers and employees – in having employees use separate phones for work and personal activities.

We seek to retain the benefits of BYOD (a single phone) while mitigating these types of security and privacy concerns. Toward addressing these concerns, recent work [14] [15] [16]

[1] An earlier version of this work appeared as a Microsoft Research Technical Report [57].

[17] suggests virtualization to partition a device into a business and personal workspace, such that work and personal data are isolated and can be governed by distinct security and privacy policies. We argue that these approaches are insufficient to meet users' needs. First, classical virtualization approaches come with a significant overhead for mobile platforms because they require duplicating the phone's operating system. Second, even lightweight virtualization, such as Cells [14], which replicates only the middleware, does not allow users to have both partitions running at the same time. As described in Section 3.2, we verified the importance of this requirement by surveying employees of a large IT company and the vast majority of respondents reported concurrently running both work and non-work apps at least a few times a day.

This Chapter presents the design, implementation and evaluation of AppFork, an Android-based platform designed to provide the security and privacy properties of two (or more) dedicated phones, but with a single phone. (We choose Android due to its popularity, but note that other phone platforms likely exhibit similar challenges.) From a security point of view, AppFork allows users to have on the same phone both a work and a personal profile, isolated and governed by profile-specific policies. From a functionality point of view, AppFork is designed to be compatible with existing unmodified apps and to enable the novel concept of a user profile that is per-app. Unlike user accounts, per-app profiles let users switch a single app from one active profile to another (e.g., from "work" to "personal") without switching all other apps' active profiles.

Although previous systems have considered our security goal of profile isolation (e.g., Cells [14], MOSES [5], TrustDroid [19]), to the best of our knowledge, AppFork is the first system to meet both the security and functionality goals described above. Enabling per-app profiles is especially challenging due to the difficulties involved in isolating profiles within an app and between apps. First, in addition to using the same phone for work and personal purposes, users often use the same app for both purposes – hence the need for isolation within the app. We accomplish this by creating partitions at the file system level and minimizing the number of files duplicated across profiles, thus keeping the storage overhead small. Second, although phone platforms provide app isolation (e.g., on Android, apps have private storage folders), there are many other channels that allow data to be shared across apps and therefore across profiles (e.g., apps running under different profiles can store data in a shared public directory).

To quantify these challenges, we first build ChannelCheck, a tool that performs both static and dynamic analysis to automatically identify cross-profile channels in an Android app. We present results from running ChannelCheck on more than 14,000 Android apps. Driven by this analysis, we implement a two-part solution to either automatically block those channels via AppFork, or to notify the user, the app, or the employer of an additional potential cross-profile violation via ChannelCheck. Our AppFork implementation ran successfully on all 24+ apps we tested in depth and was able to isolate channels that could lead to data leaks. Based on our implementation experiences, we outline additional recommendations to Android (and other mobile phone platforms) that, if adopted, could lead to even more robust AppFork-like systems.

Overall, this work makes the following contributions:

- We provide a concrete definition of the BYOD problem for modern smartphones and particularly the Android platform. We introduce the concept of per-app profiles and motivate their need via a user study.
- We create ChannelCheck, a tool for automatically identifying which channels an Android app uses to share data (and possibly leak sensitive data across profiles), and we quantify how common these channels are via an analysis of 14,000 Android apps.
- We design AppFork, which automatically blocks the most prominent cross-profile leakage channels found in our measurements. AppFork supports per-app profiles without requiring modifications to existing apps. We implement and evaluate AppFork as a modified version of Android, with a small storage overhead.
- We make recommendations to Android that, if adopted, would allow for the further integration of AppFork-like capabilities in Android.

## 3.2 MOTIVATION

We designed AppFork to enable owners of personal mobile devices, such as smartphones and tablets, to use the same physical device in different activities, such as work and personal (as in BYOD), with security and privacy properties comparable with having distinct devices for each activity. We set out to explore security and privacy in the context of user interactions on a single device with multiple personas and applications. Of more importance is the fact that, people share intimate details of their lives through their mobile devices which are also used for work

purposes. As a result, it is highly imperative to provide BYOD solutions that properly isolate and secure personal profiles from their work counterparts.

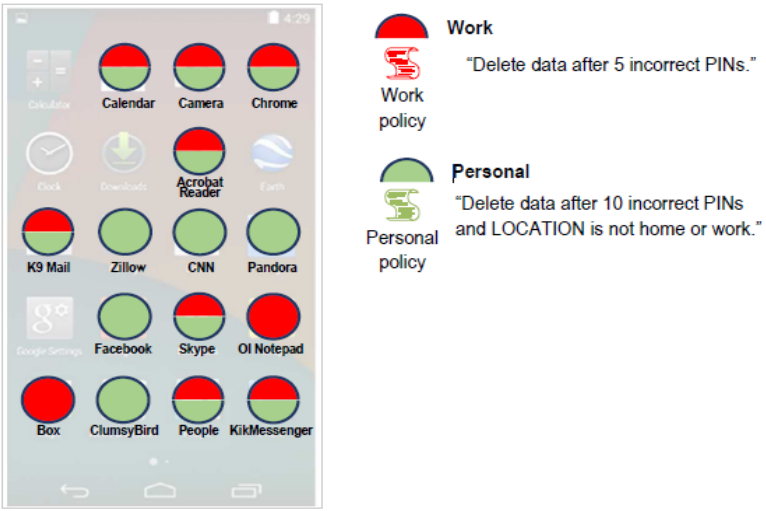


Figure 3.1. Examples of profiles with desired security policies

### 3.2.1 *Scenario*

We start by describing a BYOD scenario. Consider a software developer working for an IT company. Some of the apps installed on his smartphone are shown in Figure 3.1. We highlight apps that contain sensitive information and distinguish those used for work (in red) and/or for personal (in green) activities. His employer allows him to use his personal phone at work to access corporate data via a restricted set of approved apps. In addition, for security reasons, his phone must be PIN enabled and must implement a corporate policy wiping the phone after five consecutive incorrect PIN attempts. At work, he uses his phone mostly for emails (K9 Mail), calendar reminders, web browsing (Chrome), taking notes (OI Notepad), and accessing data in the cloud (Box). Occasionally, he uses the phone to take photos of whiteboard discussions or to chat with colleagues. However, while at work, he also uses apps without a direct work-related purpose. He uses K9 Mail not only to check work emails, but also for personal purposes. He uses Facebook and CNN for personal purposes, while other work apps are running at the same time. Though some of his apps can be uniquely associated with either work or personal activities, a significant number of apps, either for convenience or for efficiency, are multi-purpose (e.g., Calendar, Camera, K9 Mail, Acrobat Reader).

#### 3.2.1.1 Employer and User Requirements

In the context of this scenario, we described the users' and employers' requirements for a BYOD device, also summarized in Table 3.1. The advantages of being able to run work and personal apps on the same device (P1 in Table 3.1) and at the same time (P4-P6) are clear to our user, but he has two additional requirements.

First, he would like that his personal data, present in both single- and multi-purpose apps (P5), to be protected from access or deletion by his employer (P3, P7). Similarly, the employer wishes that the user's work-related data is not accidentally leaked via his use of personal apps (P8, P9). Data separation of this form is not a feature of modern phones. Unless apps explicitly build support for multiple isolated accounts, users cannot ensure their personal and work data are never commingled. This is difficult for multi-purpose apps, as well as apps that communicate via the phone platform's sharing channels (detailed in Section 3.4).

Table 3.1. Desirable BYOD properties supported by AppFork compared with state-of-the-art approaches

| **Desirable BYOD properties** | **One Phone** | **Two Phones** | **AppFork** |
|---|---|---|---|
| P1. User: Low pocket weight. | ✓ | ✗ | ✓ |
| P2. Employer: Wipe all corporate data after authentication failure. | ✓ | ✓ | ✓ |
| P3. User: Employer cannot wipe personal data. | ✗ | ✓ | ✓ |
| P4. User: Two profiles active at the same time. | ✗ | ✓ | ✓ |
| P5. User: Same app can run under two profiles. | ✗ | ✓ | ✓ |
| P6. User: Same app can be in use in two profiles at once. | ✗ | ✓ | ✗ |
| P7. User: Protect privacy from employer. | ✗ | ✓ | ✓ |
| P8: Employer/User: Avoid network leakage between profiles. | ✗ | ✗ | ✗ |
| P9: Employer: Control all apps that access corporate data. | ✗ | ✓ | ✓ |
| P10: Employer: Monitor apps running on the corporate network. | ✗ | ✓ | ✗ |

Second, as a father of two kids who often "borrow" his phone to play games, the user would like to protect his data in different ways, depending on whether the phone is at home, at work or elsewhere. For example, at home he would prefer that his personal data is not erased if his kids enter several wrong PINs (P3). Unfortunately, in current phone platforms, one policy rules all: a corporate security policy to wipe data applies to all apps and data with no distinction (P2). Instead, it should be possible to have distinct policies for work and personal data, to protect them according to different threat models.

From the employer's point of view, it is essential to prevent access to corporate data from malicious parties (P9) as well as ensure that personal apps cannot compromise corporate resources (e.g., a corporate wireless network). These security needs lead employers to enforce strict policies on their employees' devices (e.g., data wiping (P2)) and even monitoring apps' network activities (P10).

#### 3.2.1.2 User Survey

As our scenario already illustrates, personal and work apps often need to be run simultaneously, but still be isolated from a security and privacy point of view. We verified this hypothesis with a small anonymous survey conducted in a large IT company[2]. We surveyed 56 people (43 M and 13 F, age distribution 20-30 (8), 31-40 (26), 41-50 (18) and >50 (4)). We asked a total of 14 questions about their work and personal phone use. The survey did not explicitly mention BYOD until the very end, when we asked whether they had ever heard about BYOD (42 said "yes"). We summarized three key findings of this survey as follows:

- All participants used their personal phone for work purposes with 50 of 56 participants running apps needed for work and personal purposes.
- Only 9% of participants never run work and personal apps simultaneously on their phones.
- 29 out of 55[3] participants prefer to have work and personal data isolated from each other, while 15 of them were unsure and 11 said no.
    - The main reason for the "no" responses (5/11) was a matter of convenience.

[2] This work was done while at Microsoft Research and this survey was approved by the designated group responsible for reviewing human subjects studies.

[3] 55 instead of 56 because we excluded one participant who had no multi-purpose apps on his phone.

- Three options for isolating work and personal data were posed to the "yes" and "unsure" responses and the results are as follows:
  - 4 participants chose ***accounts*** – our baseline due to its familiarity.
  - 22 participants chose ***per-app switch***.[4]
  - 10 participants indicated both.

A larger scale study to ensure representativeness of the survey was outside the scope of this work. Nevertheless, the study highlights the value of an alternative to traditional per-use accounts and demonstrates the importance of multi-profile apps for users.

The desire among users for alternative ways to distinguish between personal and work data on their mobile devices led us to define specific goals and outline the threat model that our work would try to resolve. We cover such requirements in the following section.

## 3.3 Goals and Threat Model

AppFork was designed to provide the security and privacy properties of two (or more) phones, but with a single phone.

### 3.3.1 *Security Goals*

The primary security goals of AppFork were:

- *Profile isolation.* The apps and data associated with one profile should not interact, within the phone itself, with the apps and data associated with other profiles.
- *Per-profile policies.* Each profile may have its own security and privacy policies.

As an example of per-profile polices, AppFork may destroy all app data associated with the work profile after five incorrect PIN attempts (a policy some companies have today); all the app data associated with the personal profile may remain untouched.

Regarding our profile isolation goal, we noted that it was impossible to guarantee isolation between apps installed on different phones if those apps have network access: those apps may use the network (and remote servers) to communicate (P8 in Table 3.1). Thus, we limited our own goal to providing isolation within the phone itself.

[4] Described as "a switch attached to each app, so that you could selectively switch each app into work or personal mode. Using this solution, some apps could be running under your work profile while other apps could be running under your personal profile."

We additionally observed the challenge of side-channels: we could not rule out the future discovery of novel new side channels which could allow an app in one profile to infer information about apps in another profile. For example, researchers previously observed that background apps can use accelerometer data to infer private information about touch events in foreground apps [41]. Once discovered, it is possible to employ targeted mechanisms to mitigate known side channels, as the authors of the above-cited paper discuss. However, since it is impossible to predict what future side channels would arise, since the mitigation techniques could be ad hoc rather than principled and generalizable, and since side-channels are a potential concern for any single-phone solution to the BYOD problem, we viewed side channel prevention as out of scope of this paper. Moreover, we observed that side-channels can also arise in two-phone scenarios, e.g., hypothesizing extensions to [42] that use the microphone on one phone to extract information from the other phone.

In addition to the above security goals, users should be able to know which profile an app is running under, and be able to control the switching of an app between profiles. While it was our goal to ensure that the user (and not the app) was allowed to switch apps from one profile to another, an explicit non-goal for our prototype implementation was the elegance and effectiveness of the user interface.

### 3.3.2 *Functionality Goals*

We strove for a system that met not only the above security goals, but also the following functionality goals:

- Compatibility with existing apps. The system should work with existing apps whenever possible, without requiring changes to their code bases.
- Per-app profile switching. It should be possible for the user to switch an individual app's active profile without switching the active profile for all the apps on the phone.
- Low storage overhead. The storage overhead for using the system should be minimal.
- Low switching overhead. The overhead for switching an app's profile should be minimal.

Additional goals included being compatible with the existing Android architecture and understanding ways to modify Android (and other platforms) to facilitate even stronger security properties.

### 3.3.3 *Threat Model*

Our threat model was closely associated with our security goals above. An AppFork-equipped system has the following actors: the phone (including the OS and AppFork), the user, the profile owners, and the apps. We used profile owners to refer to the entities responsible for establishing the policies for different profiles. For example, the owner of the work profile might be the user's employer, and the owner of the personal one the user himself. Apps could be installed under one or more profiles. All parties, i.e., all profile owners and the phone's user, trust the AppFork system. This requirement is similar to the requirement today that these parties trust the phone hardware and underlying operating system.

Our threat model was guided by our goal to achieve the security of two phones. Thus, we considered out of scope any threats that were also not addressed by using two phones. For example, we assumed that profile owners trust device users not to be malicious. A malicious user, even with separate phones for each profile, could always maliciously leak information stored on one phone to another phone (e.g., by taking a picture of the first phone).

Similarly, if a profile owner (e.g., a business) approves the installation of an app under a profile (e.g., the work profile), then the profile owner trusts that app. To justify this trust, consider the following: if an app installed on the work profile is untrustworthy, it may be able to extract and leak information about the data of other apps installed on the work profile. But such an app could leak that information even if it was installed on a dedicated, work-issued phone (e.g., via the network). Hence, an AppFork profile owner must trust the apps he installs (or approves for installation) under that profile, just as he must trust the apps he would install on a dedicated phone.

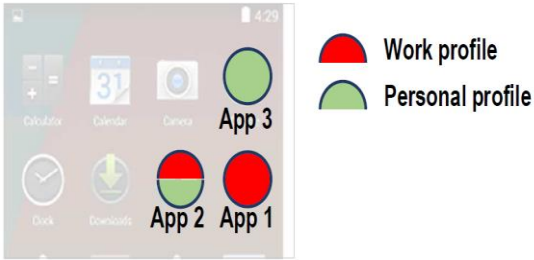


Figure 3.2. The owner of "work" approves App1 and App2, but does not trust App3 approved in "personal"

A profile owner may not, however, trust apps installed under other profiles (P7 and P9 in Table 3.1). Consider the scenario depicted in Figure 3.2, in which App1 and App2 are installed in the work profile, and App2 and App3 are installed in the personal profile. The owner of the work profile may not trust App3. Indeed, if all three apps were installed on the same conventional (non-AppFork) phone, App3 might seek to extract and exfiltrate work-related data from App2. Hence, our profile isolation security goal above was critical: the work profile data must be isolated from App3.

In the dedicated-phone case, a profile owner who approves the installation of App1 and App2 on a phone must trust App1 and App2. However, we observed that in the dedicated phone case, the profile owner would not evaluate the risks associated with App2 also being used in other profiles. Hence, we did not require the profile owner to trust that App2 will correctly handle data spanning multiple profiles; our profile isolation security goal is designed to address this issue.

#### 3.3.3.1 Goals in context.

Other works have addressed goals similar to ours, but to our knowledge we are the first to consider all these goals simultaneously. Traditional virtualization (that duplicates the phone OS) or systems like MOSES [5] satisfy the security goals of AppFork, but not its functional goals. Lightweight virtualization, such as Cells [14], and security frameworks for domain isolation (e.g., TrustDroid [19]) also satisfy the security goals of AppFork with a smaller storage overhead, but they still do not address the functionality goals of per-app profiles and low-switching overhead.

Finally, it is important to note that it was not a goal of AppFork to make Android apps more secure. We assumed profile owners trust the apps that they approve for their profile. We assumed such an approval implies verifying that apps are not malicious (or accepting the risks) and ensuring the Android platform is used as per guidelines. This assumption is the same assumption used with today's solutions to the BYOD problem as well; if a large IT company approves the use of applications for employees' mobile phones, and if one of those applications proves to be untrustworthy, then that application can cause harm or leak data.

## 3.4 RELATED WORKS

Some recent work on virtualization has been successfully applied to smartphones [14] [15] [16] [17]. "Classical virtualization" approaches require duplicating the phone OS and the Android stack, a challenge on resource-constrained mobile devices [19] [43]; optimizations can reduce this overhead. For example, Cells [14] is a lightweight OS-level virtualization architecture that enables multiple virtual phones to run on the same physical smartphone, enabling virtual work and personal phones. In contrast to AppFork, applications are duplicated across virtual phones with a clear overhead with such an approach. Moreover, foreground and background applications must belong to the same domain (profile). User accounts, recently introduced on tablets (e.g., Microsoft Surface and Android 4.2 tablets) suffer from the same problems.

Other (non-virtualization-based) security frameworks exist for untrusted domain isolation. Examples include TLR [18] and TrustDroid [19]. We could apply this approach to the BYOD context, although it is unclear whether the trusted domain is work or personal – in fact, both profiles have sensitive information to hide from each other. Moreover, unlike AppFork, TrustDroid does not consider leaks through external storage, and allows an application to belong only to one domain.

Previous work on building trusted execution environments [20] [21] [22] [18] [23] typically rely on a Trusted Platform Module (TPM) and a trusted kernel. The Trusted Language Runtime (TLR) architecture [18] [23] is such a system that aims (unlike others) to also be easy to program. In the BYOD context, one could use TLR to run apps with sensitive data and avoid leakage to another profile. While extremely secure, solutions like TLR are not likely to scale for BYOD applications: they would require rewriting all apps, and they are not designed to run entire apps within the trustbox.

MOSES [5] [6], a policy-based framework for enforcing software isolation of applications and data, is closely related to AppFork. Unlike AppFork, MOSES does not support per-application profiles and requires heavyweight taint tracking [7] at run time.

Finally, Divide [8] provides a workspace application that supports work-related functions like calendar, email and office-like applications. Worklight [9] provides a secure SDK for developing applications. Such solutions are easy to deploy, but they cannot support existing unmodified applications. Moreover, their application isolation relies on the Android permission

framework, which, as we discussed in Section 3.4, is not sufficient, and is subject to attacks such as privilege escalation [10] [11] [12] [13].

## 3.5 Cross-Profile Isolation

AppFork sought to enable per-app profiles in unmodified existing apps. Because we wished to allow a single app to support and switch between multiple profiles, we faced two design challenges: (1) isolating profiles within an app, and (2) isolating profiles between apps. For example, an email client like K9 Mail, when running under the work profile should not – even accidentally – leak work data to the K9 Mail personal profile, or to any other application running in personal mode.

We designed AppFork for Android due to its popularity. Here, we discuss in more detail why and how the above challenges arise in Android and the system's available cross profile communication channels.

### 3.5.1 *Android Background*

Android isolates apps from each other by running them with separate Linux user identifiers (UIDs), and restricts their access to system resources and other apps by requiring that they request specific permissions at installation time. Further, an app's files are stored in a private folder accessible only to that app (unless the app's UID is shared with other apps). Thus, stock Android already provides some level of isolation. However, there are still many ways in which cross profile communication can occur.

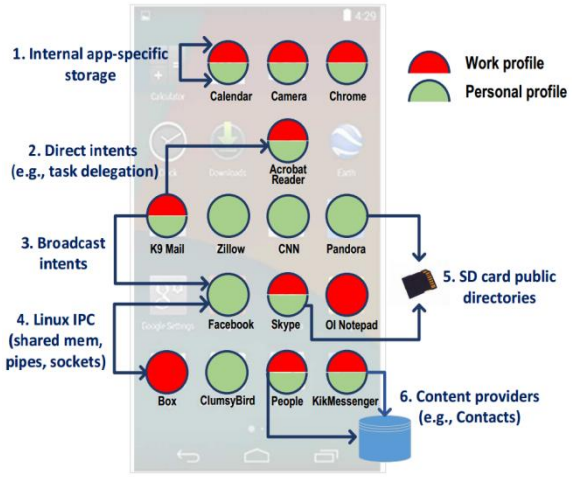


Figure 3.3. Cross-profile data sharing channels in Android (network and side channels are not shown)

#### 3.5.1.1 Explicit Communication Channels

Figure 3.3 summarizes key channels in which Android apps might share data. As discussed in Section 3.2, we did not consider network communication since the network is also available to apps running on two separate phones. Channels of type 1 usually apply to individual apps that span multiple profiles (within-app channels). In reality, they can also allow communication between different apps that share the same UID, but, as shown later, this is relatively rare. Channels of type 2-6 apply both to single apps running in multiple profiles (within-app channels) and multiple apps running in different profiles (between-apps channels). We elaborate on these channels below.

#### 3.5.1.2 Inter-Component Communication

Android apps consist of components, including Activities, Services, Content Providers and Broadcast Receivers. Components can communicate in many ways:

- Direct intents: One Activity or Service can launch another using a direct Android Intent. Intents can be used for task delegation, e.g., K9 Mail uses an Intent to launch Acrobat Reader to open an attachment. They can also be used to set up communication sessions between components, i.e., by binding to a Service, which can expose an AIDL (Android Interface Definition Language) interface.

- Broadcast intents: Apps may also send and receive Broadcast Intents. Broadcasts may originate from the system (e.g., notifying the device's screen is off) or from apps, and they are delivered to each registered receiver.
- Content providers: Content providers handle shared sets of data like SMSs or contacts. Apps can use built-in content providers or expose their own custom content providers. Two apps (or two profiles of the same app) can communicate by one writing to a content provider and the other reading from it.

#### 3.5.1.3 External Storage

Apps can share data by writing to world readable locations on external storage (the SD card). Prior to Android 4.4, all files on external storage were accessible to any app with the READ_EXTERNAL_STORAGE permission. Starting with Android 4.4, external storage is structured like internal storage, using app-specific directories accessible only to that app. However, apps in Android 4.4 can still share data via the SD card through public shared directories, such as Music.

#### 3.5.1.4 Linux IPC

Apps can also communicate via standard Linux inter-process communication methods. Android offers Java APIs for Linux IPC (android.os.MemoryFile and android.net.LocalSocket) in addition to native Linux IPC. As shown later, these methods are rarely used in Android apps, which can instead achieve the same goals using more efficient ICC methods.

#### 3.5.1.5 Side-Channel Capability

As discussed in Section 3.3, we focus on explicit data sharing channels and not side-channels. Nevertheless, we consider such possible attacks. Prior work has demonstrated that System Services, such as SensorService, WiFiManager or AudioManager, can be used as covert channels [13]. In addition, as we noted in Section 3.3, for accelerometer data, sensors can be used by an application to acquire information about another [41].

### 3.5.2 *ChannelCheck: Detecting Channel Use*

Above, we analyzed the possible cross-profile communication channels available to Android applications. To better understand how frequently these channels are used in real apps, and thus

to ultimately inform our design and implementation of AppFork, we built a tool to detect their use in existing apps. This tool, ChannelCheck, performs static and dynamic analysis on an app's binary in order to identify whether and how the app uses all channels described above. ChannelCheck detects the use of both explicit communication channels and side-channels, but excludes the network channel for reasons given above.

ChannelCheck's static analysis involves processing every method call site and looking for the presence of Java APIs related to system services, Java bindings to Linux IPC, built-in content providers, custom content providers, access to SD card, and sensor services.[5] Moreover, ChannelCheck reports whether an app shares the same UID with any app of a given set of apps, implying the app's data is shared with those apps.

For dynamic analysis, ChannelCheck executes a given app and traces, using a kernel mode tracer, calls to APIs for exchange and broadcast of intents, filesystem accesses to data stored on external storage, and Linux IPC attempts (including sockets, pipes and Android custom shared memory[6] ). The kernel mode tracer is context-sensitive, i.e. the tracer detects when the app is executing its own native code, and switches on tracing automatically. This reduces Linux IPC false positives arising from support libraries.

To automate the dynamic analysis (and be able to run it for many thousand apps), ChannelCheck relies on an existing automation framework called PUMA [44] that automatically runs an app and navigates to various pages of the app by emulating user interaction (e.g., by clicking a button, swiping a page, etc.). PUMA can be configured to explore all structurally distinct pages in an app, with a certain timeout (or a bound on the number of app interactions to perform).

[5] For content providers, ChannelCheck detects use of methods for adding or retrieving data from built-in content providers (as specified in the android.provider package) and use of APIs that must be implemented by custom content providers. For sensors, it detects use of the getSystemService() method classified based on the SENSOR_SERVICE specified in the argument, for all 13 sensor types available in Android.

[6] More precisely, we trace system calls for socket (unixdomain), connect (unixdomain), bind (unixdomain), pipe, pipe2, msgget, semget, shmget, ioctl (ashmem), mknod (fifo) and mknodat (fifo).

Table 3.2. Use of explicit cross-profile communication channels in existing Android apps based on static and dynamic analysis of 14,067 apps. The rightmost column indicates whether AppFork can automatically prevent cross-profile leakage via that channel, as described in Section 3.6: Design and Implementation.

| **Explicit communication channel** | **Num of apps** | **AppFork** |
|---|---|---|
| Direct intents | 21.1% (2994) | ✓ |
| Broadcast intents | 14.8% (2100) | ✓ |
| Built-in content providers | 14.3% (2005) | ✓ |
| Custom content providers | 8.7% (1222) | × |
| SDcard access | 49.3% (6937) | × |
| SDcard - only app-specific paths | 8.4% (1183) | ✓ |
| SDcard - public shared directory | 31.3% (4399) | × |
| Linux IPC | 0.5% (69) | × |
| Shared UIDs | 0.9% (120) | × |
| None | 44.1% (6200) | ✓ |

### 3.5.3 *Measurement Study*

To assess the relative risk posed by the channels described above we used ChannelCheck with 14,234 Android apps. Apps were crawled from the Google Play Store during the third week of December 2013, and were listed as the 500 most popular free apps in each category provided by the store. We configured PUMA to explore all pages in an app with a timeout of 3 minutes. The tool successfully ran with 14,067 apps. Static analysis failed for 146 apps (1.0% failure rate) due to APK decode and unzip errors, and dynamic analysis failed for 19 apps (0.1% failure rate) due to Android or PUMA crashes.

Table 3.2 shows the fraction of apps communicating with other apps through one of the explicit cross-profile channels. A large fraction (49%) of the apps use external storage. 8% of the apps access external storage only at app-specific paths, but we expect with the changes introduced in Android 4.4 (Section 3.5.1.3) more apps will use app-specific paths in the future. On the other hand, we found 31% of the apps using public shared directories on the SD card. Intents are also largely used with 21% of the apps using direct intents and 15% broadcast intents. Access to content providers is also relatively common: 14% of the apps use built-in content

providers, with roughly half of them accessing at least one of the three most popular content providers: Settings, Contacts and Calendar. 9% of the apps use custom content providers. Linux IPC was found only in 0.5% of the apps (in fact developers are encouraged to use instead more efficient ICC methods provided by the framework), and shared UIDs were present in less than 1% of the apps. Finally, 44% of the apps do not use any of these communication channels.

Table 3.3 reports on the frequency with which applications use functionality that may be used as side-channels. These numbers represent an upper bound on the side-channel risk through these capabilities: the use of such a feature (e.g., system services or sensors) does not necessarily mean that the app is using this channel to leak cross-profile information. In fact, in the majority of cases, we expect that this is not the case; nevertheless, we report these numbers for completeness. We find that 44% of the apps do not use any functionality associated with a known side-channel.

Table 3.3. Presence of potential side-channel capabilities in existing Android apps, based on static and dynamic analysis of 14,067 Android apps. Note that an app's use of a potential side channel does not necessarily mean that it is using that channel to actually leak information; in other words, the numbers in this table present an upper bound on the risk of side-channels. For sensors, we reported results only for those that were used in at least 0.1% of apps.

| **Side-channel capability** | **Num of apps** |
|---|---|
| System services | 56.2% (7899) |
| Sensors – accelerometer | 2.6% (363) |
| Sensors – gyro | 0.1% (13) |
| Sensors – light | 0.1% (17) |
| Sensors – magnetic field | 0.2% (22) |
| Sensors – orientation | 0.4% (60) |
| Sensors – proximity | 0.4% (63) |
| None | 43.8% (6167) |

We found that the numbers in Table 3.2 and Table 3.3 are roughly the same if we consider only the apps in Business and Communication categories, which are the two most relevant categories for BYOD. The only major difference is the larger adoption of content providers and system

services: 37.6% of these apps use built-in content providers, 14.5% use custom content providers, and 66.5% use system services.

These measurements help inform and validate our design of AppFork, discussed below. However, we also believe that these results are of independent interest to researchers studying the BYOD problem or other Android topics.

## 3.6 Design and Implementation

### 3.6.1 *AppFork Overview*

We designed AppFork with the goal of supporting existing apps with no modifications. We built AppFork on Android by modifying the Android Application Framework and provided an app for end users to manage their profiles.

Figure 3.4 shows the AppFork architecture. AppFork is implemented as an Android system service. This service, which starts when the phone boots, has three main components: the Profile Manager for managing user profiles and enforcing storage isolation, the Cross Profile Filter for preventing apps from leaking data across profiles, and the Policy Enforcer for implementing the specifications of all profile policies, by monitoring policy conditions through Monitors and executing policy actions through Actuators.

Peeking into details of our implementation, we restricted access to the AppFork service to processes with uid set to android.uid.system. This means that third-party apps could not access this service and, for instance, arbitrarily change an app's profile. Instead, only first-party apps, if granted system privileges, could invoke the AppFork service API. We signed our AppFork app with the platform key of our Android custom build so that it inherited system privileges.

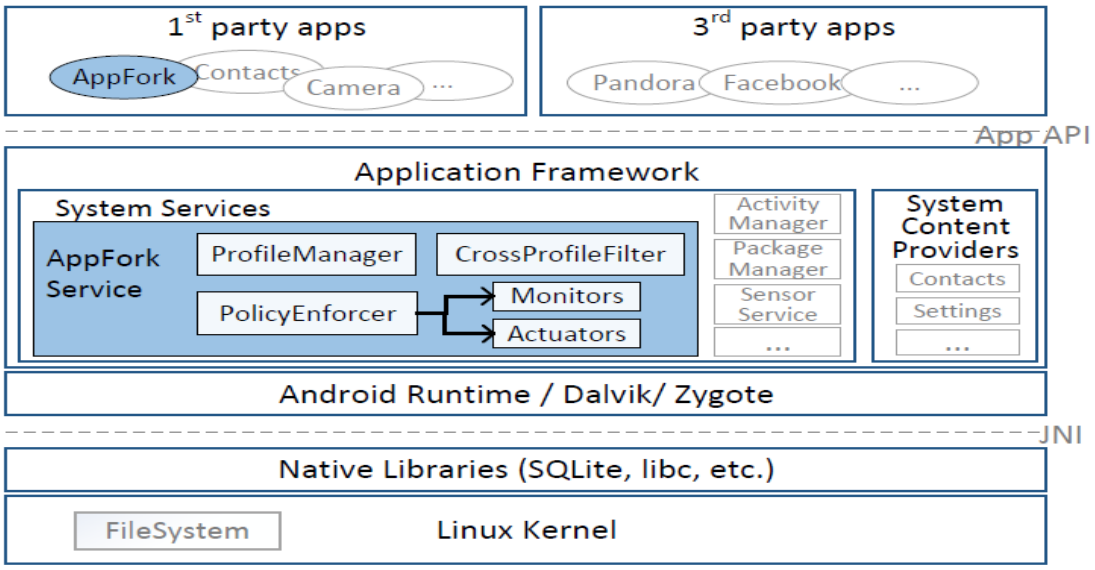


Figure 3.4. AppFork's architecture

In AppFork, a profile consists of a policy and a set of apps that are allowed to run under the profile. An app can be associated with multiple profiles. AppFork profiles are different from user accounts because they are activated/deactivated on a per-app basis.

The profile owner determines the policy and the set of apps allowed in that profile. AppFork stores this information in the file system (with access restricted to system processes). A profile owner may be the owner of the device or an external actor, such as an employer. In the latter case, the employer determines the policy and provides a list of preapproved apps that the device owner can selectively install. As discussed in Section 3.3.3, a profile owner approves apps for that profile, and AppFork trusts those apps within the scope of that profile (i.e., it is not AppFork's responsibility to block malicious apps approved by an employer); this is akin to trusting the work-installed apps on a work-issued phone. A profile's policy applies only to the apps in that profile. For instance, a work policy like the one in Figure 3.1 wipes only work-related data and apps after five consecutive failed logins.

Each time a user changes an app's active profile, AppFork checks whether the app is currently running and, if so, stops the app and all associated background processes. The app's profile is switched (more details on this below) and the app is started in the new profile.

### 3.6.2 *Profile Partitions*

AppFork maintains a separate storage partition for each profile, ensuring only data belonging to that profile is stored in that partition and is not accessible to other profiles.

Files saved to the internal storage are by default private to the app, and stored at the path /data/data/<packagename>. When the app is first installed, AppFork creates a partition for a "default" profile by moving the content of the app's original folder to /data/data/<packagename>-default and creating a symbolic link with a path of /data/data/<packagename> pointing to this folder. Suppose this app is added to both the "work" and "personal" profiles. The first time the app is switched into one of the profiles, AppFork dynamically creates a storage partition for each profile, at location /data/data/<packagename>-work if the profile is "work" or /data/data/<packagename>-personal if the profile is "personal". These newly created folders have the same structure as the "default" profile's folder, except that AppFork creates symbolic links in them to point to the "default" lib subfolder located at /data/data/<packagename>-default/lib, which contains the app's precompiled libraries. With symbolic links, the lib folder is never replicated, minimizing AppFork's storage overhead.

When the user starts or switches an app into a given profile, AppFork creates a symbolic link in the original app folder /data/data/<packagename> pointing to the partition of the active profile. The file system permissions are set so that the folder of the active profile is accessible by processes with the app's uid, while folders containing inactive profiles have android.uid.system permissions. Thus, an approved app running in profile "personal" cannot maliciously access files within the "work" partition, even if it is aware of the symbolic link switch. This approach provides isolation for all file system operations, which includes apps' SQLite databases, since they are stored in the same app-specific folders.

Note that our approach based on symbolic links can possibly generate many replicated files across profiles. A more advanced solution is to use a copy-on-write approach in which symbolic links are maintained for previously unmodified or static files resident in their app's original folder. Files are copied into the appropriate partition only if a write is scheduled from any of the profiles. This solution, however, increases the implementation complexity and potentially the processing overhead, as it requires keeping track of all write operations. In the evaluation, we compare AppFork's storage overhead against this optimized implementation.

Android apps with the READ_EXTERNAL_STORAGE or WRITE_EXTERNAL_STORAGE permissions can read or write files in external storage (the SD card). Files saved here are world-readable, so accessible to any app with such permissions. Our solution to provide isolation for external storage builds on two observations. First, as per Android guidelines, external storage offers minimal protection for stored data, hence apps should not store sensitive data here, but instead in the app-private directories which can be effectively protected. Second, starting with Android 4.4, external storage is structured like internal storage, with package specific directories such that apps can access their private partitions (e.g., /sdcard/Android/data/<packagename>) without holding the broad EXTERNAL_STORAGE permission.

We assume apps will follow the above guideline of using app-specific directories on external storage and not request the EXTERNAL_STORAGE permission. Then, AppFork can use a partitioning approach similar to that of internal storage, except that symbolic links cannot be created in external storage due to the vfat partition. At a profile switch, AppFork changes the name of the resident folder, /sdcard/Android/data/<packagename>, used in the previous profile to either /sdcard/Android/data/<packagename>-work or /sdcard/Android/data/<packagename>-personal, depending on the profile. However, AppFork does not isolate profile data stored in shared public directories on external storage, such as Music, Pictures, and Ringtones. Since these directories can be essential for some apps (e.g., to avoid a huge storage overhead or to simplify their syncing strategy), we do allow them. It is up to profile owners (e.g., corporations) to verify that approved applications comply with this policy and the Android guidelines for sensitive data. Recall AppFork's goal of approximating two phones with a single phone – since external storage is world-readable, profile owners may not approve an application violating these guidelines for sensitive data even on a dedicated phone for each profile. ChannelCheck logs how an app uses external storage and can support this assessment.

### 3.6.3 *Cross-Profile Isolation*

Android apps can share data across profiles in several ways (see Section 3.5). Table 3.2 summarizes which channels AppFork automatically blocks: we address potential leakages through ICC (discussed below), internal and external storage (discussed above), and built-in content providers (discussed below); we do not address the use of Linux IPC since it is rarely used in Android apps and the same functionality can be achieved with more lightweight ICC, and

also because ChannelCheck can identify applications that use IPC; we do not address the use of networks to leak data across profiles as this is also possible with the use of two dedicated phones (see Section 3.3.3)

#### 3.6.3.1 Direct Intents

Android apps are allowed to start other apps or services through respective calls to either startActivity or startService (bindService). Additionally, Android allows apps to delegate tasks to other apps through calls to startActivityForResult. These features are facilitated by Android's Intent class. While useful, these features pose security and privacy risks at odds with AppFork's goal of cross-profile isolation. For example, we found that a Book Catalogue app delegates scanning of barcodes to a Barcode Scanner app. If Book Catalogue runs under one profile, it may leak information to the Barcode Scanner, which might maintain a record of all scanned barcodes irrespective of Book Catalogue's current profile.

A solution to the potential cross-profile leakage threat in the case of task delegation is non-trivial. Consider the example shown in Figure 3.1, in which the user runs K9 Mail and AcrobatReader, both allowed in both profiles. K9 Mail may delegate opening an attachment to AcrobatReader, currently running under the other profile. To prevent data leakage across profiles, we envision at least four options which can be implemented at the system level: (1) The calling app cannot arbitrarily force another app to switch its current profile, so it has to wait for a timeout to expire or for the needed app to end; (2) the calling app has the right to force the called app to switch profile such that it can be used immediately; (3) the called app switches profile only after the user is prompted with a dialog and approves the switch; (4) the request of the calling app is rejected and a SecurityException is thrown or a friendlier failed status is returned to the calling app.

The first three options can lead to a denial-of-service attack: a malicious app running in the background may continuously invoke Acrobat Reader and prevent other profiles from using the app. Even in the case of the third option, the user, unaware of what is happening, may keep approving the profile switch. Another drawback of the second approach is that it is not immediately clear what profile should be given precedence, and a drawback of the third approach is that dialogs are disruptive to users.

For these reasons, we argue that this class of conflict is better resolved by taking the apps' semantics into account. In fact, whether the profile switch should be automatically authorized depends on how trusted the apps are (e.g., first party apps may be able to force a switch) and on the type of task (e.g., another app for viewing PDF files may be available for use instead). Our current solution is based on the fourth approach described above, in which the calling app receives a SecurityException thrown by AppFork from within the startActivityLocked member function of the ActivityStack class. This approach builds on the assumption that apps that delegate tasks to other apps should already be prepared to handle such exceptions, in the event that the needed app is unavailable.

For unresolved intents that result in Android's "chooser" activity, we modified the onCreate and rebuildList functions of the ResolverActivity and ResolveListAdapter classes respectively, to only display apps approved under the active profile of the intent creator.

Finally, we also ensure that app components cannot bind to services running under different profiles, with the exception of critical system services (e.g., Location Manager, Account Manager, Power Manager). This is implemented by intercepting calls to startServiceLocked and bindService of ActivityManagerService, where we deny requests to start or bind to services across profile boundaries.

#### 3.6.3.2 Broadcast Intents

Android apps may also send broadcast intents, which are delivered to all subscribed receivers (possibly subject to a predefined permission). Cross-profile data leakage can happen if a trusted app in one profile sends sensitive information to receivers in apps under a different profile. A data leak can even occur through subscriber registration because upon successful registration, the last available sticky broadcast is automatically sent to the new broadcast receiver. AppFork resolves this potential threat by filtering out registered or registering receivers with active profiles that are different from the one of the sending app. We specifically modified the broadcastIntentLocked and registerReceiver member functions of the ActivityManagerService class.

#### 3.6.3.3 Content Providers

Android apps can also share data through built-in and custom content providers. For built-in content providers like the Contacts provider, AppFork enforces a logical partition of the

provider's database. Specifically, we modified the getDatabaseLocked API of the SQLiteOpenHelper class to fork and control access to the appropriate databases for each profile. At a profile switch, AppFork forces a switch of the database to the one belonging to the active profile, for any database function specified in the ContentProvider class.

For custom content providers, we cannot modify the corresponding ContentProvider classes (per our requirement of supporting unmodified apps) so we take a different approach. AppFork checks whether the calling app and the owner of the custom content provider are within the same profile (this happens by instrumenting the acquireProvider and acquireExistingProvider APIs of the ActivityThread class). If they belong to different profiles, a null reference is returned. Otherwise, the requested provider is returned. This solution provides isolation at the cost of making custom providers available only in one profile at the time.[7]

### 3.6.4 *Policy Specification and Enforcement*

Profiles are specified in XML and stored on the device. They are currently not encrypted, but could be in the future. We provide a simple template that can be extended as more policy constructors are introduced. Each profile specification consists of two parts: a list of packages approved for use under that profile and a policy. Each policy consists of one or more conditions to be monitored and one or more actions to be executed if those conditions are detected. Figure 3.5 shows an example of a "work" and of a "personal" profile. The "work" policy specifies that after five consecutive failed logins, data belonging to that profile must be wiped; the "personal" policy specifies that apps requesting the TYPE_ACCELEROMETER resource should be denied access.

Each time the phone boots or new profiles are created, the policy specifications are parsed. The map of all supported apps approved under the profile is stored in memory, and each policy is translated into subscriptions to policy monitors. As an example, we describe the monitors and actuators we implemented for the policies shown in Figure 3.5.

For the "work" policy, we implemented a Password Monitor that keeps track of incorrect password entries. We modified the reportFailedPasswordAttempt() method of the DevicePolicyManagerService class to send a sticky broadcast with information about the number

[7] Because the support for custom content providers comes with this limitation, in Table 3.2 we list them as not supported by AppFork.

of incorrect password entries each time a wrong password is entered. If the maximum number of wrong attempts is reached, the Wipe off Actuator is invoked to erase all profile data. Once the profile has been deleted, a notification is sent to the other components and the AppFork app to reflect the changes.

```
<profile name="work">
  <packages>
   <approved name="com.google.android.gm">
   <approved name="com.mobisystems.office">
  </packages>
  <policy>
   <sensor name="failed-login" maxOccurs="5"/>
   <actuator name="wipe-profile"/>
  </policy>
 </profile>
 <profile name="personal">
  <packages>
   <approved name="com.facebook.katana">
   <approved name="com.google.android.gm">
  </packages>
  <policy>
   <sensor name="access-resource" value="
        TYPE_ACCELEROMETER"/>
   <actuator name="block-access"/>
  </policy>
 </profile>
```

Figure 3.5. Examples of AppFork profile specifications

For the “personal” policy, we implemented a Blacklisted Resources Monitor that keeps track of apps’ requests for device resources, particularly sensors such as proximity, accelerometers, and light. We modified the ContextImpl, SensorManager and SystemSensorManager classes to monitor apps’ access to device sensors. If access is requested, the Resource Block Actuator grants or denies access depending on the policy. For simplicity, if access has to be denied it filters out the app’s subscriptions for sensor readings. This approach prevents apps from crashing as opposed to if their requests were out rightly rejected. We envision other more advanced implementations where the sensor readings could be returned but in an obfuscated or generalized manner as proposed in [45].

Policy Enforcer is designed in a modular fashion such that new monitors and actuators can be easily plugged in. Additional monitors can cover important contextual information, such as home or work location, battery level, or WiFi network information. Additional actuators can provide a more comprehensive set of actions, such as blocking network traffic, switching network radio, and backing up data to the cloud. For instance, we envision policies such as “if at work and using the corporate WiFi network, use the cellular network for transmitting ‘personal’ data” or “block apps from communicating with blacklisted network domains”.

## 3.7 EVALUATION

In this section, we focus our evaluation on three goals: (1) AppFork is able to support unmodified Android apps; (2) AppFork's storage overhead is small compared to state-of-the-art solutions; and (3) the time required for switching an app from one profile to another is small enough to not impact app usability. We tested AppFork on a Samsung Nexus S phone running our custom build of Android 4.1.2. AppFork was configured with two profiles, "work" and "personal", with associated policies (see Figure 3.5).

### 3.7.1 *Support of Unmodified Apps*

Referring to the results of the app analysis Table 3.2, by counting the number of apps that make use of the channels explicitly handled by AppFork, we estimate AppFork being able to achieve automatic profile isolation for 65% (9089) of the 14,067 apps we tested using ChannelCheck. For the rest of the apps, ChannelCheck is provided to help profile owners (e.g., corporations) evaluate the possible leakage channels not explicitly handled by AppFork, particularly public shared directories and custom content providers.

We also tested AppFork in depth with a smaller set of 24 apps.[8] We selected 21 free apps based on popularity and functionality.[9] We then added 3 more apps (Box, K9 Mail, OI Notepad) because their functionality makes them popular in the BYOD context (e.g., email was the most popular multi-profile app identified by our survey respondents). Thus, we report our evaluation results on these 24 apps.

#### 3.7.1.1 Methodology

We installed each app and added it to the two profiles. First, we verified that each app could be switched successfully from one profile to the other via AppFork. Second, we interacted with

[8] During development AppFork was successfully tested with a total of 35 apps including AngryBirds, Amazon and Marvel Comics.

[9] We took the top 13 free apps in the U.S. view of the Google Play Market and the top app in each of the top 15 app categories [58] (as of February 12, 2014). We excluded from our evaluation 3 apps in the Personalization, Tools, Arcade & Action categories, respectively, because they provide services that are not sensibly associated with a unique user profile. This gave us a total of 21 apps (not 25, because there were overlaps between the two selection criteria).

each app for 3 minutes in each profile, in a manner consistent with the app's functionality.[10] For example, interacting with K9 Mail involved syncing the inbox, composing an email, sending it, checking the sent folder, and returning to the inbox. For Facebook, it involved posting a status update, visiting a friend's page, and returning to the status update page. During these tests we also explicitly initiated actions that triggered task delegation occurrences. For example, in testing YouVersion Bible and Zillow, we performed activities that resulted in starting the browser to visit embedded URLs. For apps requiring an account, such as Facebook, K9 Mail, Netflix, and Skype, we created and populated dummy accounts.

We then verified that a partition was created for the app's data under each profile and that no data was shared across profiles via the channels described in Table 3.2. For file system and external storage, we inspected the files created under each profile. For content providers, we inspected the corresponding databases. For broadcast and direct intents, we inspected the execution logs and verified such operations were confined to a profile.

#### 3.7.1.2 Results

Table 3.4 lists the tested apps. All 24 apps worked seamlessly on AppFork without any modification. For 21 apps out of 24, AppFork was able to "automatically" block any possible cross-profile data channel, so we considered them "fully isolated" (hence 'F' in the table). For 3 apps, also identified by ChannelCheck, although correctly running, AppFork detected they used folders on the SD card outside of their app-specific directories. Kik Messenger created a public folder but did not store any data in it while NBC stored temporary files in a folder outside its designated app directory. Instagram stored pictures and videos taken while using the app, in the shared "Pictures" and "Videos" directories. As discussed in Section 3.5, today AppFork has no means to automatically prevent such occurrences, and it is not intended to do so because, for non-sensitive data shared directories are a legitimate channel to share data. For this reason, AppFork defers the treatment of these cases to a manual testing of the app by the profile owner (hence 'M' in the table).

[10]We experimented also with longer interaction times and found 3 minutes of targeted and continuous activity to be sufficient for triggering an app's cross-profile channels.

Table 3.4. Apps tested on AppFork. ‘F’ means the app is automatically fully isolated and ‘M’ means the app is isolated but requires manual evaluation for a specific channel.

| **App** | **Category** | **Support** | | **Possible Leak** |
|---|---|---|---|---|
| | | **F** | **M** | |
| Candy Crush Saga | Games | ✓ | | |
| Clumsy Bird | Games | ✓ | | |
| CNN | News & Magaz. | ✓ | | |
| Duolingo | Education | ✓ | | |
| Facebook | Social | ✓ | | |
| FacebookMessenger | Social & Local | ✓ | | |
| Google Earth | Travel | ✓ | | |
| Guess the 90’s | Brain & Puzzle | ✓ | | |
| Indeed Jobs | Business | ✓ | | |
| Instagram | Social | | ✓ | SDcard |
| Ironpants | Arcade & Action | ✓ | | |
| Kik Messenger | Communication | | ✓ | SDcard |
| Myfitnesspal | Health & Fitness | ✓ | | |
| NBCSportsLiveExtra | Sports | | ✓ | SDcard |
| Netflix | Entertainment | ✓ | | |
| Pandora | Music & Audio | ✓ | | |
| Skype | Communication | ✓ | | |
| SnapChat | Social | ✓ | | |
| Unroll Me | Brain & Puzzle | ✓ | | |
| YouVersion Bible | Books & Refer. | ✓ | | |
| Zillow | Lifestyle | ✓ | | |
| Box | Business | ✓ | | |
| K9 Mail | Communication | ✓ | | |
| OI Notepad | Productivity | ✓ | | |

### 3.7.2 *Storage Overhead*

AppFork may increase the apps’ storage overhead because its symbolic link-based implementation may cause duplicated files.

#### 3.7.2.1 Methodology

We measure the storage space occupied by AppFork with two profiles and compare it to that of a baseline system, such as a virtualization-based system like Cells [14]. We ran each app on AppFork and interacted with it for 3 minutes in each profile. We then ran the same app for the same duration and repeated the same actions on a clean build of Android. For both builds, we measured the amount of data stored in each profile under the app directories in internal and external storage. This includes the app's state and files as well as APKs and dex files stored in the /data/app and /data/dalvik-cache directories, respectively. To estimate the baseline overhead, we double the total size of APK and dex files on the clean build, since the app would need to be duplicated across VMs.

#### 3.7.2.2 Results

Figure 3.6 shows the storage overhead. In addition to the results for AppFork and the device simulating the baseline, we report AppForkOptimal, which represents the ideal case of a (hypothetical) AppFork implementation that tracks and eliminates redundant files across profiles. We identify files duplicated across profiles by comparing the hashes of similarly named files.

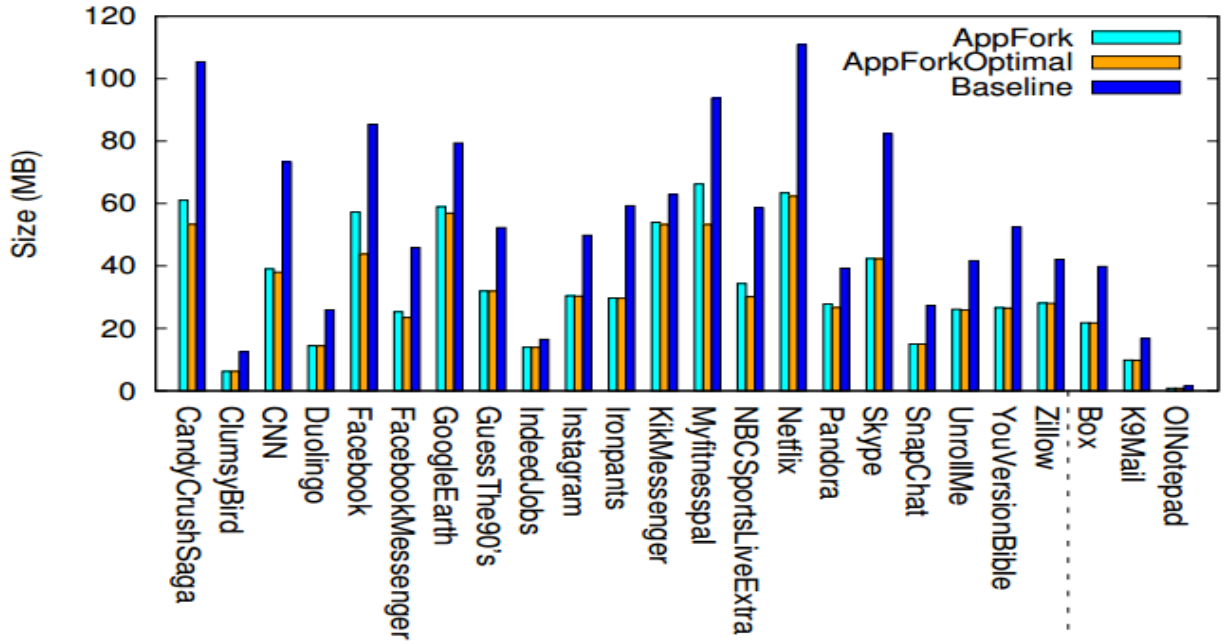


Figure 3.6. AppFork's storage overhead. Comparison between Baseline, AppFork and an ideal version of AppFork (called AppForkOptimal) that eliminates replicated files

AppFork reduces the storage overhead compared to the baseline by an average of about 36%. For most apps (e.g., Skype, Ironpants, CNN), the baseline requires almost double the storage because these apps maintain little state compared to the app's installation files. On the other hand, for apps like Facebook or IndeedJobs, the app's state is larger and the gap between baseline and AppFork is smaller. AppFork also performed well compared to AppForkOptimal. On average, the extra overhead introduced by our unoptimized implementation is 7%, not significant enough to justify the complexity of the AppForkOptimal implementation (see Section 3.5.2).

### 3.7.3 *Profile Switching Overhead*

Due to its design, AppFork's processing overhead for isolating profiles is negligible. In fact, switching symbolic links and filtering cross-profile operations (broadcast or direct intents) introduces negligible delays; also policy monitors listen to events already being broadcast. On the other hand, while usability is not a goal of our implementation or evaluation, the time necessary for switching apps between profiles may be high for some use cases.

#### 3.7.3.1 Methodology

We measure the time AppFork requires to stop an app, switch its profile, and reload it. For each app, we perform the switch 5 times and take an average.

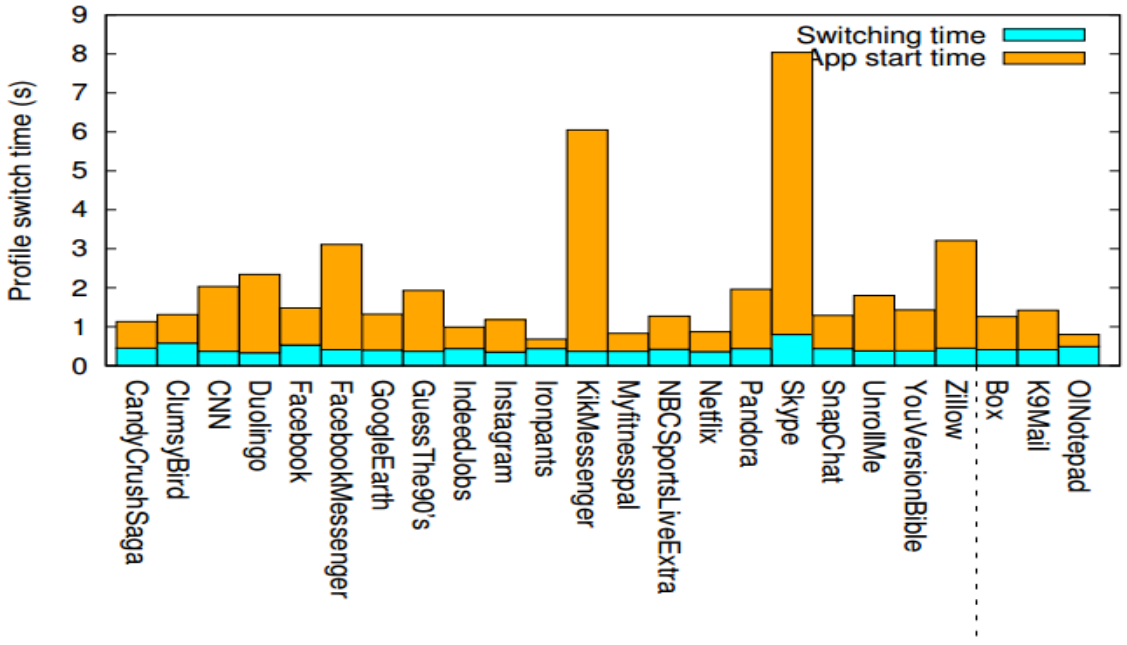


Figure 3.7. Time to switch an app between profiles

Figure 3.7 reports the switching time (including the time for terminating the app) and the app start time. The switching time is small, on average below 0.44 seconds. The time for starting an app (after being stopped) is unaffected by AppFork and, of course, varies by app; most apps took about 1.5 seconds, Skype over 7 seconds. Overall, the switching time is acceptable: it only slightly increases the time users already have to wait for apps to start.

We conclude with some observations about the AppFork user experience. In fact, one of us, who was not a developer of AppFork, used AppFork with his real work and personal accounts. He found AppFork to be functional and easy to use. His feedback solely had to do with the user interface which we stated as a non-goal. During his testing, the author accidentally entered his PIN incorrectly too many times in a row, thereby wiping his work profile. The author appreciated not having his personal profile data wiped – a direct but unintentional example of the benefits of the AppFork design, and the ability to separately enforce work and personal policies.

## 3.8 LIMITATIONS

Our study finds that it is feasible to offer the convenience of one phone, but the security properties of two phones, in a way that is compatible with many unmodified Android applications. There are, however, some limitations of AppFork, including corner cases that AppFork does not handle. These unhandled corner cases are often due to limitations in the underlying Android platform and APIs. We also stress here that our work focused on the threat model discussed in Section 3.3.3 and that future work consider stronger threat models.

**Multi-profile concurrent background apps.** In AppFork, an app in one profile remains in that profile until the user explicitly switches it. Background processes associated with that app also run only in the current profile. Automatically switching the profiles of apps running in the background raises multiple questions, including: how often should the switch happen? How can AppFork avoid switching the app at times not expected by the user? To maximize profile awareness and user control, we chose not to support such switching in our current implementation.

**App versions.** AppFork cannot run different versions of an app under different profiles, which may pose a problem if different profile owners have approved different versions of the same app.

**Exposing app weaknesses.** AppFork may expose existing flaws in Android apps that do not fail gracefully when denied permissions to start components in other packages. We argue that apps that depend on others to perform certain tasks should also anticipate the likelihood that those components may not be installed.

**Support multiple phone instances.** AppFork does not support profile isolation for telephony-related apps (e.g, Phone, SMS) which would require the phone to have multiple phone numbers. To provide full cross-profile isolation, we thus suggest that mobile platforms intended for BYOD scenarios support multiple phone instances, either physically (with two SIM cards, which is a common feature on phones in some countries) or virtually (with multiple International Mobile Subscriber Identities (IMSIs) on a single SIM card, as in [15], or with VoIP support, as in Cells [14]).

**Integrating content from multiple profiles.** Users may wish to view content from multiple profiles simultaneously. For example, in a calendar app, users may want to see both their work and personal events at once. To display isolated content from two profiles within the same user interface, we recommend that Android adopt techniques from prior work on user interface isolation [46].

**Linux IPC restrictions.** One of the possible cross-profile leakage channel not prevented by AppFork is standard Linux IPC, such as local sockets or pipes. Since these communication methods are not commonly used by Android applications (they were found in 0.5% of the apps in our measurements study) – but are hard to prevent without hooking each method individually – and since applications can achieve the same goals using Android-specific communication channels (such as intents), we recommend that mobile platforms like Android restrict apps from using Linux IPC. In the meantime, ChannelCheck can be used to detect use of Linux IPC.

## 3.9 SUMMARY

We studied the problems that BYOD raises on current mobile platforms, and proposed AppFork, a novel approach based on the notion of per-app profiles. AppFork provides the security and privacy properties of two phones, but with a single phone. Meeting both the security and functionality goals of AppFork is challenging because today's apps, despite being sandboxed, have many cross-application channels available for sharing data. These channels can turn into data leaks across profiles. We created ChannelCheck, a tool that uses static and dynamic analysis

to automatically detect such leakage channels, and we reported on the results of running it on over 14,000 Android apps. Based on these results, we studied how to prevent cross-profile data leakage via the most prominent channels, and implemented our resulting design in AppFork. Our evaluation showed that AppFork is effective at supporting unmodified existing apps, with low overheads in terms of storage and profile switching time.

# Chapter 4. NOSSN: TWO-WAY AUTHENTICATION THROUGH PHONE CALLS

## 4.1 INTRODUCTION

In the previous Chapter, we studied how to provide a trustworthy environment within the phone used by multiple entities through component isolation at the operating system level. We have also established that the amount of computational power on mobile devices, enable them to be used for a myriad of applications ranging from mobile wallets to health monitoring. However, the core function of a mobile device is still for the purpose of communication.

As with other modes of communication, one of the most pressing issues with communicating via phones is the need for a secure and robust approach to verifying the identity of the party you choose to communicate with. Hence, we now switch focus to enhancing trust for users who use their devices primarily for the purpose of communications. Specifically, we study how to use additional hardware, in a phone agnostic manner, to establish trust as a prelude to sharing sensitive information during a phone call.

Methods used today for verifying the identities of communicating parties over a phone call, fall short of providing a full guarantee of their identities among other shortcomings. For example, the de-facto way of verifying the identity of a customer placing a complaint to a customer service representative is to provide an appropriate response that only both parties know, to a knowledge-based question. During such authentication session, the customer service representative may prompt the customer to provide for example, the last four digits of her social security number or her mother's maiden name. This method of verifying one's identity is however, very easy to forge and leaves users prone to identity theft [47]. For instance, an attacker who has previously heard the answer to such a question is able to impersonate the legitimate caller at a later time.

Additionally, such authentication methods fail to preserve the privacy of the user. Acquisti et al [47] discovered that the use of the last four digits of one's social security number (SSN) as a challenge for validating the identity of a user provides an attacker with a 44% chance of deciphering the other first five digits. Previous systems have explored privacy preserving user authentication techniques such as in [48] where Bhargav-Spantzel et al studied the use of

biometrics in a two-phase authentication scheme to verify the identity of a user to a federated identity. Other like [29] and [31] have studied the use of the mobile phone as a second factor authenticator. Czeskis et al in [29] discussed PhoneAuth which employs a public key approach and uses a phone as second factor for web authentication while Li et al in [31] proposed Dolphin, a system that utilizes near field assertions through acoustic communications and sound power manipulation to resist relay attacks.

To address shortcomings in the way communicating parties authenticate each other over a phone call, we propose NoSSN, a solution that provides a privacy-preserving, two-way authentication scheme. NoSSN is a phone agnostic and portable solution that provides impersonation and replay resistance as security features. NoSSN is designed to be an additional piece of hardware (dongle) that can be attached to any mobile device through the audio jack, to provide secure and isolated profiles among users. We contribute two security protocols based on symmetric and public key (elliptic curve) cryptography implementations in addition to designing and implementing an audio modem. We evaluate and compare the security properties as well as the performance tradeoffs of the two protocols we studied. Additionally, we tested the NoSSN protocols over a live phone network and were able to successfully verify the identities of different individuals as well as identify fraudsters.

To the best of our knowledge, NoSSN is the first system to address the security and functional goals discussed in this Chapter as other systems have mostly studied user authentication to either the mobile device or in the context of using the mobile device as a second factor authenticator. Specifically, NoSSN seeks to provide two-way authentication to users communicating through the voice channel of a cellular network, regardless of make or model.

In summary, this Chapter contributes

- A class of security and functional goals that prior solutions are incapable of addressing.
- Two low bandwidth cryptographic approaches in establishing the identities of communicating parties.
- A solution that decouples and isolates the identity of users from the mobile device and supports legacy phones.

- Demonstrate the feasibility of our approach with a detailed architecture and prototype implementation of NoSSN over an actual phone network.

## 4.2 MOTIVATION

In the US today, authenticating a user to customer service representatives more often than not requires the last four digits of one's SSN or some other personal identifying information. Providing such details can at the very least be inconvenient when in public settings. As an example, a customer of a bank is seated at the gate, preparing to board an international flight. She realizes that she needs to inform her bank of her trip so she can still have access to her account while she is away without raising any red flags. Unfortunately, her bank doesn't provide the capability to notify them of this trip through an app. She must call and talk to a customer service representative who would require her to say the last four digits of her SSN or her mother's maiden name. This process of authentication raises obvious security and privacy concerns that are not currently addressed with today's technology. In the following sections, we discuss some privacy and security concerns that may arise with releasing sensitive information on a call with customer service representatives and address the current limitations of state-of-the-art technology that may be used to resolve our concerns.

Several privacy and security concerns arise from the seemingly simple request for the last four digits of one's social security number. Firstly, it has been shown that the first five digits of one's social security number can be correctly predicted 44% of the time [47]. This means that a rogue customer service representative or a malicious eavesdropper will have a 44% chance of stealing your identity if you choose to share the last four digits of your SSN.

Another potential danger of releasing sensitive information for authentication over the phone is a passive attacker. In such situations, an attacker gains access to the phone conversation, records the sensitive information provided, and becomes capable of impersonating the customer at a later time.

Lastly, an attacker may leverage social engineering attacks against a user by impersonating a customer service representative to extract sensitive information – such as SSN. There is currently no system to authenticate and validate the identity of such an imposter except to hang up and call the service's authorized and validated customer service number. A variant of such attack targeted at international students at top US institutions was reported in [2] and [3] where

fraudsters scared unsuspecting victims into sending huge sums of money on the premise that they were in danger of being deported. In each of the reported cases, the attackers already had the private immigration information of the students and appeared to be calling from legitimate U.S. Citizenship and Immigration Services (USCIS) toll free number.

NoSSN seeks to address these concerns by utilizing additional hardware connected through the phone's audio jack, to authenticate users to customer service representatives and vice versa without revealing sensitive information and eliminating the possibility to re-use the authentication credentials at a different time. NoSSN uses two cryptographic approaches – symmetric and public key – specifically optimized for low bandwidth channels to establish and verify the identities of communicating parties. Such use of additional hardware ensures that trust is removed from the phone and the delegated to the possession of the hardware by a legitimate user who has appropriate credentials to use it.

While we acknowledge that the SSN is unique to the United States, we believe that the concerns raised here are relevant to other unique identifiers – like a pin, passport or national ID number – that may be used to authenticate a user in other nation states over a phone. Consequently, we believe that our described system may be able to achieve similar authentication goals if deployed in those environments.

In the following section, we outline our desired functional and security goals for NoSSN. We then proceed to examine alternate approaches to the use of additional hardware in achieving similar goals and their limitations.

## 4.3 GOALS

Our goal is to eliminate the need to share sensitive information, such as social security numbers, when individuals talk to customer service representatives by phone. We propose using trusted additional hardware – a dongle – that is capable of sending arbitrary data over the voice channel of cellular networks to transmit authentication tokens during a call session. Utilizing the voice channel in this capacity offers the additional benefit of performing authentication as part of the phone call.

As a prelude to discussing our goals, we begin by defining the following actors in our system:

- *Service Entity*: Organization contractually bound to provide a given service to customers, e.g., banks, phone and cable companies, schools, etc. The service entity is responsible for managing servers used during authentication sessions.
- *Customer service representative:* Individual representing a service entity that seeks to address concerns of customers.
- *User or customer:* Individual seeking to be authenticated during a call to a customer service representative.
- *Phone:* A mobile device, softphone or landline used by either the user or customer service representative. The mobile device could be a basic, feature or smart phone.
- *Dongle:* Trusted hardware token for authenticating user or customer. Easily connected to both ends of a bi-directional conversation through the phone.

Below, we outline key security goals that our dongle seeks to address:

1. *Impersonation resistance*: An unauthorized third party, without appropriate credentials and without physical possession of the dongle should not be able to impersonate the legitimate user.
2. *Service authentication*: In addition to authenticating the user to the server, the user must be able to tell apart legitimate servers from imposters.
3. *Privacy preserving user authentication:* Authentication from user to service should not reveal user's private credentials (like the SSN) to the customer service representative.

These goals imply other security goals, such as resistance to replay attacks. The following are desired functional goals for the dongle:

1. *Lightweight design*: Due to the relatively high packet loss of the communications channel between the dongle and the service, via the phone's audio connection (see Evaluation section), a lightweight, yet robust authentication scheme is needed to address dongle usability.
2. *Phone agnostic and portable design*: Dongle should be compatible with a wide variety of phones including very basic or feature phones. Additionally, the user can use the dongle with multiple phones.

3. *Support for multiple services*: A single dongle should support multiple service entities but be limited to a single user, i.e., the user and only the user, may choose to use a single dongle for authentication to banks, cellular network providers, schools and so on, but the use of the same dongle by another user is prohibited.
4. *Ease of service enrollment and revocation*: Due to the previous goal, we desire that the enrollment process to enlist a new service entity should be easy and intuitive. Likewise, the process of service entity revocation should also be straightforward.
5. *Dongle revocation:* Support dongle/service revocation in a manner that is on par with or better than current schemes in use (e.g., for handling lost credit cards).
6. *Less taxing on consumer's memory*: Consumer should only be required to remember the password to the dongle. No additional information should be required for the authentication sessions.

## 4.4 THREAT MODEL

We now discuss the actors of the NoSSN system, potential adversaries, and the behavior of the system in the presence of such adversaries.

**User**. We assume that legitimate owners will not try to compromise the system in order to spoof their own identities. This means, for example, that a dongle and protocol do not need to be hardened against a user attempting to steal his or her own credentials. However, a user might try to compromise the authentication capabilities of other users. Thus, we must harden our dongle and protocol against a user seeking to gain enough information from his or her own dongle to impersonate another user. Users are untrusted by both the service entity and the customer service representative and are thus required to authenticate themselves through the dongle.

**Dongle**. Under our model, we trust that the dongle manufacturer has no malicious intention to steal locally stored user data. We further assume that the manufacturer has only included core functionality to store keys in a secure and tamper resistant manner locally on the dongle and to send legitimate authentication messages through the voice channel of cellular networks. We also assume that the dongle is password or pin protected with biometrics capabilities – like fingerprints – and that the user has not shared this information with any other individual –

leaving access to only the legitimate user. The dongle is trusted by the legitimate user, customer service representative and service entity. Lastly, we consider out-of-scope, side-channel attacks on the dongle, though note that there are existing mechanisms to protect against such attacks (e.g., tamper resistant hardware [49]).

**Service Entities**. We trust that the service entities like banks, schools, and so on are doing everything within their legal obligations to ensure the security and privacy of the user's personal information. While the user trusts that Federal/State regulations are in place to hold corporations responsible for any data breach and hence force them to be responsible with customer data, users do not trust that customer service representatives can and will always live up to the ethos demanded them by their employers. Likewise, users do not trust that their data is 100% secure from potential external attackers. For instance, there is the potential of external attacks on corporations where attackers can at the very least expose or sell stored data, depending on their motives. Nevertheless, users partially trust service entities as they are incentivized to secure customer data due to regulations. The design of the NoSSN protocols should ensure that the theft of one's data via an external attack on a particular service entity should not by itself compromise the user's credentials on other service entities.

**Customer Service Representative**. While most customer service representatives are trustworthy, we acknowledge that not all customer service representatives are. Hence, we do not trust the customer service representative with the user's social security number (SSN) under our model and assume that s/he now has a 44% chance [47] of correctly guessing the user's SSN if the user supplies the last four digits. Once a victim's SSN has been acquired in addition to other personal details – like his/her full names, date/place of birth, and so on, the rogue customer service representative can then sell this information on black markets or take out loans in the victim's name amongst other actions. We consider such an attack to be insider driven. Victims may then suffer financial loss or worse yet arrest warrants issued due to mistaken identities.

Users do not trust the customer service representative while the service entities partially do. The service entities can only claim to have made the best effort in hiring responsible and honest individuals for customer service representative roles. Hence, the architecture and dongle must be hardened against untrustworthy representatives.

**Phone**. Under our model, the user, customer service representative and service entity do not trust the phone. We assume that the phone can be compromised by an attacker who may actively or passively eavesdrop on the conversation. Additionally, the phone may have malicious applications or compromised OSes running on it that can exfiltrate keypad presses, audio conversations and the likes. Moreover, the phone may be shared amongst multiple people – e.g., family – with the phone number associated with a service entity. The trusted dongle, in all of the cases mentioned, therefore acts as an extra layer of abstraction where specific users can be accurately identified and verified.

**Network**. The actors in our system know that the network may observe, modify, and prevent the delivery of messages. Additionally, our model distinguishes between two potential network attackers – passive and active attackers. Our system should be able to defend against a passive network attacker that tries to replay authentication sessions or extract valuable information from prior authentication sessions that can be used to impersonate a user. To meet other low-bandwidth goals such as a lightweight design, our model does not consider defenses against an active network attacker as it introduces significant byte overhead. For example, we do not prevent an active attacker from taking control of the audio channel after authentication even though they cannot learn things like the user's SSN, or learn enough information to become the user in the future. Hence, our system and its actors do not trust a passive network attacker as we defend against such. Due to our lightweight design tradeoff, our system and its actors do trust an active network attacker and defer further discussions to Section 4.10.2.

**External Attackers**. It is also possible to have external attackers who have gained some knowledge about the customer (as in [2] and [3]) – through an active breach of corporate data, for instance – to impersonate customer service representatives. These attackers are capable of mounting very sophisticated social engineering attacks against unsuspecting victims as they may have some fore knowledge about the customer. We consider such attacks in scope and discuss a novel technique involving a two-way authentication over the voice channel to mitigate them.

Table 4.1. Summary of NoSSN actors and threat model: When read from left to right, the table displays the stakeholder's (along each row) level of trust in the other actors (along each column) in the system

| **Stakeholder** | **Status of Trust** | | | | | | |
|---|---|---|---|---|---|---|---|
| | User | Service Entity | Customer Service Rep | Phone | Passive Network | Active Network | Dongle |
| User | N/A | P | N | N | N | T | T |
| Service Entity | N | N/A | P | N | N | T | T |
| Customer Service Rep | N | T | N/A | N | N | T | T |

| NoSSN Actors and Threat Model Table Legend | |
|---|---|
| T | Fully Trusted |
| P | Partially Trusted |
| N | Untrusted |
| N/A | Not Applicable |

Table 4.1 summarizes which actors are trusted and untrusted under our threat model. We indicate varying levels of trust denoted by the "T" for complete trust and "P" for partial trust.

We identify robustness against side-channel attacks to be non-goals of this work.

## 4.5 Alternate Approach and Limitations

**Maintain Status Quo:** Current state of affairs in customer validation involves only knowledge-based factors such as querying the customer for sensitive information that they only should know. Such questions may revolve around the last four digits of the SSN, associations with

certain addresses, mother's maiden name, and other information that only the customer is supposed to know.

This approach fails to meet our security goals (1)-(3) – *impersonation resistance, service authentication and privacy preserving user authentication* – in Section 4.3. A passive attacker, for instance, can listen in on the answers to such questions and successfully impersonate legitimate customers at a different time. Secondly, customers have no way of validating the identity of customer service representatives on the other end of the call which leaves them severely exposed to the risk of vishing attacks – voice phishing – as evident in [2] and [3].

**Authentication via SMS**: An obvious candidate to eliminate the need for a customer's SSN is with the use of SMS. A typical authentication session would involve generating and sending a random number to a pre-registered phone number of the customer and the display of the customer service representative. The customer service representative is then able to confirm the identity of the individual, if the number read out by the customer matches what he sees at his end.

This approach eliminates the need to provide the last four of one's SSN (although some unique identifier like the customer's username may be needed to retrieve the phone number). However, it does not meet our security goal (1) – *impersonation resistance* – as it still does not provide as high enough confidence that the customer service representative is speaking to the actual customer when compared to SSN approach. For instance, if the same phone number is allowed to be used across multiple accounts, then the customer service representative cannot particularly attribute an authentication session to any one individual due to the ambiguous phone number. Similarly, if the constraint of a one-to-one mapping with phone numbers and accounts is in place, the user will not be able to authenticate herself from another phone if she for some reason, is not in possession of her phone. As such, authentication via SMS fails to meet functionality goal (2) – *phone agnostic and portable design* – where user accounts on apps may not be portable across phones. Secondly, this approach fails to meet security goal (2) – service authentication – and leaves users vulnerable to phishing attacks. An example of such phishing attacks on authentication codes sent via SMS is discussed in detail in [50].

Moreover, authenticating via SMS seems unattractive given the potential for SMSes to be manipulated through the untrusted network (see Section 4.4) or installed malware. Additionally,

user authentication via SMS may also be less usable than actually having the customer say the last four digits of their SSN to the customer service representative. This is due to the fact that, with repetitive use of the last of four digits of their SSN, the customer most likely knows this information by heart when compared to having to check and recite a randomly generated number each time during authentication.

Lastly, SMS may experience delivery delays and may be unreliable in different parts of the world.

**Cellular Phone App**: Another appealing approach is to use a phone app or data channel of cellular networks for authentication. However, this approach requires that the phone and network should become trusted entities and as such, does not provide the additional layer of security that a dongle like NoSSN provides. For instance, if an adversary compromises the phone, then security may be lost on the device.

Like authentication through SMS, this approach suffers similar constraints by failing to meet functionality goal (2) – *phone agnostic and portable design* – where authentication is portable across phones and a user is not required to have her registered device with her at the time of the call, i.e., she can use another person's phone and be authenticated without any hassles. Lastly, the range of phones that can be supported is limited due to the fact that a data connection and at least a feature phone is needed.

**Customer PINs:** Some services like Union Bank use models where the customer is assigned a phone pin and asked to provide that during automated telephone banking sessions [51].

While this approach addresses security goal (4) – *privacy preserving user authentication*, it fails to meet our security goals (1)-(2) – *impersonation resistance, service authentication*. A passive attacker can still eavesdrop on the conversation and record either the DTMF tones for the user's pin or the user's human-understandable relay of their pin number. Additionally, this approach fails to meet our functionality goals (3) and (6) – *support for multiple services and less taxing on consumer's memory* – requiring each user to remember multiple pins, for multiple services, which can easily become cumbersome.

**Second Factor Authentication:** Finally, we are careful to draw the distinction between second factor authentication (2FA) and NoSSN. With NoSSN, the user is able to initiate the authentication process through the push of a button on the password, pin, or biometrics protected dongle similar to the user experience of the Universal Authentication Framework (UAF) by FIDO Alliance [32]. However, NoSSN offers a different user experience from 2FA – as seen in the Universal Second Factor (U2F) FIDO standard [32] and RSA SecureID Hardware Tokens [52], for example – in the sense that no additional information like the username and password is requested from the user. 2FA may also utilize various user contexts such as the GPS coordinates of the caller or an NFC enabled card that the user can bring in close proximity with the phone to confirm her identity. Krol et al [53] have shown 2FA to be less usable even though they provide similar security goals of NoSSN.

Having examined our goals and threat model, we switch our focus to describing the design and architecture of NoSSN in the following section.

## 4.6 System Design and Architecture

In this work we sought to investigate the feasibility of building a system capable of achieving the goals outlined above, and under our threat model. We found that it was possible to do so, and give an overview of the system design and architecture of our resulting system, NoSSN, here. A detailed discussion of the validation process is given in Section 4.6.2.3.

Figure 4.1 shows our NoSSN architecture. At the core of NoSSN's functionality is our cryptographic protocol, built on the audio modem and transport layer. The physical and transport layers provide the reliable transportation of arbitrary data through the voice channel of cellular networks.

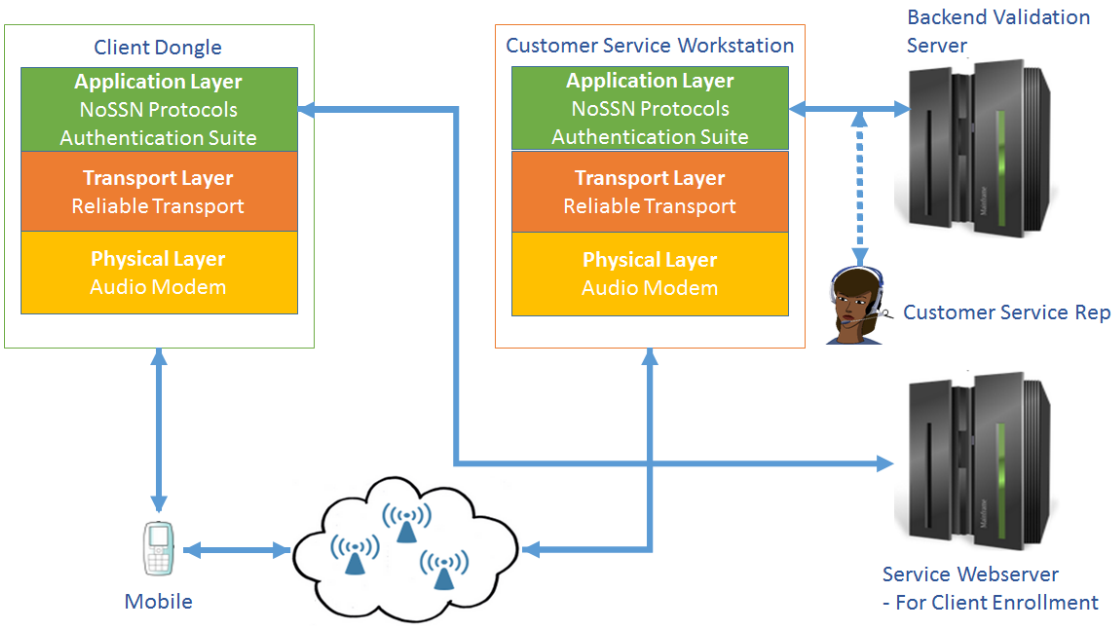


Figure 4.1. NoSSN Architecture

The application layer comprises of NoSSN protocols and an authentication suite to establish and verify the identity of a consumer. Specifically, we explore the design of a symmetric key and a public key approach – both optimized for a low bandwidth application – in providing two-way authentication between the customer service representative and the user. Designing for low bandwidth applications, allows the modem to still be comparatively effective even in bad network conditions. Additional server components – for enrollment and validation – are present from the perspective of the customer service representative or service entity. In our model, we envisage that an instance of the backend validation server provides services to multiple instances of customer service workstations. This would mean that each customer service workstation would need to authenticate itself to the validation server whenever a request to validate the identity of a user is sent. The client and workstation dongles are connected via the audio jack of the phones. We note here again that the dongles are capable of sending arbitrary data bits through the Hermes [54] based audio modem. Additionally, the dongles are protected using passwords, pins or biometrics to prevent unauthorized access to them.

If users choose to authenticate with NoSSN, they get better security and privacy when compared to the status quo. They, however are able to revert to the old fashion way of authentication as backup if the dongle fails to work. We note here that the option of a backup

method creates the potential for customers to be vished. For example, an attacker can mount rollback attacks, as seen with cryptographic protocols, e.g., [55], by claiming that the NoSSN backend is down, cajoling unsuspecting victims to use insecure methods of authentication. We defer discussions of the attack to the Section 4.10.

Before discussing specific details of the NoSSN protocols – symmetric and public key approaches, we give a background on the design and implementation of the audio modem and transport layer of the NoSSN architecture depicted in Figure 4.1.

### 4.6.1 *Background: Audio Modem*

**Physical Layer.** Sending non-voice data via the voice channel of cellular networks is non-trivial. This is due to the fact that cellular codecs are optimized for signals within the auditory range of humans (~300 Hz – 3.4 KHz) and employ techniques like Voice Activity Detection (VAD) and Automatic Gain Control (AGC) that distort non-auditory signals. Previous industry and research efforts have, however, shown that it is possible to send arbitrary data over cellular network channels.

To the avoid the use of proprietary protocols, we built an audio modem – 759 lines of code – based on the Hermes protocol proposed by Dhananjay et al [54], for the physical layer of NoSSN. The Hermes protocol utilizes techniques like periodic amplitude variation to imitate voice signals, in addition to maintaining a fixed fundamental frequency in the modulated signals.

We employ the ideas described in Hermes in our implementation, but note here that since our goal is to study the feasibility of sending arbitrary data over the voice channel of cellular networks in NoSSN, we do not attempt to optimize the underlying modulation scheme. Dhananjay et al [54] claim to have achieved a 1.2 kbps goodput with their Hermes implementation.

**Transport Layer.** To ensure the delivery of data over audio, we implemented a reliable transport layer that supports variable-length packets of up to 70 bits including a 7 bit preamble. Similar to sockets, our transport layer exposes *sendAll* and *receiveData* APIs for sending and receiving any given number of bytes, respectively.

### 4.6.2 *NoSSN Protocols*

#### 4.6.2.1 Dongle Enrollment

We envision that the dongle can be used for authentication on multiple services, i.e., the same dongle can be used both to authorize the user to their bank and to their phone company, to name a few. Consequently, a user would need to enroll the dongle on the website of the service of interest before it can be used for authentication. The dongle ensures that data owned by one service entity is securely stored and differentiated from other resident services using a unique service ID or service public key.

The service entity's web server is responsible for a one-time enrollment phase. The user signs into the service entity's web page with their user account and provisions her NoSSN dongle to be used for that particular service entity via a USB connection.

Table 4.2 shows the tokens generated and stored during the enrollment of a dongle in a service using the symmetric key NoSSN protocol. The service ID uniquely identifies the service to the dongle and is stored on both the server and the dongle. However, the 8 byte salt is only stored on the dongle while the server stores its 32 byte SHA256 hash. This is done to ensure that a would be passive attacker – with a read-only access to the service's backend – would still require the salt stored on the secure dongle to be able to impersonate the client to customer service representative.

Table 4.2. Essential tokens generated and stored during the enrollment of a dongle in a service using the symmetric key NoSSN protocol

| **Token** | **Length in Bytes** | **Storage Location** |
|---|---|---|
| Service ID | 8 | Client, Server |
| Salt | 8 | Client |
| Salt Hash | 32 | Server |
| Symmetric Key | 32 | Client, Server |
| Salt and Symmetric Key Expiration Date | 4 | Client, Server |

| | | |
|---|---|---|
| User's Personal Data or Unique Identifier | variable | Client, Server |

Table 4.3 shows the tokens generated and stored during the enrollment of a dongle in a service using the public key NoSSN protocol. With the public key approach, both parties exchange their respective public keys that will be used in the generation of a shared secret key (see Section 4.6.2.3) and their associated expiry dates.

Table 4.3. Essential tokens generated and stored during the enrollment of a dongle in a service using the public key NoSSN protocol

| **Token** | **Length in Bytes** | **Storage Location** |
|---|---|---|
| Server's Public Key | 32 | Client, Server |
| Server's Private Key | 32 | Server |
| Client's Public Key | 32 | Client, Server |
| Client's Private Key | 32 | Client |
| Salt | 8 | Client |
| Salt Hash | 32 | Server |
| Salt Expiration Date | 4 | Client, Server |
| Client's Public Key Expiration Date | 4 | Client, Server |
| Server's Public Key Expiration Date | 4 | Client, Server |
| User's Personal Data or Unique Identifier | variable | Client, Server |

Once any of the keys are expired, for both the symmetric and public key approaches, the user is notified and would have to login to her service entity's website with her credentials (as describe above) to re-provision the dongle. We acknowledge the potential of having vulnerabilities in the USB protocol stack or malware – capable of eavesdropping on the secret tokens – installed on host machines, and we defer such discussions to Section 4.10.

#### 4.6.2.2 Dongle Revocation

We model our approach of revoking a dongle after the process of deactivating credit or debit cards in the event that an owner's wallet is stolen or lost. Such a person is required to call the customer service representatives of each credit card in his wallet to report the loss or theft. In the event of a dongle loss or theft, the user signs into her service entity's web account in similar fashion to the dongle enrollment phase and deactivates/decommissions her dongle for that particular service entity. She would then have to sign into the web accounts of other service entities for which she had provisioned her dongle and deactivate them in similar fashion. Alternatively, she may choose to call the customer service representative authenticating herself using traditional methods, such as providing information about her mother's maiden name, etc., and notifying him of a loss or theft of her NoSSN dongle. It is important to note that, similar to credit or debit card loss, should the user note any fraudulent activity since their dongle loss, the process of filing a claim would be an activity separate from deactivating the dongle.

To avoid the user having to notify each service entity, a variant design would entail the use of public-key cryptography where a dongle and its corresponding registered services can be decommissioned by notifying the trusted certificate authority (CA) of the loss or theft of the device. We also note that this option would require the service to check a certificate revocation list.

#### 4.6.2.3 Verification Protocol

User verification takes place in-band with the phone call after the user has connected one end of the dongle to the audio jack of the phone and the other end to a headset. On the other end, the customer service representative connects his desk or softphone to the validation server via his workstation. Both parties then exchange information detailing their respectively supported versions and tokens necessary to establish and verify their identities.

In the following paragraphs, we describe two design specifications of verification protocols utilizing symmetric and public key crypto implementations respectively.

**Symmetric Key Crypto Approach.** The verification protocol is comprised of a handshake phase (shown in Figure 4.2 and steps 3-6 in Figure 4.5), as well as a client and server validation phase. In the handshake phase, the client sends a hello along with a random 8 byte challenge and

protocol version number for which the server responds with an 8 byte protocol check status code – indicating whether the protocol version is supported or not. Additionally, the server sends a random 8 byte challenge and service ID to the client. The generation and exchange of random challenges by both parties ensures that we achieve our implicit security goal of *replay resistance*. Without such measure, a passive attacker would be able to record and reuse the audio tones of an authentication session at a later time. The handshake phase is finalized when the client confirms via the protocol check status code that its version of the verification protocol is supported and that the received service ID corresponds to a previously provisioned service entity. Having version numbers ensures that both the client and server can negotiate the most secure protocol to employ during client/server authentication.

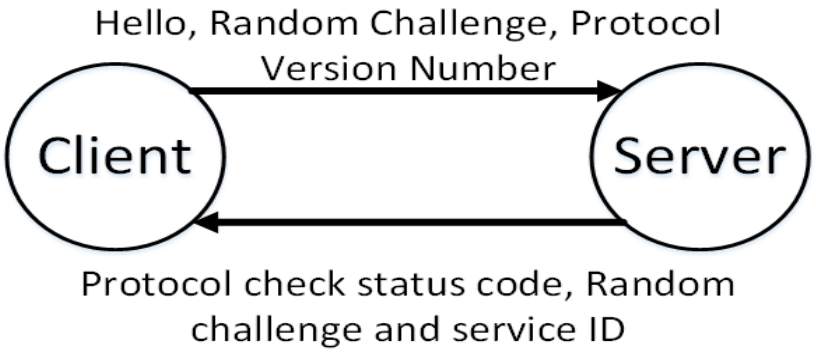


Figure 4.2. Handshake Phase

To initiate the client validation phase (shown in Figure 4.3 and steps 7-9 in Figure 4.5), the dongle retrieves the user's data for that particular service ID from its secure and trusted partition. It then generates a MAC key through a password based key derivation function (PBKDF) from the previously stored salt and symmetric key provided during the service enrollment phase. A digest is then computed from the user's personal data, client and server challenges, the protocol version number and protocol check status code and sent along with the salt and client's unique identifier to the server. To complete the client validation phase, the server retrieves the user's stored data (if it exists) and compares the computed hash of the received salt to what was stored during the enrollment phase. In the event that the user does not exist, an authentication failure code is sent to the client along with the hash of the received digest computed using a randomly generated key. The extra step of computing the hash is done to prevent the leakage of

information about the non-existence of a user on any service entity. Likewise, the MAC key is generated on the server side through a PBKDF using the symmetric key as the password and the received salt – once it has been verified. A digest is also computed on the same set of user data, client and server challenges, protocol version number and protocol check status code and compared to the digest received from the client. With a match, a successful authentication code is then sent to the client while a failure authentication code is sent in the event of a non-match. We included the agreed upon protocol version number and protocol check status code in computing the digest to prevent protocol rollback attacks as described in [55].

Using a PBKDF to compute the MAC key during each authentication session ensures that a compromise of the service entity's backend – where data is only revealed and not modified – does not reveal the MAC key of the user. The attacker would still need an additional piece of information – the salt – which is securely stored on the dongle to be able to impersonate a client. Hence, the use of a PBKDF to compute the MAC key during each authentication session enables NoSSN to achieve our security goal (1) – *impersonation resistance.*

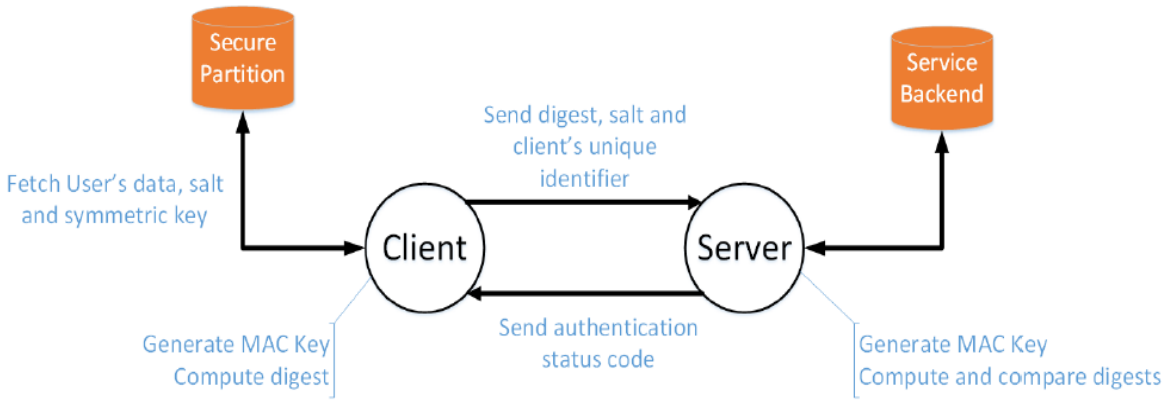


Figure 4.3. Client Validation Phase

Finally, as a way to establish and *validate the identity of the server* – security goal (2), the server computes the digest (m″) of the entire authentication transcript using the generated HMAC Key and sends it to the client. Likewise, the client also computes the digest of the entire authentication transcript at its end and compares it to the digest (m″) received from the server. The client is then appropriately notified of the result of the server authentication. This is shown in Figure 4.4 and steps 10-11 of Figure 4.5.

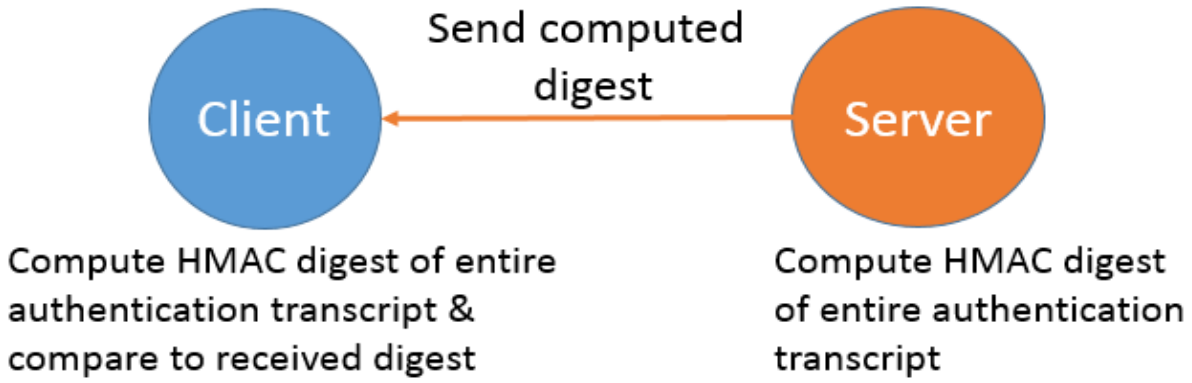


Figure 4.4. Server Validation Phase

In establishing and verifying the identities of the client and server, the design of NoSSN – as described above – does not reveal sensitive data to the customer service representative or the customer, thereby achieving our security goal (3) of *privacy preserving user authentication.*

The aforementioned steps are summarized below in Figure 4.5:

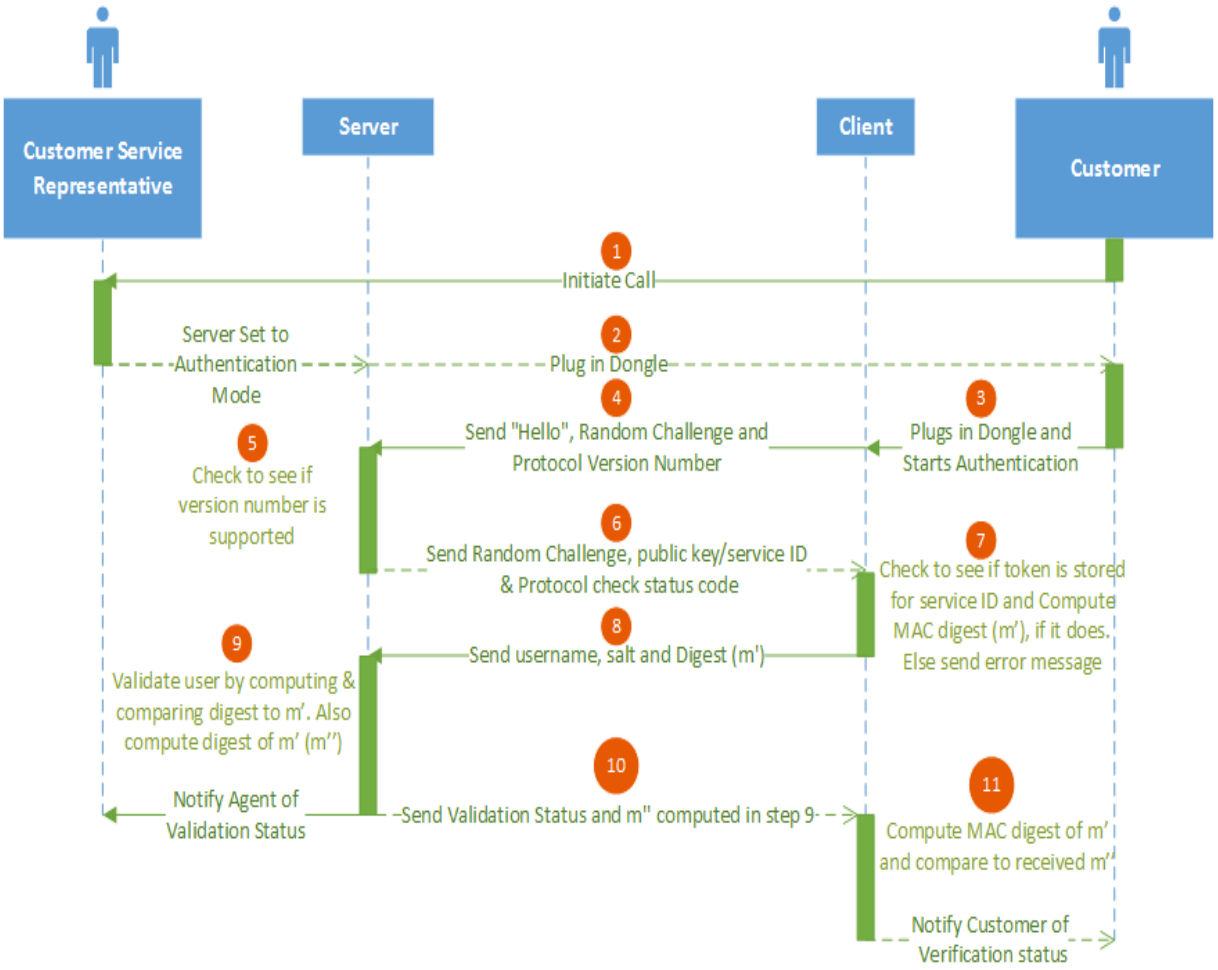


Figure 4.5. Summary of Symmetric and Public Key Crypto Approaches

**Public Key Crypto Approach.** As with the symmetric key crypto approach, NoSSN's verification protocol comprises of three phases: the handshake, client and server validation.

The handshake phase however slightly differs from the symmetric key approach where the server responds with its public key as opposed to the service ID. The client checks to see if there was a prior service enrollment by looking up the server's public key in its record. As with the symmetric key approach, the client and server in this design, both send an 8 byte random challenge to achieve our implicit security goal of *replay resistance* – during the validation phases. The handshake phase is shown below in Figure 4.6 and in steps 4-6 of Figure 4.5.

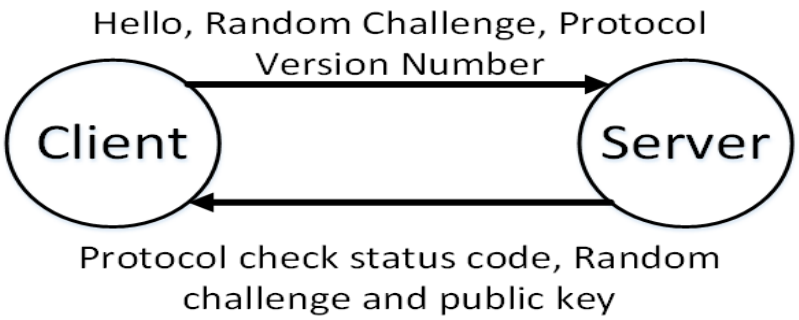


Figure 4.6. Handshake Phase

The client validation phase here, differs from the symmetric key approach in the way that the shared symmetric key is handled. As opposed to storing the shared symmetric key on both ends, the client and the server – during every authentication session – generate the shared symmetric key using their respective private keys and the public keys of their counterparts through an elliptic curve Diffie-Hellman key agreement scheme. The private key of the client is stored securely on the tamper-resistant dongle while that of the server can be stored in a similarly secure fashion. Every other detail ranging from the generation of the HMAC key to the computation and comparison of the digest of the user's data and random challenges follows in similar fashion as described in the symmetric key approach. The client validation phase is shown below in Figure 4.7 and in steps 7-9 of Figure 4.5.

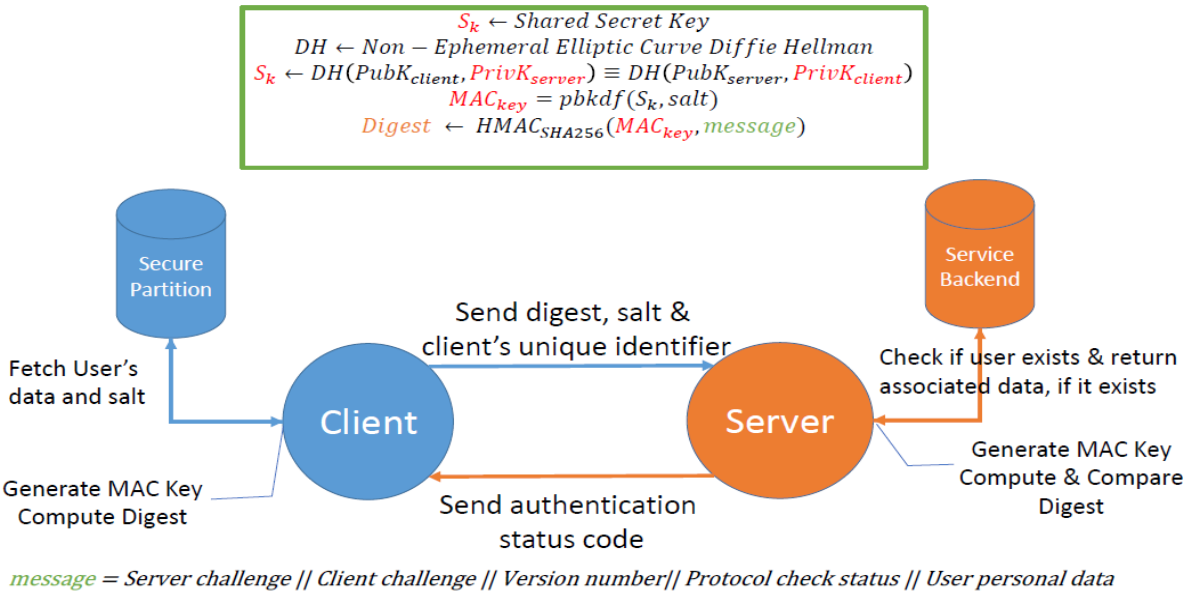


Figure 4.7. Client Validation Phase

Likewise, the *server validation* phase – security goal (2) – remains unchanged from the symmetric key approach as described in Section 4.6.2.3. Refer to Figure 4.4 and steps 10-11 of Figure 4.5 for a summary of the server validation phase.

## 4.7 IMPLEMENTATION

In the following paragraphs, we discuss our implementation of NoSSN with the bulk of our work centered on the verification protocol and the design of the audio modem. Further, we describe our implementation of both symmetric and public key crypto based verification protocols.

To support dongle enrollment and revocation, each service entity is required to deploy additional web pages to their already existing online portals used by customers to access their services. We consider this additional webpage to be trivial and cost effective thereby achieving our functional goal of (4) *ease of service enrollment and revocation*. Hence, we believe that this addition will not deter service entities from adopting the use of NoSSN for authentication.

In generating the 32 byte symmetric key, 8 byte salt, and 8 byte random challenges, we utilized pycrypto[11] library's *Crypto.Random.random.StrongRandom.getrandbits* function. We envisage that the symmetric key and salt are to be refreshed after a given period of time.

To compute the SHA256 hash of the salt that was stored on the server and the digest of the message comprising of the user's data and random challenges, we utilized Python's hashlib and hmac modules. We again note here that storing the hash of the salt and not the actual salt on the server ensures that there is still a required piece of information (securely stored on the dongle) needed from the client to assume her identity, if the server has been compromised.

For the public key approach, we used a Python implementation – curve25519-donna 1.3[12] – of the Curve25519 algorithm described in [56]. Curve25519 offers a fast elliptic-curve key agreement protocol where both communicating parties – each having a private and public key pair – exchange their public keys. With this set of information, both parties are then able to generate a shared secret key. In our case, the 32 byte shared secret key used to generate the HMAC key was computed through a call to *curve25519.Private.get_shared_key*.

[11] https://pypi.python.org/pypi/pycrypto
[12] https://pypi.python.org/pypi/curve25519-donna

Finally, in both the symmetric and public key approaches, we utilized hashlib's *pbkdf2_hmac* function to generate the HMAC key used in the computation of user specific digests.

We note here that both implementations go about authenticating the user and the customer service representative to each other without revealing sensitive information, thereby achieving our security goal of (3) privacy preserving user authentication.

## 4.8 EXPERIMENTAL SETUP

For the purposes of our future discussions, we hereby describe our experimental setup used in the design, implementation and evaluation of NoSSN.

Our setup consists of a pair of AT&T Samsung Galaxy Nexus phones connected to two Linux powered desktop computers used as instances of a client dongle and a customer service representative's workstation. Both phones were connected through a phone call over a live AT&T cellular connection. We designed, implemented and evaluated NoSSN software modules using Python 3.4.3.

## 4.9 PROTOTYPE EVALUATION

We investigated two measurement metrics – total bytes sent or received and the average time for various authentication phases – while evaluating our NoSSN prototype. In deducing the average authentication time for the various phases, we conducted experiments consisting of an average of 10 validation trials.

The total amount of bytes sent or received indicates the tradeoffs between the symmetric and public key approaches of NoSSN. In Figure 4.8, we note an overhead of 32 bytes sent using the public key approach when compared to the symmetric key approach. There is also a similar overhead of 22 bytes received in the public key approach compared to the symmetric key approach.

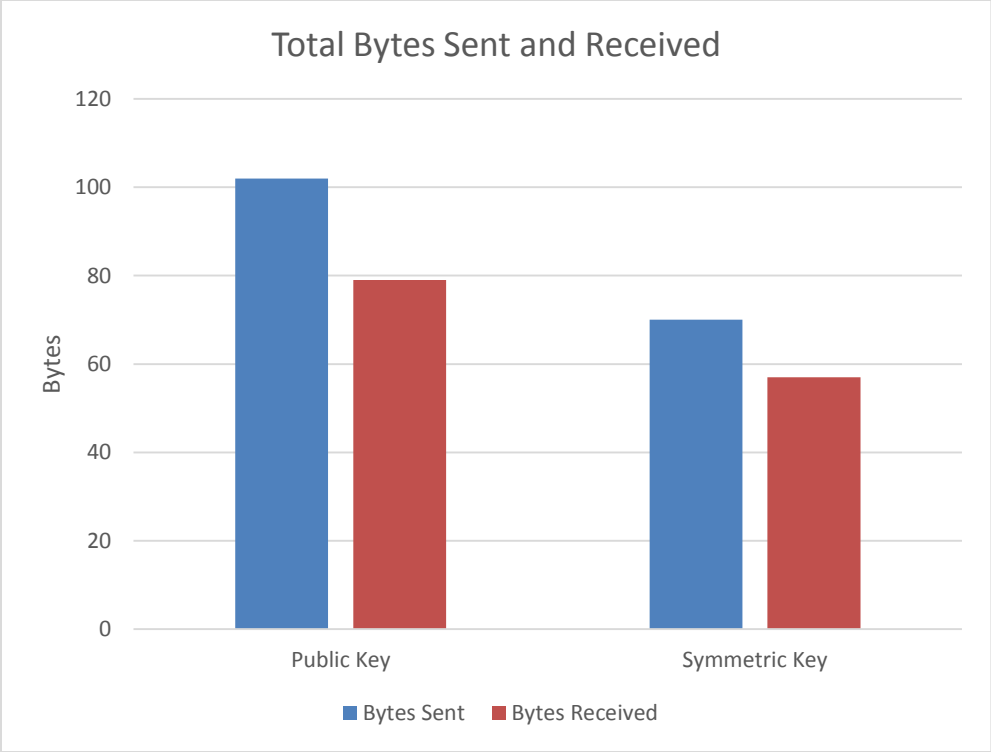


Figure 4.8. Number of Bytes Sent/Received in NoSSN's Public and Symmetric key Protocols

Figure 4.9 shows the average time for an authentication session for both protocols along with the average time for the three phases discussed in Section 4.6.2.3 – handshake phase, client validation phase and server validation phase. The average total authentication time for the public key approach again has an overhead of approximately 1.16 minutes when compared to the symmetric key protocol. Additionally, the average client validation amounts to an overhead of approximately 34.2 seconds for the public key approach while the difference of the average server validation for both approaches is negligent.

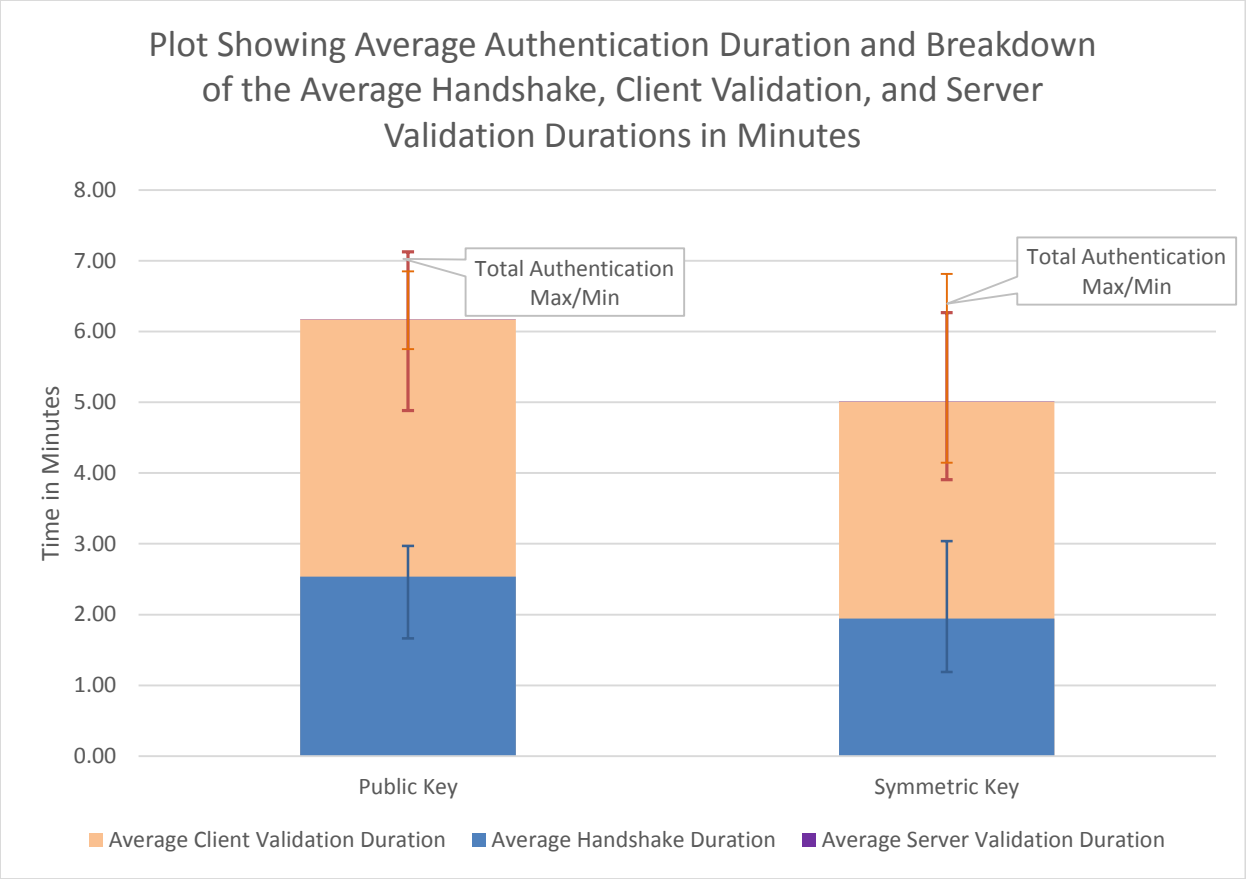


Figure 4.9. Average Total Authentication Time and Breakdown of Average Handshake, Client and Server Durations in Minutes for NoSSN's Public and Symmetric Key Protocols

**Microbenchmarks**. In profiling the physical and transport layers of NoSSN, we measured the average round trip time (RTT) for a packet, the packet loss rate and the time to reliably send a fixed number of packets. We used a packet size of 70 bits, with a payload of 27 bits for all measurements.

The average RTT measurement gave us an understanding of how efficient our physical and transport layers were. We measured an average RTT of 4.91 seconds with a standard deviation of 0.02 seconds for a total of 100 packets.

Next, the packet loss rate metric defined according to Equation 1, quantifies the reliability of the physical layer. For a sequence of 1000 transmitted packets, we observed a packet loss rate of 0.375.

$$Packet\ Loss = 1 - \frac{\#\ of\ Received\ Packets}{\#\ of\ Sent\ Packets} \qquad (1)$$

Lastly, through a call to our audio modem's *sendAll* API, we measured the time to reliably send a fixed number of packets to understand the overhead incurred by our transport layer due to packetization and transcoding. The average time observed for five trials to send 100 packets – retransmissions and receipt of all ACKs included – was 14.9 minutes (standard deviation of 1.6 minutes).

Due to varying packet loss rates – potentially from changes in the cellular channel over time, we observed some variability in the time observed on the NoSSN protocols. We emphasize that our goal is not to optimize the data-over-audio channel, but rather to study the feasibility and practicality of NoSSN protocols. Moreover, Dhananjay et al [54] claim to have achieved a 1.2 kbps goodput with their Hermes implementation.

## 4.10 DISCUSSIONS

### 4.10.1 *Deployment Challenges*

**Ease of deployment by service entities and dongle adoption by customers.** Under our deployment model, we assumed that service entities can easily and readily set up additional web pages for dongle enrollment. We also assumed that the setup of the validation server is relatively easy and scalable for different service entities. However, while we recognize that there will be some challenges with deploying this system in the wild, we do envision that a corporation – like VISA or MasterCard, for example – can champion the deployment of such a system at a small fee to the corporations while ensuring a standardized platform across the board to guarantee dongle compatibility with multiple services.

Moreover, we are yet to see if consumers or service entities will be incentivized to adopt NoSSN as the de-facto means of authentication. This is partly due to the fact that users may not be thrilled to have yet another hardware dongle to carry around or concern themselves about, in terms of their security and privacy. Lastly, we do recognize that people as well as corporations may be averse to any sort of change in the way authentication is carried out today.

**Token Attestation.** In the event that there is a competitive market for the deployment of NoSSN dongles – similar to the kind of market available for credit card companies like VISA, MasterCard, AmericanExpress, etc., each competitor can burn an x509 cert signed by the dongle

manufacturer on each dongle (during manufacture) to provide some device attestation for dongles supported by cooperating service entities.

**Family Fraud.** To prevent family fraud, we propose a password/pin/biometrics protected dongle as we would want to tell apart members of a family whenever they make a claim through the customer service representative. This feature comes in handy with shared family plans where each member of the family can have a password/pin/biometrics protected dongle that would be needed to authenticate themselves. While we acknowledge that there are drawbacks with this approach in terms of security and usability, we consider the approach sufficient for our purposes.

### 4.10.2 *Limitations*

**Compromised host machine or USB vulnerability.** The enrollment of a dongle for a particular service currently requires the user to connect the dongle to her computer via the USB port for the exchange of tokens. With this approach, there is a potential threat to the integrity of the enrollment process with the possibility that the user's host machine may contain malware capable of compromising the dongle or stealing the shared secrets. Additionally, a vulnerability in the USB protocol stack also raises integrity concerns as data belonging to several services may be at risk of theft. The question of how to securely go about dongle enrollment, hence remains.

To mitigate the issue of a compromised host machine, a potential solution would be to have trusted host machines installed at every outlet of a service entity for customers to come enroll their dongles. The usability of NoSSN however suffers with this approach and the "trusted" host machines are themselves vulnerable to external attackers.

Other interfaces such as NFC may be used in addition to trust host machines to address potential vulnerabilities with the USB protocol stack. We however note that, these other interfaces are not without potential security threats.

**Protocol rollback attacks.** There exists a possibility for an attacker to successfully mount vishing attacks against an unsuspecting victim. An obvious approach for an attacker would be to feign oneself as a customer service representative and claim that the service entity's verification server is down, resulting in a need to revert to the traditional way of authenticating a user. The attacker is then able to get the appropriate answer to the knowledge based question and mount a

similar attacker on the actual customer service representative claiming that the customer's dongle is damaged. With this method, an attacker is then able to impersonate the legitimate account holder without appropriate credentials and without physical access to the dongle.

One feasible way to counter this attacker is to stipulate that customers and customer service representatives are always required to use the NoSSN dongle for authentication. In the event that a dongle is legitimately misplaced, stolen or damaged, the customer would have to re-provision another dongle for the service. Under this model, all the service entities are obligated to ensure that their validation servers are never out of commission.

**Service Entity Compromise.** What happens in the event of a server compromise of one of the subscribed services? Can the attacker now impersonate customers through the compromise? The answers to these questions are two fold depending on if the attacker has read or write access to the backend.

With a read-only access, the attacker is able to impersonate a user only if they have access to an authentication session of the user. The goal of the attacker in this case is to deduce the salt – retrieved and sent during every authentication session – as he already has access to the shared symmetric key (for the particular user) or the private key of the server and public key of the user – and can hence, compute the MAC key of the user.

With a write access, the attacker can change the hash of the salt to whatever he wishes and hence, does not need to have access to any of the users' authentication sessions to impersonate them.

In summary, an attacker is able to impersonate a user with various levels of ease that depend on the type of service entity compromise. We do however stress here, that the two NoSSN protocols described in this Chapter were designed to be used for low bandwidth scenarios. For higher bandwidth applications, more robust authentication schemes like the FIDO Alliance standard [32] can be used in their place with some modification to support two-way authentication.

**Malicious Phone App.** NoSSN does not protect against a malicious phone app that can interrogate the dongle and relay information through the attacker to the customer service representative. This attack however, would require the legitimate user to plug in the dongle and

authenticate herself to it before such a malicious app can relay any information. Besides, the malicious app will only be able to get information about the salt and user ID but may not be able to impersonate the user as the shared symmetric key or private key of the dongle remains secure on it, except the malicious app hijacks a call for which the user has been authenticated.

**Active Network Attacker.** As mentioned in Section 4.4, NoSSN and its associated actors trust active network attackers. We do acknowledge that an active network attacker is capable of violating the authenticity of NoSSN by hijacking the call once the user has been authenticated. Moreover, considering the fact that NoSSN offers no protection of the channel, the resulting communications may not be authentic, despite NoSSN's secure implementation. We however note here that the user's sensitive data, such as the last four digits of her SSN, is kept secure from the attacker in the event of a breach. One way to address this issue is to employ more robust authentication protocols – at the cost of a low-bandwidth design – with more elegant use of public key cryptography, such as the FIDO Alliance standard [32].

## 4.11 SUMMARY

In this Chapter, we studied the lack of trust in the way people communicate and choose to share data with other parties over a phone call. We proposed NoSSN, a two-way authentication protocol and phone agnostic approach of establishing the identities of communicating parties as a prelude to sharing sensitive information over the phone. We discussed the design and implementation of two low-bandwidth NoSSN protocols and evaluated their performance on an actual live cellular voice network. We gathered metrics on the total amount of bytes sent and received as well as a profile of the average authentication duration for multiple phases in the authentication process. We also discussed limitations of our approach and potential solutions to address attacks that circumvent it. Despite these limitations, NoSSN contributes to addressing a class of security and functional goals that current authentication approaches fail to address.

We conclude this Chapter by noting that NoSSN provides a novel and deployable approach – as we showed – for communicating users to verify each other's identity without the use of conventional privacy revealing information.

# Chapter 5. CONCLUSION

Mobile phones are pervasive in almost every area of societal interactions, and they provide great benefits and convenience to consumers, from basic telephony communication to mobile payments and health monitoring. The mobile phone market contains devices that range in complexity, cost, and offerings, which include smart, feature or basic phones.

However, these ubiquitous devices are not without security and privacy vulnerabilities which can leave users open to data theft. To address this concern, we focused our research on leakage channels in modern mobile operating systems and the sharing of sensitive data via mobile devices. Our work also explored the shortcomings in current approaches to deal with leakage channels and validate users with whom data is shared.

In this dissertation, we examined the lack of trust in mobile devices and the systems that use them. We modified the phone by separating components in order to isolate data and created a mechanisms through which trust may be established on mobile devices used for both work and personal purposes. We further added external hardware to the phone in order to separate trust among users. In this vein, we explored establishing trust as a prelude to sharing information with other parties over a phone call.

The result of our work shows that it is possible to attain trust through component isolation on mobile operating systems without compromising the usability of the device. Secondly, we discovered that trust between users communicating via a phone call can be attained without radically changing the design of a phone and the accompanying user experience, through the use of additional hardware. Lastly, we suggest that our work on the use of additional hardware, with further improvements, can be extended to address the lack trust in a myriad of other application areas, such as mobile banking, home automation systems and encrypted file sharing, to name a few.

In summary, trust is being increasingly demanded in mobile devices and associated systems, and our work helps lay the groundwork for the further development of functional and marketable solutions to meet such demands. Our solutions, we envision, will influence the design of future mobile operating systems and usable accessory hardware that bolster trust within and external to the mobile device. Moreover, through our work, the mobile device, irrespective of caliber, has taken a major step in becoming a trusted computing and communication platform. It is our hope

that the research community will be spurred on through our work, to address trust issues pertinent to applications areas that utilize additional hardware in conjunction with mobile devices, e.g., mobile banking, wearables and the Internet of Things.

BIBLIOGRAPHY


[1] Rob Cole and Rob Waugh, "Top 'free' Android apps secretly leak users' private contact lists to advertising companies," Daily Mail, 6 March 2012. [Online]. Available: http://www.dailymail.co.uk/sciencetech/article-2110599/Top-free-Android-apps-leak-users-private-contact-lists-advertising-companies--asking-permission.html.

[2] Purdue University, "Warning–Scam Targets International Students," 31 July 2013. [Online]. Available: http://webs.purduecal.edu/intl/warning-scam-targets-international-students/.

[3] Hu, Sofia, "Scam Targeting International Cornellians Persists, Police Say," The Cornell Daily Sun, 16 October 2014. [Online]. Available: http://cornellsun.com/blog/2014/10/16/scam-targeting-international-cornellians-persists-police-say/. [Accessed 19 August 2015].

[4] IDC, "Smartphone OS Market Share, Q1 2015," IDC, 2015. [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[5] G. Russello, M. Conti, B. Crispo, and E. Fernandes, "MOSES: Supporting Operation Modes on Smartphones," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies, SACMAT '12, pages 3–12*, 2012.

[6] G. Russello, M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "Demonstrating the effectiveness of moses for separation of execution modes," in *Proc. of the 2012 ACM Conference on Computer and Communications Security, CCS '12, pages 998–1000. ACM*, 2012.

[7] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10), pages 393–407. USENIX Association*, 2010.

[8] Divide, "Divide," Divide, [Online]. Available: http://www.divide.com/. [Accessed March 2014].

[9] WorkLight Inc., "WorkLight Mobile Platform," [Online]. Available: http://www.worklight.com/. [Accessed March 2014].

[10] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks," Technical Report TR-2011-04, Technische Universität Darmstadt, 2011.

[11] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," Proc. of the 13th International Conference on Information Security, ISC'10, pages 346–360. Springer-Verlag, 2011.

[12] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. of the 20th USENIX Conference on Security, SEC'11, pages 22–22. USENIX Association*, 2011.

[13] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones," in *Proc. of the 18th Annual Network and Distributed System Security Conference (NDSS'11)*, 2011.

[14] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: A virtual mobile smartphone architecture," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, pages 173–187. ACM*, 2011.

[15] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis, "The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket?," in *SIGOPS Oper. Syst. Rev., 44(4):124–135*, 2010.

[16] O. K. Labs, "Ok:android," O. K. Labs, [Online]. Available: http://www.ok-labs.com/products/ok-android/. [Accessed 23 January 2015].

[17] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4Android: A Generic Operating System Framework for Secure Smartphones," in *Proc. of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '11, pages 39–50*, 2011.

[18] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones," in *Proc. of the 12th Workshop on Mobile Computing Systems and Applications, HotMobile '11, pages 21–26. ACM*, 2011.

[19] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and Lightweight Domain Isolation on Android," in *Proc, of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '11, pages 51–62. ACM*, 2011.

[20] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra, "A virtual machine-based platform for trusted computing," in *Proc. of the 19th ACM Symposium on Operating Systems Principles, SOSP '03, pages 193–206*, 2003.

[21] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10, pages 143–158. IEEE Computer Society*, 2010.

[22] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker, "An Execution Infrastructure for TCB Minimization," in *Proc. of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, Eurosys '08, pages 315–328*, 2008.

[23] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications," in *Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, 2014.

[24] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot and Frank Stajano, "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes," *IEEE Symposium on Security and Privacy (SP)*, pp. 553 - 567, 2012.

[25] A. Menezes, P. van Oorschot, and S. Vanstone, "Identification and Entity Authentication," in *Handbook of Applied Cryptography*, CRC Press, 1997, pp. 385-424.

[26] Scott Garriss, Ramon Caceres, Stefan Berger, Reiner Sailer, Leendert van Doorn and Xiaolan Zhang, "Trustworthy and personalized computing on public kiosks," in *MobiSys '08 Proceedings of the 6th international conference on Mobile systems, applications and services*, 2008.

[27] Goodrich, M.T.; Sirivianos, M; Solis, J; Tsudik, G; Uzun, E., "Loud and Clear: Human-Verifiable Authentication Based on Audio," in *26th IEEE International Conference on Distributed Computing Systems*, 2006.

[28] K. Silvester, "Apparatus and method for wireless device set-up and authentication using audio authentication—information". US Patent US7254708 B2, 7 August 2007.

[29] Alexei Czeskis, Michael Dietz, Tadayoshi Kohno, Dan Wallach and Dirk Balfanz, "Strengthening user authentication through opportunistic cryptographic identity assertions," in *CCS*, 2012.

[30] Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse and Peter Rutenbar, "Device-Enabled Authorization in the Grey System," in *Proc. 8th Int',l Conf. Information Security (ISC 05)*, 2005.

[31] Lingjun Li, Guoliang Xue and Xinxin Zhao, "The Power of Whispering: Near Field Assertions via Acoustic Communications," in *Asia CCS '15*, 2015.

[32] FIDO Alliance, "Fido Alliance," [Online]. Available: https://fidoalliance.org/.

[33] Gartner, "Gartner Predicts by 2017, Half of Employers will Require Employees to Supply Their Own Device for Work Purposes," Gartner, 1 May 2013. [Online]. Available: http://www.gartner.com/newsroom/id/2466615.

[34] Zielinski, D., "Bring your own devices.," *Society for Human Resource Management,* vol. 57, no. 2, 2012.

[35] Kaneshige, Tom, "BYOD: Time to Adjust Your Privacy Expectations," CIO, 30 May 2012. [Online]. Available: http://www.cio.com/article/2395604/byod/byod--time-to-adjust-your-privacy-expectations.html.

[36] McNickle, Michelle, "BYOD Security Tops Doctors' Mobile Device Worries," Information Week, 31 October 2012. [Online]. Available: http://www.informationweek.com/mobile/byod-security-tops-doctors-mobile-device-worries/d/d-id/1107153?.

[37] K. Miller, J. Voas, and G. Hurlburt, "BYOD: Security and Privacy Considerations," *IT Professional*, vol. 14, no. 5, pp. 53-55, 2012.

[38] Hill, Kelly, "BYOD grows, stirs privacy concerns," RCR Wireless News, 28 September 2012. [Online]. Available: http://www.rcrwireless.com/20120928/wireless/byod-grows-privacy-concerns.

[39] United States District Court, S.D. Texas, Houston Division, "SAMAN RAJAEE, Plaintiff, v. Design Tech Homes, LTD and Design Tech Homes of Texas, LLC, Defendants.," Google Scholar, 11 November 2014. [Online]. Available: https://scholar.google.com/scholar_case?case=17791284937819645574.

[40] Garry G. Mathiason et al, "The "Bring Your Own Device" to Work Movement," Littler, May 2012. [Online]. Available: http://www.littler.com/bring-your-own-device-work-movement.

[41] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang, "Accessory: Password inference using accelerometers on smartphones," in *Proc. of the 12th Workshop on Mobile Computing Systems and Applications, HotMobile '12, ACM*, 2012.

[42] D. Genkin, A. Shamir, and E. Tromer, "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis," Cryptology ePrint Archive, Report 2013/857, http://eprint.iacr.org/, 2013.

[43] Y. Xu, F. Bruns, E. Gonzalez, S. Traboulsi, K. Mott, and A. Bilgic, "Performance evaluation of para-virtualization on modern mobile phone platform," in *Proc. of International Conference on Computer, Electrical, and Systems Science, and Engineering (ICCESSE '10)*, 2010.

[44] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-Automation for Large Scale Dynamic Analysis of Mobile Apps," Technical Report 14-941, University of Southern California, 2014.

[45] 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 *Proc. of the 18th ACM Conference on Computer and Communications Security, CCS '11, pages 639–652. ACM*, 2011.

[46] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *USENIX Security Symposium*, 2013.

[47] Alessandro Acquisti, Ralph Gross, "Predicting Social Security numbers from public data," *Proceedings of the National Academy of Sciences*, vol. 106, no. 27, pp. 10975 - 10980, 2009.

[48] Abhilasha Bhargav-Spantzel, Anna Squicciarini and Elisa Bertino, "Privacy preserving multi-factor authentication with biometrics," in *DIM '06 Proceedings of the second ACM workshop on Digital identity management*, 2006.

[49] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan and Akshay Wadia, "Founding Cryptography on Tamper-Proof Hardware Tokens," in *7th International Conference on Theory of Cryptography*, Zurich, 2010.

[50] John Scott Railton and Katie Kleemola, "London Calling: Two-Factor Authentication Phishing From Iran," The Citizen Lab - University of Toronto, 27 August 2015. [Online]. Available: https://citizenlab.org/2015/08/iran_two_factor_phishing/.

[51] Union Bank, "Bank Securely by Phone," [Online]. Available: https://www.unionbank.com/personal-banking/checking-savings/checking/telephone-banking.jsp.

[52] RSA, "RSA SecureID Hardware Tokens," [Online]. Available: http://www.emc.com/collateral/data-sheet/h13821-ds-rsa-securid-hardware-tokens.pdf.

[53] Sasse, Kat Krol and Eleni Philippou and Emiliano De Cristofaro and Martina Angela, "They brought in the horrible key ring thing!" Analysing the Usability of Two-Factor Authentication in {UK} Online Banking," *CoRR*, vol. abs/1501.04434, 2015.

[54] Dhananjay, Aditya and Sharma, Ashlesh and Paik, Michael and Chen, Jay and Kuppusamy, Trishank Karthik and Li, Jinyang and Subramanian, Lakshminarayanan, "Hermes: Data Transmission over Unknown Voice Channels," in *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, Chicago, Illinois, 2010.

[55] Wagner, David and Schneier, Bruce, "Analysis of the SSL 3.0 Protocol," in *WOEC'96*, Oakland, 1996.

[56] Daniel J. Bernstein, "Curve25519: new Diffie-Hellman speed records," in *Public Key Cryptography - PKC 2006*, Springer Berlin Heidelberg, 2006, pp. 207-228.

[57] Temitope Oluwafemi and Earlence Fernandes and Oriana Riva and Franziska Roesner and Suman Nath and Tadayoshi Kohno, "Per-App Profiles with AppFork: The Security of Two Phones with the Convenience of One," Microsoft, December 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=246811.

[58] Android AppBrain Stats., "Most Popular Google Play Categories," [Online]. Available: http://www.appbrain.com/stats/android-market-app-categories.

# VITA

Temitope Oluwafemi, an EE PhD student at the University of Washington was advised by Tadayoshi Kohno and Franziska Roesner of the Security and Privacy Research lab. His interests include the security and privacy of mobile operating and home automation systems. He is also very interested in the design and implementation of robust, secure and usable systems in developing regions. In his spare time, Temitope Oluwafemi enjoys playing and watching soccer and travelling.