Using the SIM as a Trusted Element to Secure the Mobile Web

Henry Irish, Trinity Hall
Computer Science Tripos Part III

13 June 2013
Abstract

This project describes the design and implementation of a system to secure web browsing for Android phones. It prevents or mitigates against a set of common issues and vulnerabilities encountered with mobile (and desktop) browsing. The project is separated into three parts: a cloud application containing secure user data; a daemon application for hijacking user connections; and the SIM acting as a secure element and the main part of our trusted computing base.

Declaration

I, Henry Irish of Trinity Hall, being a candidate for Part III of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date Thursday, 13 June 2013
# Contents

## 1. Introduction

1.1 The aim

1.2 Malware attacks on Android
   1.2.1 Phishing
   1.2.2 Session Hijacking

1.3 Usability issues with Android
   1.3.1 Entering passwords
   1.3.2 Use of HTTPS

1.4 SSL certificate validation

1.5 Tracking of users
   1.5.1 Advertiser networks
   1.5.2 Government and ISPs

1.6 Blocking malicious sites

1.7 Mobile trusted execution environments
   1.7.1 The SIM as a secure element
   1.7.2 Other types of secure element

## 2. Design

2.1 Threat models
   2.1.1 Privacy conscious user
   2.1.2 Malware infested device
2.1.3 Fully compromised device
2.1.4 Attack on cloud service
2.1 Design goals
2.2 Overview
2.3 High level design
   2.3.1 Features
2.4 Cryptographic design
   2.4.1 Basic request
   2.4.2 Decrypting cloud data
   2.4.3 Cookies
2.5 Similar projects
   2.5.1 MITM Proxy, BURP Proxy and Privoxy
   2.5.2 Digital wallets
   2.5.3 Password managers
   2.5.4 Ghostery and Adblock
3. Implementation
   3.1 Routing connections into the system
      3.1.1 Hijacking HTTP
      3.1.2 Passing to the cloud and web-application
      3.1.3 Hijacking HTTPS
      3.1.4 Threading for scalability
      3.1.5 Daemon application
   3.2 Secure element
      3.2.1 API
      3.2.2 Coding for the SIM
      3.2.3 Limitations
   3.3 Communication between components
      3.3.2 Communication between the cloud and daemon
3.3.1 Communication between the cloud and SIM

3.4 Bootstrapping the system

3.5 Automated login and registration
   3.5.1 Detecting login pages
   3.5.2 Making login requests
   3.5.3 Registration

3.6 Cookie interception
   3.6.1 Continuing sessions

3.7 Blocklists
   3.7.1 Malicious sites
   3.7.2 Certificate validation
   3.7.3 Adverts
   3.7.4 Preventing tracking

4. Evaluation
   4.1 Testing
      4.1.1 Custom sites
      4.1.2 Real world sites
   4.2 Threat models revisited
      4.2.1 Privacy conscious user
      4.2.2 Malware infested device
      4.2.3 Fully compromised device
      4.2.4 Attack on cloud service
   4.3 Benchmarks
   4.4 Limitations
   4.5 Advantages
   4.6 Disadvantages

5. Conclusion
   5.1 Future directions
Bibliography

A. Project Proposal

A.1 Introduction and Description of Work
A.2 Starting Point
A.3 Substance and Structure of the Project
A.4 Evaluation
A.5 Timetable

B. Sample Trace
Chapter 1

Introduction

1.1 The aim

This paper discusses the creation of a system to secure web browsing for Android phones by preventing or defending against a set of issues and vulnerabilities; each of which will be discussed later in this chapter:

1. Phishing attacks.
2. Tracking of users online.
3. Password authentication.
5. Session hijacking.
7. HTTPS encryption.

To accomplish this, we remove the user’s personal data from their device and store it in a cloud layer, although the user retains control. The project is separated into three parts. The SIM is used as a secure element, acting as a key to the cloud data. The cloud service is where more resource intensive computations involved within the system are done. This ensures good performance even within the limited capabilities of modern day SIM and phone technology. Finally, a daemon service acts as a transport intermediary between these components, as well as a user gateway into the system.

1.2 Malware attacks on Android

As recent mobile devices have become more powerful, they have become indispensable for users, who fill them with valuable personal data. Consequently they have become a target for attackers, and malware is increasingly becoming a problem. Malware can appear through several distribution mediums — through App Stores (e.g. the Google Play store) or through zero-day attacks against the
various communication systems of a device. A common, non-malicious, example of the latter is the iOS jailbreak scene. This project explores how we can protect users in compromised environments.

Although here are many types of Android malware [6], this project focuses on those that steal identities and web-service account credentials, as described below.

1.2.1 Phishing

Phishing is a criminal activity focused on stealing user's personal information [5]. Most commonly this information comprises the user's identifying information, authentication details for web applications, and financial information. Historically, phishing attacks have been web based or targeted user's emails, but malware is now a common attack vector.

A 2011 study [8] found that 60% of the malware applications studied included capabilities to steal user information. One vulnerability was demonstrated at Defcon-19 [7] allowing malware to steal UI focus, with the attackers demonstrating an attack against sites including Facebook and Google.

Phishing attacks are usually mitigated via filtering, via blacklists, or machine learning [9]. Though these exist for mobile browsers (iOS Safari warns when it detects fraudulent websites), they are of no use against malware looking at or loading local content.

1.2.2 Session Hijacking

Figure 1 — FireSheep in action

When a user authenticates with a web application, the application often issues a session token (stored as a cookie) to perform future authentication cheaply. Unfortunately, many [10] websites use HTTP rather than the more secure HTTPS and this token is therefore sent in plaintext. Attackers
can steal this token and take over the session, impersonating the user. A famous attack showcased this vulnerability; the famous Firefox plugin FireSheep (shown in figure 1). Even if the session tokens were originally sent over HTTPS, malware with root access could hypothetically extract session tokens stored in the browser.

1.3 Usability issues with Android

As a requirement, usability often conflicts with security. This is exacerbated in mobile devices where usability is already significantly worse due to smaller screen sizes and less accurate input mechanisms. This results in greater compromises — one of most obvious examples being password entry, which is discussed below.

Users also have different expectations from mobile OSs and applications. Things are expected to “just work”, and minimal configuration is important for a security solution to be effective [12].

1.3.1 Entering passwords

Perhaps the best illustration of the compromise between usability and security is found when a user authenticates with a device. This is done on PCs using a user-defined password. However, inputting text into a mobile device is comparatively clunky and therefore designers have improved usability (to the detriment of security), by instead using PINs made from four numeric characters, or even abandoning authentication altogether (e.g. Viber and WhatsApp).

Other compromises are made. Devices rely on touchscreen keyboards, which are less accurate than their larger hardware counterparts and users are more likely to make mistakes when entering their password. Common security techniques such as masking the inputted password only exacerbate this and designers have compromised by unmasking the most recently typed character, encouraging attacks like shoulder surfing.

1.3.2 Use of HTTPS

Using HTTPS prevents request data being sent in plaintext and potentially intercepted. The majority of websites do not force HTTPS by default [10], requiring users to explicitly request it (this is exacerbated when users follow links from other pages). Although this usability issue has been mitigated by browser extensions in desktop browsers, it is untouched among mobile devices, which require correctly configured servers to automatically upgrade connections to HTTPS.
1.4 SSL certificate validation

Certificate validation is implemented on a per-browser basis. Browsers are packaged with a set of root certificates from a set of trusted vendors. These vendors then sign website certificates, after a varying series of checks (which might include domain validation — DV or extended identity checks — EV). When a browser connects to a site, it verifies the certificate received has been signed by a trusted CA before showing the page.

Browsers are therefore essentially in control of SSL certificates. If a root certificate is not bundled, then the average user will receive an error page when connecting to sites which use it, as shown in figure 2. Browser vendor's incentives are not necessarily aligned with those of the user, although most major browsers do not profit from this, while Governments typically have their own certificates for legal interception.

![The site's security certificate is not trusted!](image)

*Figure 2 — Error page by Google Chrome shown on untrusted websites.*

Certificate revocation is the mechanism by which trust for a certificate can be withdrawn. Two methods exist to revoke a certificate — via OCSP (online certificate status protocol), and via checking CRLs (certificate revocation lists). However, browsers do not necessarily implement these protocols correctly — a recent revocation was ignored for over a week [2]. This is a problem both on desktops and mobile devices, where the majority of browsers (including Firefox, Chrome with default settings and Safari) do not check for revocation unless the certificate is EV. End users could be receiving fraudulent/tampered-with websites while under the assumption they are secure.
1.5 Tracking of users

This project focuses on two types of tracking — commercial tracking done by advertisers, and non-commercial tracking done by entities like governments.

1.5.1 Advertiser networks

There are several major advertising networks, including Google and AdMob, tracking users through the web with cookies sent alongside their advertisements. They also look at the referral headers on the adverts which are loaded. Facebook and other social networks do similarly with “Like” buttons inserted into pages. This data gives these companies a commercial advantage in targeting adverts and content towards users.

The data is also sold and can even be inferred by third parties [39]. One recent solution to this is the DNT (do not track) header, which tells web applications that the user does not want to be tracked. However, this provides no guarantees to the user — the server can choose whether or not to heed it, and also provide no indication whether they do.

1.5.2 Government and ISPs

In recent years state surveillance of web communications has significantly increased, both in countries with authoritarian regimes [14], and in more recently in western states. ISPs are also tracking and filtering the content they serve, as well as selling it on [17]. Although sold data is “anonymised”, this is very difficult to effectively do [16].

The common solution against these kinds of attacks is using a proxy service, which provides anonymity by placing users in a crowd. More sophisticated avoidance techniques exist, e.g. tor [15], but this is beyond the scope of this project.

1.6 Blocking malicious sites

Many online sites are malicious, and should be blocked to protect the average user. Some attempt to phish users, while others contain links to malware, or can even exploit the browser. Although non-malicious, jailbreak.me demonstrates the problem. It utilised a zero day exploit in iOS’s PDF handling capabilities to gain root on Apple devices. Some sites can be unpleasant or undesirable for social or legal reasons. Lastly, some users like to block advertisements. This is possible with desktop plugins [18], but is not possible on mobile.
1.7 Mobile trusted execution environments

One of this project’s goals is to ensure the user keeps control over their data, rather than handing it off to a “trusted” third party. This is impossible to do on an insecure device, and so we use either a trusted execution environment (TEE) or secure element (SE). TEEs are usually isolated parts of the device processor, while SEs are typically on another hardware chip. They typically are used to provide a secure API for security critical functions. When TEEs/SEs are used properly they can be add secure capabilities to an inherently insecure device.

1.7.1 The SIM as a secure element

We focus on using the SIM as a secure element. Some reasons are obvious; they are ubiquitous, low cost and homogenous across devices. A subtlety when developing for SIMs is the issue of control — they are typically tightly controlled by network providers, who decide which applications can run on them. This could provide a barrier to widespread adoption.

![Diagram](image)

*Figure 3 — Android communication with the SIM. Taken from [3].*
SIM cards are a type of JavaCard, and are therefore completely isolated from the OS. The SIM also firewalls applets from one another, ensuring our applet can be trusted. However, they run on relatively limited resources — with up to 512KB ROM and 8KB RAM. SIM applications have faded as faster and more capable phones arrive. Nonetheless, SIM applications are still prevalent in African countries, where they are used as e-currency and for public transport ticketing [20]. It is also used in non-smartphones to store contact details and SMS messages.

The SIM is linked to a device’s baseband; which is the firmware found on the device’s cellular modem. The Android operating system communicates with this using a device specific RIL (radio interface layer), with the STK (SIM toolkit) providing higher level communication with the SIM, as shown in figure 3.

1.7.2 Other types of secure element

Many other devices could potentially be used as the secure element in place of SIM cards. Currently they unfortunately all share similar disadvantages when compared to SIMs, both in terms of pricing and availability.

One of the current major pushes in TPM technology is from GlobalPlatform, who hope to create both TEEs and SEs [23]. GlobalPlatform is working on creating standards for these secure chips, and have published several API specifications [24]. These have the advantage of being stored within the device, and are also custom designed for purpose. They should therefore be more capable than SIMs, although at greater cost. Unfortunately they are in development and are not yet available.

There is no requirement that the TPM is internal. A handheld dongle, such as Stajano’s Pico [22], or a touchscreen watch could be paired to the mobile device through Bluetooth. This has several advantages. Firstly, the TPM is no longer tied to a single phone, but could be used by all of the user’s devices concurrently, sharing his sessions between them. However, an external device is an extra thing to carry, as well as more expensive and less ubiquitous and than an internal chip.

One interesting aspect of the above devices is that they may be able to provide a trusted path from the SE/TEE to the user. A trusted path is an authentic communications medium between the user and SE. An implementation might involve the SE/TEE taking direct control of the hardware/screen. Trusted paths are useful in determining whether requests originated from the user or from malware.
Chapter 2

Design

2.1 Threat models

When designed a security application it is important to look at the types of threat models that exist, to ensure a proposed design protects against them. We look at them in increasing severity.

2.1.1 Privacy conscious user

The simplest threat model is a user who is concerned about upstream traffic being monitored by a third party like a malicious government or when using insecure WiFi. In this model we assume that the user’s device is not compromised, although their connection may be. Using HTTPS would prevent a message’s details being leaked, however an attacker can still find out who a message is to.

This threat model is easy to protect against — tunnelling and encrypting connections through another location protects against most third parties. When more powerful organisations are involved then privacy can be increased using Tor [15], although this is beyond this project’s scope.

2.1.2 Malware infested device

Perhaps the most common threat model is a device which contains malware, but where the malware has not gained root access. This type of malware can still be capable, as described in section 1.2, particularly since users do not typically check permissions on installation [28]. Phishing is a large problem in this category — applications can easily serve fake webpages with apparently genuine UI elements, as well impersonating other applications to steal login details. This project counters this...
by ensuring the user never authenticates or inputs personal details on their device again, as
discussed further in section 2.2.

Within the scope of this project, malware in this category may be able to make malicious requests.
There is no way for the daemon application to discern between a user request and a malware
request. We mitigate this by ensuring the user confirm with the daemon when the SIM API is used.

2.1.3 Fully compromised device

An extreme threat model is where a user's device has been fully compromised — i.e. malware has
root access. This means that traditional security controls (e.g. application sandboxing) fail, and the
malware can access/alter all data stored on the device, even altering the device OS. Any security
policies relied upon within the OS or in the daemon application are vulnerable — they can be
removed by the malware. It is even more difficult to distinguish input and requests made by the
malware from those made by the user, since any daemon-confirmation cannot be trusted.

One way that this kind of attack can be mitigated is with a trusted path from the secure element to
the user, ideally via a screen. This could be through the device, in the case of the SIM or a
specialised chip, or via some external pathway like the screen on a Pico [22] or other device. The
secure path can be used to differentiate user interaction with requests from malware, since the user
can authentically verify each transaction.

2.1.4 Attack on cloud service

The design, outlined below in section 2.3, utilises a cloud service. This adds a further avenue of
attack. We assume that the owner of the cloud service is non-malicious (the user selects a provider
they trust) and that the server could eventually be compromised, but that the provider will discover
this — i.e. we design for one off attacks which might, for example, dump data. It is therefore
important that data is not stored on the server in a permanent extractable form.

Although it is hypothetically feasible that an attacker could crack the service and install software
which collects decrypted data, this is significantly more difficult for the attacker. It is also currently
a risk user's live with and accept when interacting with normal services via HTTPS.

2.1 Design goals

There are project goals beyond fixing the above threat models, focusing on both technical and
usability challenges:

**Transportable:** Moving the secure element (i.e. the SIM) between devices should move the
user’s session and data to the new device. This provides another incentive to use the system.
**Trust in service provider:** It is important that we minimise the amount of trust a user has to place in the provider of the cloud application. Therefore we ensure that it is both impossible for the service provider to access their data. This prevents disaster in the event of a server compromise (e.g. a database dump). It is also important that the cloud software is released such that a user can run their own servers.

**Secure element API:** Large APIs are prone to bugs, and have been the downfall of other trusted execution environments [29]. Minimising the size of API both decreases the likelihood of security critical bugs, but also makes the development of different secure elements easier.

**Robustness:** This system becomes the user's main communications route and must be robust. Connections are not always wellformed, and user behaviour can be unpredictable (e.g. stopping a request before completion). One example is that the cloud service should be as stateless as possible, ensuring the ordering of requests is unimportant, and that failures are handled gracefully.

### 2.2 Overview

The overall idea of this project is to remove the user data that is normally either stored on the device or entered into the device, and push it to a secure cloud service. The secure cloud service acts as an external security layer around the device, inserting the aforementioned user data or authenticating with web applications as required. User data is stored encrypted in the cloud, with each user's SIM acting as the key to access it. Since the SIM is physically in complete user control, this ensures that the data remains within the user's hands. Cookies and session-tokens are also treated as user data, and are therefore intercepted by the cloud service. This ensures the device contains no useful information (which could be stolen), and ensures that a user's session can move with their SIM (if they were to upgrade their phone, for example). A major aim is to reduce mobile phishing, as explained in section 1.2.1.

By using a cloud service as a security layer, we also gain other benefits. Since connections are routed through the cloud, we can prevent the tracking detailed in section [tracking]. We can also block malicious pages, as detailed in section [maliciousPages], and ensure SSL certificate validity. All of this can be implemented in such a way that it is essentially configuration-less, removing one of the main barriers to adoption to current high end security systems. With additional effort, it would also be trivial to allow these settings to be customised by power users who want to extract maximum security from the system.

The trusted computing base therefore includes only the SIM and over the short term, the cloud application (although the cloud cannot be trusted for non transient data).
2.3 High level design

The simplest solution would route all communication through the SIM, which would then add a layer of trust to connections. This is not technically feasible — the SIM does not have enough power of bandwidth, hampering performance, particularly for high traffic applications. A software handoff for these applications would have to exist within the OS, which would itself be vulnerable.

Accepting these limitations, we use the SIM for the bare minimum — as a key store which does cryptographic operations. Each user of the system has a unique identifier, which is stored within their SIM, and identifies the user to the server. The heavy computing load is moved into a cloud application, and a daemon application on the phone is therefore required to coordinate communication between the three parties (cloud, SIM and user). This is demonstrated in figure 4.

The major problems are:

**User-daemon communication**: A simple method must be found to hijack the users HTTP and HTTPS connections and reroute them through this system.

**Cloud-SIM communication**: A protocol must be written to allow communication between the cloud and SIM, via the phone daemon. Since the cloud application is designed to be stateless and cannot initiate a connection with the device, this will either need to piggyback upon user activity, or maintain a constant connection (which would be technically infeasible).

**Decrypting cloud data**: Creating a protocol for decryption of user data stored in the cloud, via the SIM, ensuring that it is never held in permanent storage in the cloud application.

*Figure 4 — High level component connections.*
2.3.1 Features

Once the groundwork is created, it is possible to add the security-enhancing features on top:

**Authentication and registration:** The authentication/registration feature has to automatically detect login and registration forms, filling them and submitting them for the user. For added security, the SE may request confirmation before decoding user data.

**Cookie interception:** Cookies contain private information, including session-keys. These should be intercepted before being passed to the device. We also destroy malicious cookies.

**Blocklists:** A mechanism for detecting both malicious websites, advertisements and other unwanted sites. A major aim for this is in minimising configuration.

**Upgrading HTTP connections:** Users do not typically use HTTPS themselves, but instead require correctly configured servers to upgrade their connections. The cloud service should automatically detect the presence of a HTTPS service and attempt to upgrade if possible.

**Moving device:** The SIM must be able to identify itself, which ensures the cloud is able to continue a user's sessions, and carry over their information when the user switches devices.

2.4 Cryptographic design

This system is designed with the data stored securely in the cloud. This has several requirements:

**Inaccessible to cloud service:** The user data must be stored in the cloud such that it is impossible for the cloud to decrypt the data without the user's permission. Although the decrypted data will pass through the cloud service again, this will be transitory and should never hit a permanent store.

**Inaccessible to device OS:** The overriding aim is to prevent a user's personal details being phished by malware stored on the device's OS. The simplest way to guarantee this is to ensure that the user's details are never accessible to the device OS. This means that any data passing through the device daemon service should be impossible to decrypt on the device.

**Not susceptible to replay attacks:** It is also important to prevent malicious software from being able to make bad requests to web applications. The protocols must prevent a replay attack happening with data collected after it has gone through the SIM.

**No decryption oracle:** Decryption requests to the SIM must be authentic, preventing malware from using the SIM as a decryption oracle.
Encryption is handled via RSA [25], and specifically the PKCS#1 standard with OAEP (optimal asymmetric encryption padding). This is an asymmetric encryption algorithm — keys have both public and private components. Anyone in possession of the public key can encrypt messages, but only those with the private key can decrypt messages.

Traditionally RSA is therefore used to bootstrap a key for computationally cheaper, symmetric AES encryption. Unfortunately, this is not an option. One of the requirements is that data stored in the cloud is inaccessible to the cloud. It therefore cannot be stored encrypted with a symmetric key, since this would allow the cloud to decrypt the data. We instead store it encrypted with appropriate SIM’s public RSA key. Although RSA is computationally more expensive than AES, this cost is dwarfed by 3G data latencies.

In the following, we define \( U \) as the user (or a browser acting under the user), \( C \) as the cloud application, \( S \) as the SIM, and \( W \) as the web application. The \( \rightarrow \) symbol can implicitly imply that a message passes through the daemon service, e.g. \( U \rightarrow C \) is actually \( U \rightarrow \text{Daemon} \rightarrow C \). \( \{M\}_S \) implies a message \( M \) encrypted with the public key generated by \( S \). Without loss of generality, we discuss protocols with reference to a single SIM — this system is trivially extensible to multiple users/SIMs.

### 2.4.1 Basic request

We assume a basic request to resource \( X \) looks as below:

1. \( U \rightarrow C \) GET \( X \)
2. \( C \rightarrow W \) GET \( X \)
3. \( W \rightarrow C \) \( X \)
4. \( C \rightarrow U \) \( X \)

This provides two places where requests can be altered or blocked by the cloud — at steps 2 and 3.

### 2.4.2 Decrypting cloud data

In this system, the cloud and each SIM has a distinct 2048 bit private key. The cloud application has a database containing all SIM public keys, while each secure element contains the cloud public key. This implies that we can have end-to-end encryption on the communications channel. Bootstrapping of these keys is assumed to be done at SIM creation.

The simplest decryption protocol is shown below. The message is returned from the SIM encrypted with the cloud’s public key, preventing interception as it passes through the daemon process.

1. \( C \rightarrow S \) \( \text{DECRYPT} \ \{M\}_S \)
2. \( S \rightarrow C \) \( \{M\}_C \)
Unfortunately, although the above protocol does not leak information, it is susceptible to replay attacks — the attacker could potentially cause the request to be repeated by repeating step 2. We therefore introduce a random nonce to each message — $T$, locking it to the specific message by encrypting them together with the SIM’s public key. The cloud service stores a list of active nonces, tied to the decryption request they were created for. A repeated nonce is ignored.

Finally we must ensure that decryption requests to the SIM are authentically from the cloud, and not from malware. This prevents the SIM being used as a decryption oracle. We sign each message $\{M\}_s, T$ using the cloud’s private key to generate a MAC:

1. $C \to S$ \hspace{1cm} \text{DECRYPT} \ \{\{M\}_s, T\}_s$
2. $S \to C$ \hspace{1cm} $\{M, T+1\}_c$

2.4.3 Cookies

Cookies are a special case of the above decryption protocol, since they are stored for multiple use in the browser. The nonce used above works against us here. Rather than developing a secondary protocol, we redo the Set-Cookie for each response with a new nonce. This means that each cookie (as the user sees it) is effectively use-once. A typical cookie-ed request is therefore as follows:

1. $U \to S$ \hspace{1cm} \text{GET} X, Cookie: $\{\{C\}_s, T, \text{MAC}_C\}_s$
2. $S \to C$ \hspace{1cm} \text{GET} X, Cookie: $\{C, T+1, \text{MAC}_S\}_c$
3. $C \to W$ \hspace{1cm} \text{GET} X, Cookie: $C$
4. $W \to C$ \hspace{1cm} $X$
5. $C \to U$ \hspace{1cm} $X$, Set-Cookie: $\{\{C\}_s, T'\}_s$

Obviously, when the server alters this cookie, we account for this:

1. $U \to S$ \hspace{1cm} \text{GET} X, Cookie: $\{\{C\}_s, T, \text{MAC}_C\}_s$
2. $S \to C$ \hspace{1cm} \text{GET} X, Cookie: $\{C, T+1, \text{MAC}_S\}_c$
3. $C \to W$ \hspace{1cm} \text{GET} X, Cookie: $C$
4. $W \to C$ \hspace{1cm} $X$, Set-Cookie: $C'$
5. $C \to U$ \hspace{1cm} $X$, Set-Cookie: $\{\{C'\}_s, T', \text{MAC}_C\}_s$

We reset the cookie every time, allowing us to include a new nonce. Setting up a new cookie is a subset of this protocol. Cookie decryption must happen on-demand, rather than a decrypted cookie $\{C, T+1\}_c$ being stored in the browser — this would allow remove the replay protection.
2.5 Similar projects

Some of the ideas in this project have been explored before. Several proxies have been used as aggressive tools to hijack HTTP/S. Research into digital wallets has attempted to use secure elements for real world applications like payment. Finally, several browser plugins provide various password authentication and blocking features for the desktop.

2.5.1 MITM Proxy, BURP Proxy and Privoxy

All three of these act as proxies to intercept HTTP/S traffic, although with differing goals. BURP proxy is designed for penetration testing, and is the least featured of the three. It allows for dynamically altering requests as they go through. MITM Proxy is perhaps the most flexible of the three and essentially provides a scriptable proxy. Privoxy is focused upon sanitising web-browsing, removing unwanted elements like adverts.

2.5.2 Digital wallets

Digital wallets are used for making financial transactions, and are often linked to a user's bank account. Recent developments have explored their usage with NFC on mobile, and one paper makes note of the use of the STK and SIM as a removable secure element for NFC applications [32]. Older approaches include Anderson's UEPS [33] and NetCard [11], which utilised secure cards and transactions to allow for electronic transactions in developing countries.

2.5.3 Password managers

Several password managers exist for desktops; major examples include LastPass and 1Password. These use a local password store, with a browser plugin to enter the information. Both have attempted to port their plugins to mobile browsers as a Javascript bookmarklets. This solution does not sync data and is particularly unusable — it therefore is not popular. The fact that data is stored locally is both an advantage and a disadvantage. The user is in complete control of their data, although they lose a lot of flexibility when they are switching devices, or when a device is lost.

2.5.4 Ghostery and Adblock

Ghostery and Adblock are browser plugins. Ghostery focuses on preventing users from being tracked across the web, removing bad cookies and javascript from pages before they are displayed (or sometimes even requested). Adblock is simpler, blocking HTTP requests for adverts. However, mobile browsers are not extensible and these solutions are therefore limited to the desktop. Attempts have been made to block adverts on Android, although they are contrary to Google's business model and have recently been blocked from the Google Play store [35].
Chapter 3

Implementation

3.1 Routing connections into the system

First we will outline how different types of request from the device OS are captured by the daemon application, before outlining how we use this technique to create a robust multi-connection system.

3.1.1 Hijacking HTTP

The daemon process intercepts traffic by acting as an HTTP proxy. HTTP proxies are very simple for unencrypted connections — all outgoing requests are directed to the proxy rather than the website. The daemon-proxy is hosted on the phone and listens to local connections on port 8080. HTTP requests are typically formatted as below:

1. User → Proxy GET http://henryirish.com/index.php HTTP/1.1

This implies a HTTP GET request (other common types include POST, HEAD), connecting to the URL http://henryirish.com and requested the page index.php, using the 1.1 version of HTTP. The rest of a request contains various HTTP headers which are not needed at this stage.

3.1.2 Passing to the cloud and web-application

After the connection has been hijacked by the daemon application, it is forwarded to the cloud service, as shown in the protocol described in section 2.4.1. This is done over an encrypted connection, ensuring that details cannot be intercepted.
When the connection reaches the cloud, it can be viewed in plaintext, and altered (as described later). The cloud then opens a link to the desired web application and forwards on the request. We attempt to upgrade the connection to HTTPS by connection to the web application on port 443 using SSL. If the connection is accepted, we upgrade to HTTPS, otherwise we forward the request via HTTP. One disadvantage is that the user does not know this is a secure connection — they are communicating in plaintext with the daemon service (after which the encryption begins). The plaintext connection between the browser and daemon application is not important, because this occurs within the device.

3.1.3 Hijacking HTTPS

Hijacking HTTPS traffic is more complicated. HTTPS is designed explicitly to protect against man in the middle attacks. First, we outline how a HTTPS connection is usually made through a proxy:

1. **User → Proxy** CONNECT http://henryirish.com:443 HTTP/1.1
2. **Proxy → User** HTTP/1.1 200 Connection established

At this point, a normal proxy would establish a connection with the requested URL, before forwarding packets between the two parties byte-for-byte. The user and web-application would then negotiate their SSL connection over this channel, ensuring the proxy cannot intercept data.

To combat this we create a custom certificate authority for each SIM, installing in the phone at the same time as the daemon application. Instead of acting as a proxy, we create two separate connections. simultaneously acting as web-server and client.

The first connection is established with the URL communicated by the user in step 1 above. We establish an SSL connection with the web application, acting as a client — this is “normal” behaviour and therefore no special implementation is required. This supplies the response in plaintext to the cloud, alongside the web-server's real certificate (which can be validated).

At the same time, we emulate a web-server to the user. The cloud pretends a connection has been established (by sending step 2 of the above protocol). However, we the user actually then establishes their SSL connection with our application, using a new certificate signed with our custom CA. This new certificate is populated with the details from the real certificate, ensuring it passes the client’s checks. This is essentially the same mechanism by which lawful intercepts are done on HTTPS (although CAs issue governments with trusted certificates).

Requests and responses can now be forwarded between the two connections. However, while messages are encrypted outside the system, within the cloud and daemon application they are alterable plaintext. Note this is not an attack on HTTPS — the user must trust our system’s CA.
3.1.4 Threading for scalability

Now that the connection has been hijacked, we create a proxy server in the daemon application which listens for connections on a socket, spawning a pair of listeners when a connection is received (one listens to the client while the other listens to the web application).

A simple Python outline of the proxy server is shown below. We accept connections on a server socket, determining whether the connection is HTTPS or HTTP by reading the request format from the initial line off of the socket. The paired listeners are then launched, as described below.

```python
def run(self):
    # Accept connection.
    client_socket, client_address = self.proxy_socket.accept()
    # Take some data from the connection and pull out the host and port.
    ...snip...
    if headers.headers['Request']['method'] == "CONNECT":
        # HTTPS connection.
        ...snip...
    else:
        # HTTP connection.
        ...snip...

    # Setup listeners
    ...snip...
    client_listener.start()
    server_listener.start()
```

A pair of listeners is spawned for each connection, ensuring robustness and reasonable scalability with many users. Both the Java implemented daemon application and the Python implemented server application depend on analogous listener threads (the Python interface is shown below), which are subclassed by appropriate client_listener and server_listener threads.

```python
class Listener(threading.Thread):
    def run(self):
        ...snip...
        while not self.stop:
            self.send(data)
            ...snip...
            self.paired_listener.stop = True

    def send(self, data):
        self.alter(data)
        self.output_socket.sendall(message.reform())

    def alter(self, message):
        ...snip...
        ...snip...
```

- 23 of 54 -
These are almost entirely stateless (only the client id is stored) threads, ensuring the order of requests and responses does not affect the application. They listen on connections and forward data through to paired socket, allowing the data to be altered via the `alter(self, message)` call first.

### 3.1.5 Daemon application

The daemon application’s (shown in figure 5) main role is to hijack HTTP/S, as detailed above, along with routing connections to the SIM. The daemon application’s functionality is relatively limited, because it is installed on the OS and is therefore not part of the trusted computing base.

![Figure 5 — The daemon application.](image)

In an outbound request, the daemon alters the connection in one way — it appends an encrypted version of the user identifier to the request headers, as detailed later in section 3.3.2. This is loaded from the SIM. It then passes the request securely to the cloud application.

When a response is returned by the cloud application, the daemon receives it from the secure socket. It analyses the headers in the response, extracting those that it “knows” about. These have predefined behaviours (e.g. a `SE-decrypt-username` header triggers a SIM decryption request). The daemon evaluates these, potentially altering or dropping the response (or even generating new requests, as detailed in section 3.5.1), before passing it to the browser.
3.2 Secure element

Although this project focuses on using the mobile device’s SIM as the secure element, we design the daemon application so that other potential secure elements (as described in section 1.7.2) can be used instead. Ideally (in a finished product) the user will be able to select the type that they have. To achieve this, we define an API for secure elements, as described in section 3.2.1.

In this project, we provide a secure element implementation for the SIM, alongside an implementation written in Java for Android (although this provides no real security, it is fine as a stand-in for testing — the reasons for this are detailed in section 3.2.3).

3.2.1 API

We minimise the API for secure elements for two reasons. Firstly, large APIs can be vulnerable to attack, since there are the many more ways they can be used which the designer did not envision. As an API grows, it is difficult to manage the increasing complexity. Secondly, smaller APIs are simpler to implement, potentially requiring less powerful secure elements and ensuring that it is easy to design future secure elements which are compatible. A diverse and well supported range of SEs will improve the system’s uptake and may offer increased flexibility to the user.

The secure element API is implemented in Java as an interface:

```java
public interface SecureElement {
    public String authenticationWithCloud();
    public String encryptForSE(String plainText) throws SecureElementException;
    public String encryptForCloud(String plainText) throws SecureElementException;
    public String decryptForCloud(String cipherText) throws SecureElementException;
}
```

Each user has a unique ID, which is stored within the SE. The method `authenticationWithCloud()` is used to authenticate with the cloud, providing the user ID alongside a secure token (which ensures that fake SE’s cannot be used to impersonate other users). This is returned encrypted with the cloud’s public key, a nonce and a MAC so that it cannot be tampered with en route or replayed.

The next two methods, `encryptForSE(String plainText)` and `encryptForCloud(String plainText)` are simple methods. As expected, they encrypt the plain text string using either the public key for the SE or the cloud’s public key. The first could be used to enter data into the cloud (although this feature is not yet implemented within the system), while the second is used for sending secure communications to the cloud.
The final method decryptForCloud(String cipherText) is the most important method, and is used to decrypt the user-data that is stored within the cloud. This method takes the data encrypted with the SE’s public key, decrypts it (using a private key stored within the SE), before re-encrypting it with the cloud's public key as described in the protocol in section 2.4.2.

3.2.2 Coding for the SIM

The SIM is a Javacard, and therefore coding for it as relatively simple. This project used Gemalto development and simulation tools as an IDE. We specifically develop a SimToolKit (STK) applet, which extends from the javacard.framework.Applet class. The standards for the STK are somewhat complicated, and are enumerated in 3GPP TS 43.019 and 3GPP TS 51.014. However, very little of these are actually required (much of the standards concerns SMS and data downloads).

Essentially, all communication with the SIM comes via APDU (application protocol data unit) commands, and these are handled by the Javacard framework. Methods on our STK application are called on receipt of these APDUs. The STK application must register with the Javacard for certain events at initialisation. These events then triggered the application via the framework, which select the applet. A handler method in the applet accepts the event and either accepts or returns data, before again going inactive.

Javacard provides implementations of the Java cryptographic libraries used in the Android secure element implementation in javacard.security and javacardx.crypto. This means that coding this type of application for the SIM is actually relatively easy, with no external library requirements.

It is also worth noting that because SIM applets are firewalled from each other, we do not have to consider the possibility of other applets interfering with this one.

3.2.3 Limitations

Hypothetically, it is possible for SIMs to initiate connections over the air, and therefore to download data. Therefore the SIM could technically stand in for the cloud application. However, the SIM is a limited device and we do not do this, as explained in section 1.7.1.

The Samsung implementation of the RIL (radio interface layer, used to communicate between the OS and SIM) is slightly non standard and Android support for the STK has been removed in recent official versions. Although it has been forward-ported in community builds, including CyanogenMod, SDK support is gone and therefore coding for the STK on Android is very difficult. Due to project time constraints, we tested the SIM implementation in a Gemalto simulator, with a software implementation of the API being used as a stand-in in the daemon. Due to the small size of the API, we can be confident the tests are valid.
3.3 Communication between components

We now define how the components actually communicate with each other. Communication between the cloud application and the SIM occurs via the daemon. Communication between the cloud and daemon is used as a lower level transport for communication between the cloud and SIM.

3.3.2 Communication between the cloud and daemon

As mentioned earlier in section 2.3.1, this communication has to piggy-back upon user requests, because an alternative would require either the cloud to be able to instantly instantiate a socket with the device (impossible), or a constant connection to be present between the device and cloud (an impractical requirement for mobile devices for obvious reasons).

Since the application is designed to work with HTTP, we use HTTP headers for communication between the two applications. We essentially extend the HTTP header protocol [37] to include new headers (e.g. SE-authentication and SE-decrypt-username). The daemon service extracts these headers before triggering some action (e.g. calling the SE decryption method on some ciphertext).

Both the daemon and cloud therefore have to be able to parse the HTTP headers and alter them, either extracting and removing headers (when receiving a communication), or adding new ones (when sending a communication). This is taken care of two similar “Headers” classes, which convert HTTP headers into alterable hash maps, before reforming them.

3.3.1 Communication between the cloud and SIM

Much of the high level protocol details between the SIM and the cloud are discussed earlier in section 2.4. In this section we instead focus on the implementation of these protocols.

As mentioned, we use the and PKCS#1 standard of RSA, with OAEP (padding). Rather than using a custom implementation (which would likely be flawed and difficult), we use the PyCrypto library [38] and native Java implementations (specifically the rsa/ECB/OAEPWithSHA1AndMGF1Padding implementation of the Cipher class). These rely on SSL keys generated by OpenSSL [13]. Because RSA can only encrypt blocks of a certain size, we use CBC chaining.

RSA encryption produces bytes which look random, and this is inappropriate for direct inclusion in a HTTP header. This could be a problem if a ciphertext includes new line characters which break the HTTP header standard. Therefore ciphertexts are base64 encoded before transmission.
3.4 Bootstrapping the system

To succeed, this system must be easy for average users to setup. The simplest bootstrapping model (and the one implemented) is to assume the SIM keys are preloaded at manufacture, and automatically stored in the cloud. Other users may desire more control however, and so we must consider how to bootstrap new SIM keys for registration in a custom cloud service.

Bootstrapping involves first generating a new SIM RSA key pair. The public key must then be uploaded to the new cloud application, while the cloud application's public key must be sent to the user. It is impossible to bootstrap on an insecure device without a secure path — messages can be MITMed before reaching the SIM. Fortunately the SIM is removable and could be bootstrapped on another device, with the assumption that both devices are unlikely to contain the same malware. Future SEs with secure path allow the user to verify the key exchange at both the cloud and SE.

3.5 Automated login and registration

The automated login system allows the user to authenticate with sites without entering credentials on their device. By training the user not to enter information, we ensure they are never phished.

The overall mechanism for an automated login is shown in figure 6. The major problems to be solved in this are detecting the login page and sending the login request, which are covered later in this section, alongside decryption which is described more in section 2.4.

3.5.1 Detecting login pages

Detecting a login page simply involves finding the appropriate form element within a website, if it exists. This is impossible to with 100% accuracy, because there are no set standards for login pages. Nonetheless we can use heuristics to make good guesses.

The simplest way of doing login page detection would be to hardcode the URLs for the internet’s most popular services, and then detecting them as they are accessed. Although this is very basic, it could still be very useful — just 2 companies — Facebook and Google account for over 20% of an average user’s overall internet usage [36]. A slightly more complex extension of this (using regular expressions to detect pages rather than URLs) would potentially increase robustness, particularly for login forms embedded other sites (e.g. OpenID authentication).

More complex methods can take advantage of the fact that most login forms are relatively uniform, with a user identifier — this is typically a username or an email address — alongside a password. A typical example, taken from Facebook.com, is shown below:
Web forms are often designed to be readable by screen readers (devices which make standard HTML webpages accessible to the blind), and therefore follow simple, readable conventions (e.g. the human-readable name attributes on each tag in the above code). We use the Python BeautifulSoup module to create a model of each HTML page, and extract all `<form>` elements from it, taking all those forms which have similar characteristics to the above (e.g. a username field and a password field). There may be multiple forms on the same page, and therefore we use the action of the form to detect which is the login form.

![Figure 7 — User verifying decryption.](image)

Once the login form has been extracted, we load the relevant (encrypted) authentication data from the database. This means that the identity of the website needs to be extracted.

Applications can be stored on domains (e.g. example.com), subdomains (e.g. app.example.com), or subdirectories (e.g. example.com/app or example.com/apps/app). Multiple applications can also be stored on the same server. Each authentication is identified by the domain (including subdomains) of the requesting app. Subdomains are removed until authentication data is found.
Finally, we decrypt the data at the SE. This requires user confirmation, as shown in figure 7. Currently this is done in the daemon application, although future SEs could utilise a secure path for greater authenticity.

3.5.2 Making login requests

Rather than having the application put the decrypted authentication data into forms and the user submitting them (which would be insecure, exposing data to the OS), we automatically submit the form. We do this by analysing the login form to create an app-specific HTTP request for logging in.
It is possible to extract information about this request from the \texttt{<form>} tag. The URL to which the request must be made is stored in the \texttt{action} attribute, while the type of request is stored in the \texttt{method} attribute — \texttt{GET} methods have form information stored in the URL, while \texttt{POST} methods have form information stored in their own HTTP headers.

### 3.5.3 Registration

Registration is similar to authentication, although with subtle differences. Registration forms are less uniform between sites, but we can take advantage of the fact that pages are again formatted with screen readers in mind and make use of readable \texttt{name} attributes. Common elements include elements to “confirm” emails or passwords, and we can use these to detect registration forms. We also analyse the \texttt{action} attribute — registrations often submit to pages like \texttt{registration.php}.

We load encrypted user data (in the event that some specific part of required user data is unavailable, we abort the registration attempt) and send it to the SE for decryption. The data must then be appropriate re-encrypted by the secure element before the cloud fills in the form. Unfortunately it was not possible to fully implement registration given the project’s time constraints.

![Figure 8 — A CAPTCHA form](image)

Another subtlety relates to CAPTCHA forms, as shown in figure 8. These are elements designed to prevent automated registrations by spam bots. Hypothetically the CAPTCHA image could be passed to the user while (or perhaps instead of) they enter the secure element PIN.

Finally, a triggering mechanism must be designed (to prevent the application registering to every site visited!). This could be implemented into the user-verifies-decryption stage.

### 3.6 Cookie interception

Cookies are pieces of state information sent in HTTP requests (and setup in HTTP responses by the server) with multiple uses. Sessions are based on cookies, and are tokens allowing prolonged authentication of a user after they originally enter their credentials. As mentioned in section 1.2.2, these can be vulnerable and we therefore only allow the client browser to see ciphertext versions.
The protocol for this exchange is described in section 2.4.3, but essentially each cookie is encrypted with a nonce and MAC, and passed to the browser. When the browser sends a request, the nonce is “used up” and the cookie must be re-encrypted with a new nonce and re-sent to the browser along with the server response. Cookies are therefore stored, encrypted with the secure element’s public key, in a database indexed by user id. The cookie’s expiry date is also stored, ensuring that the database purges cookies at the same time as the browser. A HTTP request with cookies is shown:

1. $U \rightarrow S$ GET $X$, Cookie: $\{\{C\}_s, T, MAC_c\}_s$
2. $S \rightarrow C$ GET $X$, Cookie: $\{C, T+1, MAC_c\}_c$

The cloud application validates the nonce against a list of valid nonces per user id.

When a response is sent by the server, there are two stages of evaluation in the cloud application:

**Deconstruction:** The set-cookie headers are removed from the response, encrypted with the appropriate SE’s public key, and added to the database. If cookies are set to a blank string they are assumed deleted — if a cookie under that name is stored in the database, it is deleted.

**Reconstruction:** The response header is then reconstructed. All cookies corresponding to the user ID are loaded from the database and encrypted with new nonces. They are then added to the response as set-cookie headers, along with the appropriate stored meta data.

By preventing the browser from ever seeing the decrypted cookies, we prevent the cookies from being altered on the client side (e.g. in javascript). Fortunately, this is rarely done by applications and therefore will not effect the majority of user’s browsing experience. It could be mitigated in some cases (i.e. where information is added to a cookie, or the cookie is deleted) by detecting these changes on the encrypted cookie and making analogous changes to the original.

It should also be noted that cookies can be used for attacks like XSS [21]. It is therefore possible (although unimplemented) for the cloud to sanitise these malicious cookies during reconstruction.

### 3.6.1 Continuing sessions

One major advantage of the implementation detailed above is that it is now simple to allow sessions to be continued between devices. When a user transfers their SE between devices, all web applications remain logged in, and their user data is “transferred”.

When the SIM is inserted into a differing device, new HTTP requests will not contain any of the expected cookie data. Fortunately, this data is stored in the database and therefore is restored to the new browser via the first server response. Unfortunately, cookies are not inserted into the initial
request made by the new browser. We could send back a cloud-generated response with the text “transferring SE, please refresh” on detecting this first request, although this is not implemented.

This approach has one major unintended advantage, along with a minor disadvantage. It allows applications on a phone to share the same session, something previously impossible. For example, if a user has two browser, then the sessions from one will be automatically transferred to the other when they switch. This is because it is impossible to distinguish between a device switch and an application switch from the cloud. Unfortunately, the “stickiness” of the cookies means that they are impossible to clear from the browser, and can only be altered by the cloud or original web server. A simple work around for this might be integrating a “clear cookies” button into the cloud application.

3.7 Blocklists

3.7.1 Malicious sites

There are many approaches which could be used to detect malicious websites. The desktop solution is to use antivirus software to verify each site visited, although these are not available on mobile. Another option is to use feeds to check whether other organisations block the site. A commercial example of this is Netcraft's anti-phishing feed [34], which allows URLs to be tested against an API.

This topic could clearly be taken much further, with heuristics and features being fed into classifiers to determine whether a site was bad. This is beyond the scope of this project. Future integration with the author's Part II project on automated phishing classification [26] might prove fruitful.

3.7.2 Certificate validation

Certificate validation occurs in the cloud application and has two levels. The application maintains a list of trusted root certificates and certificate revocation lists, while another layer allows the user to personalise these. Certificate revocation lists are published for each CA. For example, Verisign's CRLs are published at http://www.verisign.com/repository/crl.html.

CRLs are essentially signed lists of certificate serial numbers, each serial number pertaining to those certificates which have been revoked. Testing involves checking certificate serial numbers against an updated CRL.
To do actual certificate validation, we leverage the Python library Twisted [27], as shown in the code above (adapted from [30]). certificate_authorities and crls are pre-generated with the combined certificates and CRLs. We tell the SSL context to (i.e. to check the server certificate), and to fail if either the certificate subject name does not match the site's hostname, or if the certificate has been revoked.

3.7.3 Adverts

The cloud application’s advert filtering relies on publicly available blacklists. For example, Adblock Plus maintains an “easylist”, along with foreign language specific variants. These are maintained by an active community, and are therefore up to date. Although this project only implements the blocking of advertising URLs, easylist also maintains a list of bad elements within a page. Since the entire HTML content of each page is intercepted by the cloud application, it is possible to strip these from the page, although this would be costly.

Primitive attempts have been made to block adverts without blacklists [31]. These use machine learning to infer page structure from HTML and then classify these elements. Future machine learning methods may be effective in producing smaller or quicker advert blockers. Performance could also be improved by parallelising the blacklist using map reduce, or potentially by converting easylist to use a trie-based approach.
3.7.4 Preventing tracking

The naïve implementation for preventing tracking uses the do not track header:

```
GET /thirdpartycontent.html HTTP/1.1
Host: thirdparty.example.com
DNT: 1
```

This informs servers that the user does not want to be tracked. Unfortunately, it requires the server to honour this — the vast majority do not.

This system therefore creates a mechanism whereby each tracker has a profile which encapsulates the cookies they use. It is then trivial to block based on these. In a real-world implementation, community involvement would ensure the best performance (both in coverage and reactivity).

Many browsers allow for security policies which block cookies from sites which have a differing hostname to that originally requested. This prevents embedded content from setting unwanted cookies. Unfortunately, since each site is accessed via a separate connection through the proxy, implementing this within the cloud application would require large amounts of state. HTTP referral headers can also be used by web applications to track users. These are stripped by the cloud.
Chapter 4

Evaluation

4.1 Testing

We first test whether the system implemented performs as designed. This is done by testing each feature, first upon a custom site (built so that it can test each element), and then upon real world sites. We evaluate in section 4.2 whether the threat models described in the design are contained.

4.1.1 Custom sites

Most testing was done on a simple custom written site. This was designed using modern tools and authentication mechanism, with the aim of emulating the core functionality most current web applications but without the inherent complexity of testing upon them. It is shown in figure 9.

The site contained a simple login form. This was linked to a SQLite database, where inputted authentication details (via POST) were checked. On success, a session was started with the user, allowing access to a secondary, secure page, containing a single advert. The user is then able to logout, deleting their session. There is also an option for registration, which creates a new user/password combination in the database. This site was implemented in Python, using the Flask web application framework. The site is also available over HTTPS, using a self-signed certificate.

This site is used to test the basic functionality of each component of the system. This was done via a number of unit tests, described below.
HTTP hijacking: We first test the core functionality of the system. We configure an Android phone with the appropriate daemon application, configuring the proxy to route connections through `localhost:8080`. All additional functionality is turned off in the cloud application. Requests are then made to http://authapp.henryirish.com (the location of the sample website). Finally we check the source and the response text against that received when the connection is not routed through the proxy. This resulted in no difference, a success. A trace of the connection was also recorded through the cloud, and is displayed in Appendix B.

HTTPS hijacking: We test whether HTTPS hijacking works by again turning off all features on the cloud application. Rather than testing on the mobile browser, we configure a desktop to browser to connect through the Android phone's daemon, since this allows more information about a response to be extracted. The appropriate custom root CA is installed with the browser. The request (to https://authapp.henryirish.com) was again compared to a normal request for validity; this was successful.

Certificate validation: This was an extension of the previous test, where certification validation is turned on in the cloud application. Because the certificate being used to sign the custom application is self-signed, validation fails and the connection ends, as expected.

Authentication: The SQLite database used in the custom site was set to contain known credentials. The browser loaded the site's login page, and was presented with the second image in figure 7. When tested in a desktop browser, a session cookie had been created.

Cookie interception: This test followed on from the above. The session cookies created by the server are only 54 characters long. The cookie received by the client however was different (and longer, implying encryption) than that originally set by the server, implying it was correctly hidden. When a new request was made to the server, the server received the appropriate cookie, implying that the cloud correctly decrypts it.
Blacklisting: This was tested in three stages. Firstly, http://authapp.henryirish.com was added to the blacklist, which the system then blocked. Secondly, a profile was created stating that cookies called session from henryirish.com were used for tracking. The system subsequently stopped forwarding this cookie (breaking the application!). Lastly, the page displayed one of several ads from the NYT website. These were all successfully blocked.

As evidenced above, the system correctly implements all functionality, as described.

4.1.2 Real world sites

Obviously several of the previous aspects should be tested on real world websites. Hijacking HTTP/HTTPS, cookie validation, blacklisting and cookie interception do not depend on the content of the requested pages, and therefore results will not change when different pages are requested. These therefore will not benefit from further testing on other sites.

Authentication — Linode: Linode is the company hosting the custom application. They were chosen because they had a simple and clean form, allowing authentication to be tested without having to worry about strange side effects caused by Javascript. The user was able to authenticate successfully using the application.

Authentication — Facebook: Unfortunately, the cloud application was unable to login to Facebook. This was due to what was likely Facebook’s CSRF prevention mechanism — a hidden token is added to every form to prevent users unintentionally “submitting” the form from another page. Every Facebook form contains <input type="hidden" ... name="m_ts" value="1370009857" />. This could be fixed by automatically detecting and loading hidden form elements into the cloud application as they came through.

Advertisements — New York Times: The result of this test is shown in figure 10. The cloud application is successfully able to remove prevent the advert image on the New York Times website from being loaded.
4.2 Threat models revisited

During the design, we defined 4 major threat models (in section 2.1). In this section we evaluate to what extent each threat model is countered by the implemented system.

4.2.1 Privacy conscious user

The first threat model ignored threats to the device and focussed upon the link between the device and the web server. It was concerned with ISPs, governments or WiFi snoopers being able to intercept (and potentially steal) information.

This project counters the threat of ISPs and WiFi snoopers entirely. Connections between the user and the cloud are entirely encrypted. All data from the device is routed through the cloud, ensuring it is impossible to tell which web applications a user is using. Because the cloud application services many different users, we gain an element of anonymity (although this is not guaranteed), providing protection against more powerful opponents capable of sniffing traffic upstream of the cloud application. By automatically upgrading connections to HTTPS, we also help to prevent the upstream data being intercepted.

Unfortunately, governments can use similar hijacking methods to those described to lawfully intercept encrypted communications upstream of the cloud application. This is impossible to detect or prevent. Using the cloud application as a threshold mix could provide a degree of anonymity to these connections however (or, for stronger anonymity, tor). Lawful interception between the cloud and device could be prevented by removing all CAs except the cloud’s own.

4.2.2 Malware infested device

The second threat model describes a device containing malware, but where the malware does not have root access. Attacks focussed on phishing account details and making malicious requests.

The authentication mechanism should prevent phishing attacks on the device, although considerable user training will be required to teach users not to enter their details anywhere on the device. If this barrier is overcome then we can guarantee no data is ever stolen as no data is ever present on the device OS. Discerning malicious requests automatically generated by malware from requests by a human is possible by confirming in the daemon before each decryption.

4.2.3 Fully compromised device

This threat model is similar to 4.2.2, although allows malware to have root access to the device OS. Arbitrary code can therefore be rewritten and therefore the OS is therefore no longer trustworthy. This allows for attacks on the daemon.
We are now no longer able to trust the daemon application, and therefore cannot trust the confirmation dialogues it creates. Without hardware support it is impossible to validate whether a request originated from the user, although if a SE with trusted path became available then this could be used. We also cannot necessarily trust data coming in and out of the daemon, however since it is made inaccessible to the OS (via encryption), this issue is solved. We also prevent replay attacks and ensure messages are authentic.

4.2.4 Attack on cloud service

The final threat model was concerned with a direct attack on the cloud application used in this design. Since this is a centralised service, we assume the cloud service administrator is both trusted and competent. This reduces the chances of an attack, particularly when compared to the competencies of the average user this service is targeted at.

It is almost impossible to defend against a long term attack — the attacker may inject code and leave. This is an inherent risk users take when they connect to every service on the internet.

The project is instead designed to counter short term attacks, particularly database dumps (which are one of the most frequently attacked aspects of an application). The data is encrypted in such a way that it is impossible to access data without also collecting the appropriate secure elements from users (an impossible task).

4.3 Benchmarks

This project was not engineered with performance in mind, but as a proof of concept. Nonetheless, it is useful to benchmark the different elements in the connection to find an upper bound on the performance cost of this application per request.

<table>
<thead>
<tr>
<th></th>
<th>Custom App</th>
<th>Facebook</th>
<th>Linode</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Request</td>
<td>336</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>HTTPS Request</td>
<td>778</td>
<td>1012</td>
<td>803</td>
</tr>
<tr>
<td>Authentication</td>
<td>1302</td>
<td>n/a</td>
<td>1409</td>
</tr>
</tbody>
</table>

Table 1 — Comparison of times for different types of request and to different web services (in ms).
Measure from request exiting to response entering the daemon.

Table 1 therefore compares the cost of several features against several websites. None of these costs are excessively high although as expected, HTTPS requests are more expensive. Authentication is the slowest of all, but because it is relatively rare this is of less importance. These benchmarks are
calculated on a Samsung Galaxy S2 phone, with the cloud application running on a 512MB Linode VPS (hosted in London). They are run through WiFi internet to minimise ISP based delays. Each measure was repeated 10 times. The control measure was not run through the application at all.

<table>
<thead>
<tr>
<th></th>
<th>Outgoing Request</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Device</td>
<td>Cloud</td>
<td>Server</td>
<td>Cloud</td>
<td>Device</td>
<td>Transit</td>
</tr>
<tr>
<td>Control WiFi</td>
<td>35</td>
<td>0</td>
<td>138</td>
<td>0</td>
<td>1846</td>
<td>106</td>
</tr>
<tr>
<td>System WiFi</td>
<td>69</td>
<td>172</td>
<td>137</td>
<td>201</td>
<td>1959</td>
<td>112</td>
</tr>
<tr>
<td>Control 3G</td>
<td>40</td>
<td>0</td>
<td>140</td>
<td>0</td>
<td>1850</td>
<td>2336</td>
</tr>
<tr>
<td>System 3G</td>
<td>68</td>
<td>165</td>
<td>138</td>
<td>197</td>
<td>1972</td>
<td>2350</td>
</tr>
</tbody>
</table>

Table 2 — Amount of time a request/response to sample site spends at each stage (in ms). Transmit measures total time to send message between stages. In-device includes rendering times.

Table 2 shows a breakdown of the different performance costs within the application, as compared to those incurred a normal request. Requests over WiFi and the slower 3G are compared. As is evident in figure 11, the large of cost in a mobile web request of fetching the page over WiFi (transit), and rendering the page (incoming response — device) dwarfs all other costs. The total cost of routing the connection through the cloud (and doing processing there) over routing it directly to the destination is ~350ms in comparison to a total cost of ~2500ms on WiFi and ~4700ms over 3G. This only amounts to between a 7-15% increase in page load time when using the system.

![Figure 11](image-url) — Visualising the connection times (in ms) for the system vs control.

We can therefore conclude that the added performance cost of this system is very small for mobile browser users using WiFi. This cost is pushed towards being complete negligible when compared with the time taken to request data over mobile networks like 3G.
4.4 Limitations

Perhaps one of the biggest limitations is caused by the Android platform. On Android, applications have to make special efforts to use the OS-configured proxy settings. This is in contrast to iOS, where application network connections are automatically routed through them. This means that applications will not automatically be hijacked. However, most major applications and, most importantly, browsers (e.g. Android browser, Chrome) do support this, minimising the limitation.

As described in section 3.2.3, the recent lack of support for the STK in Android, alongside the nonstandard RIL implementation used by Samsung has made actually connecting the SIM application to the daemon application infeasible. Nonetheless, it is entirely feasible. The project was tested with an identical Java implementation running within Android.

Lastly, when Android is fully compromised there is currently no way of telling whether a request made to the daemon application was truly from the user or whether it is from malware. This is impossible without a trusted path between the secure element and the user (an insecure path might be altered by the corrupt OS), although this may become available in future SEs.

4.5 Advantages

Although the project was created to meet the goals outlined in chapters 1 and 2, there were a number of unintended advantages created by the design.

In its default state, the application requires very little configuration before it is usable. The daemon application must be downloaded and installed from an app store, and requires only the host address of the cloud application. Finally, the phone is configured with the daemon proxy by visited a URL and automatically downloading a proxy configuration file.

By routing many user’s connections through this system, we have a centralised point of security. Because each user has delegated their security to a (hopefully) more competent administrator, it is less likely that they will be compromised.

4.6 Disadvantages

Most disadvantages to this system are caused by the centralised aspect of the cloud application. As a centralised point of security, the system also acts as a central point of failure. Although an attacker is perhaps less likely to penetrate the system (defences can be “concentrated”), the number of attacks and the consequences of a successful attack are far greater. This is mitigated because the system does not store data in an accessible manner, although this will not help in an extended...
breach. There are also robustness considerations. In the event of a bug, service is disrupted for all users, potentially preventing internet connectivity. Using multiple proxy services ensures that a disruption will effect fewer users.

The system uses encryption between the user and cloud, preventing interception by the user’s ISP. This is effectively a shift in trust from the ISP to the system administrator. However, it is already common knowledge that ISPs sell browsing data [17], and therefore users are shifting their trust from an entity that they already actively do not trust. For those still unhappy, they are able to run their own cloud server.

Lastly, processing and bandwidth use is also centralised. When these security features are implemented for desktop browsers, each user does their own processing. Bandwidth becomes a problem as mobile users consume more video and audio content. This issue could be mitigated by using heuristics in the daemon to determine which connections to secure, as well as using an appropriate business model.
Chapter 5

Conclusion

This project has been very successful and has allowed the creation of an additional security layer around Android mobile devices. This prevents some of the most common attacks by current malware, while also providing useful privacy and security features to protect users from external threats. As shown in the evaluation in chapter 4, the system prevents phishing attacks, user tracking and session hijacking alongside additional convenience features like adblocking, and automated mobile password authentication. Finally, by using the user's SIM as a secure element to key the system, we ensure that the user maintains control of their data, and that data is transportable between devices.

There key areas of innovation within this project were:

The SIM keying a semi-trusted cloud application: Although the SIM has been used as a secure element for small applications (e.g. payment or ticketing), it has never been used to key for a larger, more complex and functional applications. This project combines the power and flexibility of a server application with the security of trusted SIM applications.

Utilising traditional attack methodologies for defence: MITM attacks on HTTP and HTTPS connections are typically used maliciously, either against the user or against web applications (as is the case with applications like Burp Proxy). This project instead used such aggressive techniques to implement defensive security features for the user.
Cross application security/privacy: This project implements a set of security features, currently unavailable for mobile browsers, although relied on in desktops. These features are application agnostic and do not rely upon a trusted OS, making them resilient to malware.

Making a secure mobile setup simple: All of these features are packaged in a way that is extremely simple to setup for the average users. Although some of these features are available for desktop, they typically require separate configuration and are therefore mainly used by those who are technologically literate. This solution requires minimal setup to use its features.

Although the system implemented is a research project and therefore not yet ready for the real world, with additional work it could be deployed in a number of scenarios. For the average user it is perhaps most useful if supplied by their mobile phone network. Phones could be bundled with both the daemon and SIM application, and therefore a user could be set up with zero configuration. Alternatively, this could be setup as a third party service (for those who do not trust their network providers), or power users who want more control can even run their own cloud application.

5.1 Future directions

There are many ways via which this system could be improved. This project has focussed upon how incoming web application responses can be tuned, changed and blocked to increase user security. There are many other gains to be made by analysing outgoing connections. Some connections do not need high security (e.g. video feeds or entertainment). Being able to develop heuristics to automatically detect those applications and not route them through the cloud security layer would increase performance for both the user and the cloud application. At the same time, analysing outgoing connections could allow malware to be both detected and prevent it from calling home (in many cases removing its threat).

Another obvious area of exploration is in moving (a version of) the cloud application to future secure elements. Obviously this is not currently feasible given the limited power and bandwidth of the SIM, but doing this on future secure element hardware as it is released will mitigate some of the concerns users might have with trusting another third party application. Equally, moving the daemon application into a trusted execution environment would prevent it from being attacked and allow many current cloud features to be moved to it without risk.
Bibliography


7. Percoco, Nicholas, and Sean Schulte. “This is REALLY not the droid you're looking for...” DEFCON 19 (2011).


10. Gilbertson Scott. “HTTPS is more secure, so why isn't the web using it?” May 2011.


Appendix A

Project Proposal

A.1 Introduction and Description of Work

The overall aim of the project is to use the SIM of a mobile device as a trusted element to secure the mobile internet, and specifically to address the vulnerabilities described below. This includes SSL certificate verification and password management.

Malware is prevalent on mobile devices, with F-Secure detecting 51,447 different Android samples in the wild, and 42 new variants found in the last quarter alone\(^1\). Due to the complexity of smartphone operating systems it is impossible to fully secure them. Many root exploits exist, both legitimate (Cyanogen Mod for Android and Redsn0w for iOS) and illegitimate\(^2\), ensuring that the root operating system cannot be trusted either. Malware has many different threat models, although this project will focus on those which attack the user directly, by stealing personal information. This can be done in a variety of ways; root-kits can steal saved application data, or can trick the user into entering personal information. Another big threat surface involves malware intercepting internet communications. Here they can steal information as it is transmitted, potentially sniffing cookies (allowing for session hijacking) or taking account details entered by users when logging into sensitive web applications.

---

1 F-secure Mobile Global Threat Report, Q3 2012

These issues can potentially be solved by moving some of the work to a trusted execution environment (a TEE). Although there has been work on developing a TEE for the Google wallet, this was rejected by carriers who desired control over the trusted path. There is also currently work being done by several companies3, though these are still in development and are not yet available. The SIM can also be used a carrier controlled secure element, and several applications (including VISA's mobile payment applications) have already been developed for it. The SIM is capable of communicating directly with the internet via the phone's baseband. By routing all traffic through the SIM (again, via the baseband), security critical tasks like certificate validation can be done there, before the data reaches the insecure OS. The system might also prevent session hijacking by analysing site security policies and invisibly substituting secure cookies before they are passed to the OS.

Vulnerabilities when browsing are not limited to information being stolen. The certificate validation mechanism used by HTTPS (for encrypted browsing) might be attacked — either changing the certificate verification code, or altering the certificates known to the browser. This might result in the user being directed to sites with bad certificates and phished.

Although certificate validation theoretically works, the real world implementation gives a broken security model for economic reasons. Most governments want their own certificate authority (CA), greatly increasing the attack surface. The business practices of the big root CAs are also cause for great scrutiny — companies are not held accountable and their profits are directly related to the number of certificates that they issue. One CA was caught issuing a no-questions asked certificate to someone claiming (and lying) to be from Mozilla4, and the same authority was hacked in 2011 allowing domains to be hijacked5.

Several fixes have been proposed to the certificate security model. The EFF proposes Sovereign Keys6, which removes the dependance on third parties, while providing the same encryption and domain validation guarantees that are currently available. Another potential solution is Convergence7, which provides similar guarantees. A potential extension to this project would involve implementing one of these solutions (after research to determine which is better).


4 The Register, CA issues no-questions asked Mozilla cert. Dec 2008.


7 convergence.io
A severe issue in security usability, unrelated to malware, is in password authentication. Users typically do not pick strong passwords and mobile device input mechanisms exacerbate this (making special characters difficult to access). Users who do have strong password security models, with unique passwords per web service, run the risk of forgetting details. Many of these problems have been solved on the desktop by use of password managers. These are applications which manage the creation and storage of passwords, automatically inserting them into pages when required. Applications like this exist for Android, but do not allow complete integration into the browser. By routing all connections through my application on SIM (and therefore, through the password manager), login details can be filled in painlessly.

A.2 Starting Point

This project is receiving limited support from Samsung who fund my supervisor, Laurent Simon, and funded me to do a UROP over the summer. They have supplied a Samsung Galaxy SII, and may be able to provide development tools (e.g. an SDK) for SIM programming. If Samsung are unable to provide any SDK, then the open source JavaCard framework will be used.

Although I have experience writing Android applications, I have never attempted programming for the SIM, and therefore “learning time” will need to be factored in. No similar projects exist, and I am not even aware of similar applications (e.g. proxy password managers/implementations of SovereignKeys) existing for Android.

A.3 Substance and Structure of the Project

The project can be broken down into several components. There will be a SIM card application and several Android applications for interfacing with that application.

**Proxy Browser:** To intercept connections from the users browser, a SOCKS proxy will be used. An application will listen on some port, and reroute the phone's connections so that they run through the SIM card. The phone will register with the proxy using the OS implementation found in the settings. A potential extension of this project might allow the user to define what types of connections they deem secure (which are routed through the SIM), and those they don't mind about the security of (which do not touch the SIM).

**SIM Application:** The core of the project will be the SIM application. This will take connections from the Android proxy application and route them directly to the internet. It will also perform certificate validation and may attempt to automatically fill in the user’s login details.

**Password & Certificate Manager UI:** There must be a mechanism for adding passwords to the SIM's password manager, as well as altering them and enabling/disabling them. The SIM
must also be loaded with acceptable certificates. This Android application will provide a UI front end to this.

**Further Certificate Validation:** As a potential extension, one of the improved certificate validation algorithms described earlier could be implemented. They would have to be evaluated to see which would be better for the project (e.g. time/space/implementation complexity, the availability of reference implementations and documentation).

As a potential extension, one of the improved certificate validation algorithms described earlier could be implemented. They would have to be evaluated to see which would be better for the project (e.g. time/space/implementation complexity, the availability of reference implementations and documentation).

It should be noted that these implementations will suffer from a number of limitations and will therefore be proof of concepts, rather than user-ready. One issue is the speed of the link between the SIM and OS, which utilises a bidirectional serial link. This will limit the performance when downloading pages of many hundreds of KB. SIM cards also have limited storage — typically 64KB or 128KB. Certificates are around 1KB each, and this means that only a limited number of certificates can be stored, with the analogous problem for the password manager. Next generation SIM cards are purported to have faster links and larger storage, and this will make them more practical.

The application which runs the proxy is running on the Android OS and is therefore vulnerable to attacks by malware, which could theoretically ensure that connections are never routed through the SIM (and therefore receive no extra security). A solution might be to remove the OS’s ability to connect to the internet without going through the SIM. This is a prototype and therefore such complex OS modifications are unnecessary, however.

**A.4 Evaluation**

The project will be considered a success if a prototype application is created for SIM which allows for the exploration of the implications of using the SIM as a secure element. The major problems, as discussed, are that of secure connections to websites and investigation into whether having a new, carrier controlled trusted path via the SIM is feasible over current TPM style secure elements (which are not carrier controlled).

**A.5 Timetable**

*Week 1 (10th-16th Dec)*

Exploration of SIM card SDKs. If application is not feasible, then I want to fail fast.
Weeks 2-3 (17th-30th Dec)
Development of Android proxy application for capturing and rerouting connections, and can be initially tested by routing connections to, for example, a computer. This will allow development of the core SIM application.

Weeks 4-6 (31st Dec-22nd Jan)
A basic SIM application for receiving the connections from the proxy application, and routing them to the internet, and vice-versa. One week spent on holiday.

Weeks 7-9 (23rd Jan-12th Feb)
Development of a certificate validator, given some hardcoded certificates.

Weeks 10-11 (13th-26th Feb)
Code for inputting (hardcoded) username/password pairs into pages of specific web domains.

Week 12 (27th Feb-4th Mar)
Mockup of all UIs for Android applications.

Weeks 13-15 (5th-25th Mar)
Development of Certificate manager, Password manager and Proxy manager Android applications. API will be developed on SIM for adding new passwords/certificates.

Weeks 16-17 (26th Mar-8th Apr)
Catchup time.

Weeks 18-21 (9th-29th Apr)
Evaluation and implementation of novel certificate validation algorithms.

Weeks 22-23 (30th Apr-13th May)
Further extension work: secure/insecure traffic selector in proxy manager.

Weeks 24-28 (14th May-14th Jun)
Write up.
Appendix B

Sample Trace

Below is the trace of a connection between a user to the site discussed in section 4.1.1

```
Listening on  <socket._socketobject object at 0xb72dae2c>
Loading certificate
Loading certificate
Chainmail ID: 11

GET http://authapp.henryirish.com/ HTTP/1.1
Host: authapp.henryirish.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:24.0) Gecko/
20130531 Firefox/24.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: identity
Connection: keep-alive

Starting ClientListener
Starting ServerListener
Loaded ID: 11

HTTP/1.1 200 OK
Server: nginx/1.2.1
Date: Fri, 07 Jun 2013 11:54:56 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 174
Connection: keep-alive

Not Logged In<br />
<form action='/' method='GET'></form>
```

- 54 of 54 -