GuarDroid: A Trusted Path for Password Entry

Tianhao Tong and David Evans
University of Virginia
[tt4cc, evans]@virginia.edu

***Abstract*—Sensitive online transactions are now frequently executed using smartphone clients. Whereas users of personal computers execute these transactions in a browser, smartphone users tend to use installed apps. These apps use username and password pairs as the primary authentication method and may come from untrusted parties, opening users up to attacks that steal user's passwords. We present GuarDroid, a system that protects user's password from untrusted apps. The key idea is to prevent apps from seeing passwords directly and establishing a trusted path between the user and the service that leverages the smartphone operating system as a trusted computing base. Our system does not require any modifications to existing apps or services, while still providing users with high assurances that they are not providing sensitive passwords to a rogue app.**

I. Introduction

Users are increasingly using applications on smartphones to conduct sensitive transactions, such as banking, social networking, and private messaging. Smartphones differ from traditional personal computers in that their limited interfaces and small application model lead users to install customized applications (usually called apps) to conduct these transactions rather than using a generic browser. This exposes users to new risks from fraudulent applications that attempt to collect user credentials. The goal of our work is to establish a trusted path between the user, through the trusted core of the mobile device, and onto the intended trusted server in order to protect users from inadvertently providing passwords to fraudulent apps. Our solution does not require any modification to current apps or application servers.

An example of the kind of fraudulent app we are targeting is the phishing app known as FakeToken [19]. Many banks today employ two-factor authentication to protect users. In order to use two-factor authentication, users install a token generation app from their bank on the smartphone and use this app to get a one-time token that is used as part of the login process. This FakeToken app pretends to be the official token generation app by presenting a very similar UI to the user and requesting the bank login ID and password. The tricked user enters her account ID and password to this app for the token, which the app sends to a server controlled by the attacker. With this information, the attacker can conduct arbitrary transactions on these accounts. Another similar example is Android.Fakeneflic [2] which looks similar to the legitimate Android Netflix app and requests the user's ID and password upon login to send to the attacker's server.

On traditional computers, users usually execute secure transactions in a web browser where they can examine the destination of sensitive data using security indicators. These trust chains are missing on smartphones apps. Figures 1 and 2 show the difference between the traditional personal computer environment and a smartphone. When the user tries to log into Twitter through a traditional PC browser (Figure 1), she can look at the certificate of the remote server and verify the identity of the destination; with in-app login on a smartphone (Figure 2), a user is not able to verify the destination of sensitive data and must trust the app won't just send the password anywhere.

The problem here is that in the application model, the app gets control of the whole screen (a feature often needed by games and e-book readers). Any naive approach to designate a UI component as a trust indicator (analogy to the lock symbol used by the browser) could be mimicked by a malicious app by drawing its own UI component that looks exactly like the trusted UI component.

Felt and Wagner [8] studied the risk of phishing on mobile devices and raised the concern about the very problem we are trying to deal with in our paper: the lack of trusted path from the user to the service. They suggested a possible solution to this problem that would use a small portion of the screen as a trusted display for identity indication. However, this solution would undermine the user experience, especially in apps like games and video players where users want full screen display. Our work seeks to enable such a trusted path without requiring any dedicated screen real estate or hardware modification to existing devices.

A. Approach

Our idea is to isolate the passwords from untrusted third party apps by leveraging the operating system of the smartphone as a trusted computing base and establishing a trusted path directly from the user to the app server. The user will input her password in a trusted input provided by the OS, and then the password will be encrypted by a secret key stored in a file protected by system permission. The encrypted password will be sent to the third party app which does not have access to the real password. When the third party app tries to log in to the application server, our system will intercept the network traffic and spot the encrypted password, which it replaces with the decrypted password after getting the destination confirmation from the user. This hides passwords from untrusted apps while preserving external behavior.

Our approach relies on users learning to only provide their passwords to the trusted input method. Fraudulent apps

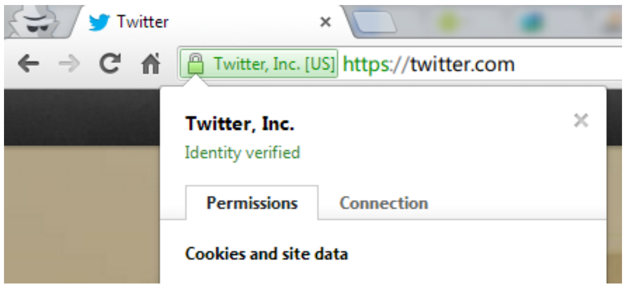
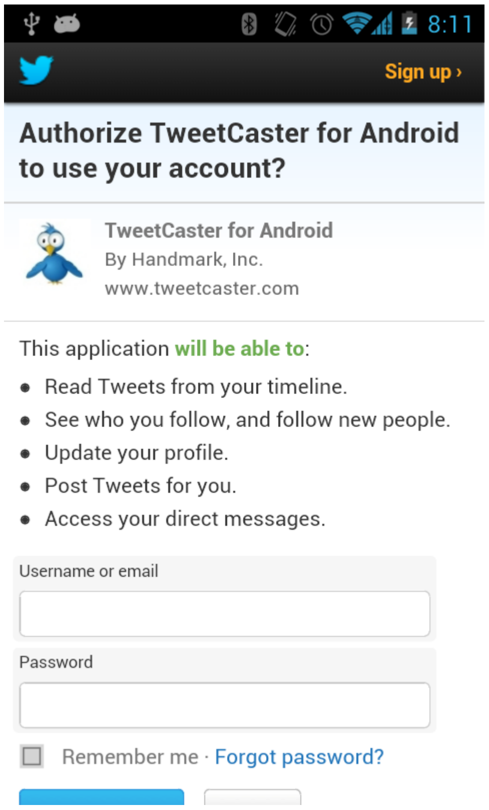


Fig. 1. User can verify certificates on a desktop browser.



Authorize TweetCaster for Android to use your account?

TweetCaster for Android
By Handmark, Inc.
www.tweetcaster.com

This application will be able to:

- Read Tweets from your timeline.
- See who you follow, and follow new people.
- Update your profile.
- Post Tweets for you.
- Access your direct messages.

Username or email

Password

Remember me · Forgot password?

Fig. 2. User has no way to verify destination using app login.

may attempt to trick users into entering their password by popping up an input box with similar user interface as the system one. Since we do not have a dedicated display to indicate trusted input, the user has no way to distinguish whether a password input window is from the system or from a fraudulent application. Therefore, we need to establish a trusted path between the user and the smartphone OS.

We establish a trusted path by using a shared secret between the user and trusted base. To set up the shared secret, we modify the boot process so the user can select a secret phrase early in the boot process, before any untrusted app has any chance to execute. The secret phrase is stored securely by the operating system and displayed to the user to signal secure input. The app never sees the entered password, which is only transmitted to trusted definitions confirmed by the user.

To summarize, the key idea of our system is to hide the user passwords from untrusted apps. This involves two main challenges, and they are the main focus of this paper:

1) Establish a trusted path from the user to the smartphone OS so the user can distinguish input boxes in which a password can be safely entered.
2) Enable the untrusted app to send the user's password to a legitimate application server without exposing the password to the app itself. This is done by modifying the outgoing network data and providing trusted destination confirmation based on TLS/SSL certificates, thus enabling an end-to-end trusted path between the user and service.

B. Threat Model

We focus on the goal of protecting the user's sensitive input from being stolen by an untrusted third party app on smartphones.

Trusted Base. We trust the operating system of the smartphone but third party apps may be malicious. This conforms with the trust model in mainstream smartphone OSes. The operating system should provide file permission protection that protects files reads and writes between different accounts, especially system and user-level apps. We assume the malicious app cannot gain the control over the operating system or break the file permission protection provided by the operating system. Although this might not always be true (e.g. [1, 20]), it does not happen very often and is not in the scope of this work. We assume the system kernel, all system services, system libraries, and system apps built into the system image can be trusted.

User Behavior Assumption. We also assume that the user makes correct security decisions based on the interfaces presented to them by GuarDroid. There are two places where we depend on users to make prudent decisions. One is the GuarDroid Safe Input (explained in Section II-C) which requires that users only enter passwords into input boxes display their secret phrases. The other one is the vigilance of users checking the destination verification dialog (explained in Section II-E). Section IV reports on a preliminary experiment on the first part of this assumption (GuarDroid Safe Input).

Model Limitation. We limit the sensitive input GuarDroid protects to password in our implementation since password is the most common way of authentication we use on smartphone apps and is clear to users to know when to expect our protection. It can be easily inferred by checking if an input box is of password-input type. We do not consider the impersonation problem of the malicious app after login in this work. Protecting user's password is valuable. Many real world attacks [2, 19] only try to collect users' passwords and abuse them later in an unexpected time and way. It is hard to track back to these apps in question. With the passwords protected, the only way to execute the malicious transaction is on the device after the user logged into the account normally. The ability of attack gets limited and this makes the malicious app easier to be discovered because of the correlation of

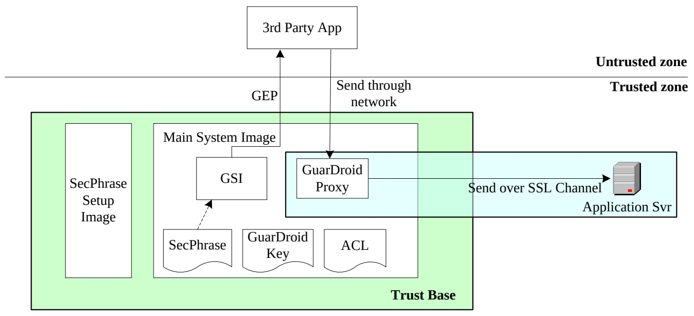


Fig. 3. The division of GuarDroid components in trusted and untrusted zone

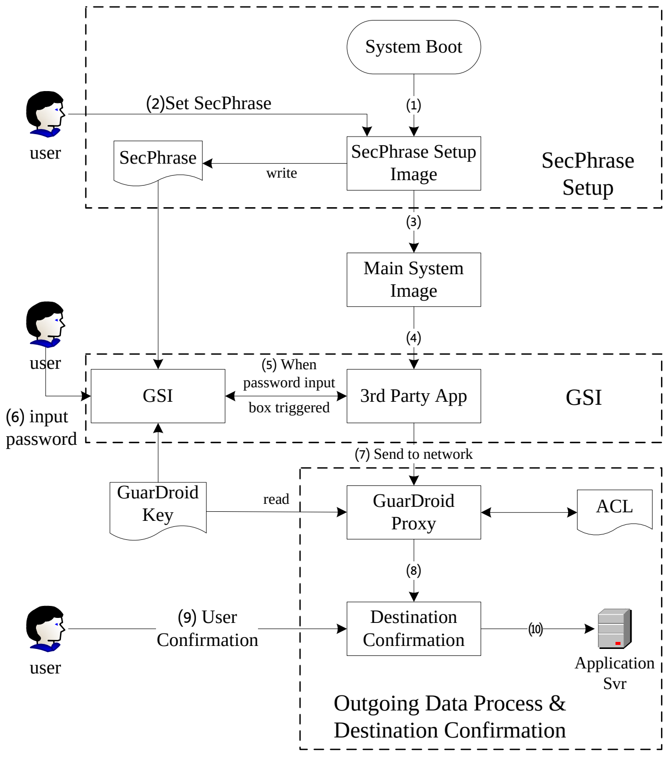


Fig. 4. The life cycle of an GuarDroid protected event

the timing. As soon as the malicious app is discovered and removed from markets and devices, the app users do not need to worry about changing their passwords since there will be no further abuse. It is still possible for a malicious app to just do the login normally and impersonate the user after this app gets privileges. GuarDroid does not prevent this directly, however, it can be prevented using our system if all sensitive fields (for example, destination bank account and amount in bank transfers) are properly marked as sensitive input by the application server. We do not try to solve the problem of inferring what piece of data is sensitive information.

Next, we explain the details of how to implement this approach in Android in Section II. Section III evaluates the compatibility and performance overhead of our system. Section IV describes a preliminary feasibility study on human users. We provide some background and recent research related to this project in Section V.

II. Implementation

We implement our prototype on Android 2.3 and tested it on a Samsung Google Nexus S phone[1]. Android gives every app a unique UID (unless two apps from the same author agree to share UIDs with each other) and the processes of these apps run with this UID. Permission management and isolation is thus based on the underlying Linux separation between different users. We develop GuarDroid based on a customized Android system called CyanogenMod7 [4] to leverage some auxiliary tools it provides. The source code of entire GuarDroid system is available at https://github.com/ursatong/GuardroidROM/.

A. Overview

GuarDroid involves four stages depicted in Figure 4: setting up a shared secret phrase between the user and trusted OS (SecPhrase Setup), GuarDroid Safe Input (GSI), Outgoing Data Processing, and Destination Confirmation.

SecPhrase Setup (Section II-B). When the phone boots up, it first boots into a separate system image (Step 1 in Figure 4) to ask the user to input a secret phrase (SecPhrase) as a shared secret between the user and the system (Step 2) before any third party code could run. The SecPhrase is stored in an OS-protected file. All trusted GuarDroid system UI components display this SecPhrase to indicate to the user that she is communicating with a trusted component.

GuarDroid Safe Input (Section II-C). After establishing the SecPhrase, the phone boots the main GuarDroid system (Step 3). Whenever the user would enter a password using a system default password input box (Step 5), our GuarDroid Safe Input (GSI) is automatically invoked. Since GSI is a system app, it has access to the SecPhrase and displays it to the user. Other input methods will fail to mimic the GSI's UI because they cannot obtain the SecPhrase. Users should only enter passwords into the GSI dialog (Step 6), which is distinguished by displaying the SecPhrase. GSI, after receiving the password, encrypts it with a secret key stored and protected by the system permission. We call the resulting encrypted password the *GuarDroid Encrypted Password* or *GEP*. For benign apps, this should make no difference since they should not perform any operation based on the clear text of password other than storing and sending it to the legitimate server. However, if a malicious app wants to steal the password and send it to some other server, that server will only receive a meaningless string and the malicious app has no way to learn the actual password. While GSI is running, the system disables access to sensors for any background processes, mitigating the risk of side-channel attacks such as those using the accelerometer to infer the password entry [3, 30].

Outgoing Data Process (Section II-D). After this, when the

[1]The phone was released in December 2010, with a 1GHz Cortex-A8 CPU, PowerVR SGX540 GPU, 512MB RAM, and 16GB Storage.

user submits her input (e.g. submits a login page), the app will typically send the user's login data over the network to the application server. We assume the password will be transferred through TLS/SSL, although it is possible to add support for other protocols. We set up a global local proxy on the device so that all network traffic will be redirected to this proxy (Step 7). This local SSL proxy acts like an in-the-middle attack to intercept raw data in the SSL connection. The proxy examines the network traffic, searches for patterns that match a GEP and decrypts then to replace the encrypted password with the plaintext password in the outgoing traffic.

Destination Confirmation (Section II-E). We need to ensure the decrypted password is only ever sent to the correct application server. To do this, the GuarDroid Proxy then asks the user for destination confirmation based on the destination server and app package name (Step 8). If the access is allowed (Step 9), the GEP will be replaced by decrypted password and then encrypted using the SSL protocol. It then will be sent out over the network to the confirmed destination (Step 10).

Figure 3 illustrates the division between untrusted and trusted components in our system. According to our threat model (Section I-B), we don't trust the third party app. The trusted base includes the local system components that cannot be modified and protected files not readable by non-system users. The trusted path is extended to the the application server by the SSL Channel managed by GuarDroid Proxy.

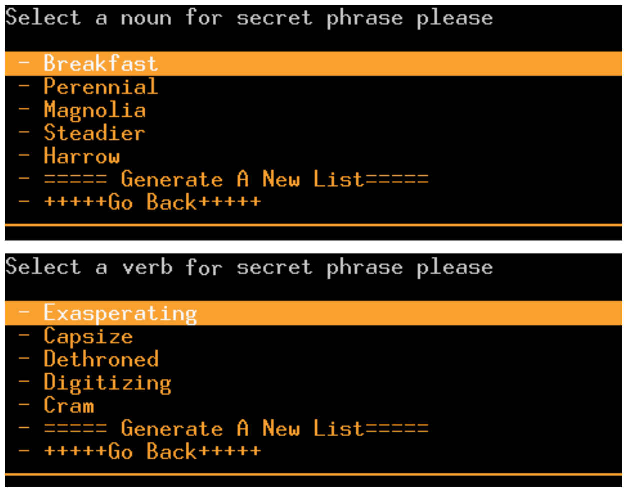
Next we will explain each stage in detail.

B. SecPhrase Setup

We want to set up the SecPhrase as early as possible in the boot process before any third party app is executed. However, this stage cannot be too early during the boot process since we need the display and input drivers to support user interaction with the system. Since Android is based on Linux, the user-interface libraries and framework reside in user space. It would be very difficult to control the system to load these libraries and then suspend all other processes to safely execute the SecPhrase setup.

Instead, we leverage the Android recovery image. The recovery image is a separate partition that Android can boot into to perform system maintenance routines for the phone like installing system updates and factory reset [10]. This image contains basic I/O drivers to enable limited user interaction and is read-only to any non-root process in the main system. This image also does not allow execution of any outside code - you cannot install any third party program in the recovery image. This makes it an ideal place to implement the SecPhrase Setup.

We modified the code of the recovery image to let the user select a SecPhrase from a list of randomly generated words. We call this modified recovery image the SecPhrase-Setup image. The interface is shown in Figure 5. On our implementation platform, only three hard-keys are supported under recovery image: volume up, volume down, and power representing up, down, and confirm respectively. The touch screen will not work here. This is the reason why we let users choose from a list of words instead of letting them input their



```
Select a noun for secret phrase please

- Breakfast
- Perennial
- Magnolia
- Steadier
- Harrow
- ===== Generate A New List=====
- +++++Go Back+++++
```

```
Select a verb for secret phrase please

- Exasperating
- Capsize
- Dethroned
- Digitizing
- Cram
- ===== Generate A New List=====
- +++++Go Back+++++
```

Fig. 5. SecPhrase selection under SecPhrase-Setup image

own. The list is generated randomly from a dictionary file with about 10k words. One noun and one verb list are generated. The user picks one from each of these. We hypothesize that this produce a good balance between sufficient entropy to make it unlikely that an app could guess and user memorability. Of course increasing the number of words could lower the possibility of the SecPhrase being guessed by the malicious app but it might also become hard to remember for the user. The user can request a new batch of random words if she likes none of the displayed ones. After the selection is made, the SecPhrase will be echoed back for confirmation. Note that a malicious app cannot mimic the SecPhrase Setup process because we use power button as selection confirmation. In main GuarDroid system, the power button will cause the phone to sleep or shutdown. This behavior cannot be overridden by any third party app.

The SecPhrase will be stored in Trusted Storage, essentially a file with root access only. All other system protected files like GuarDroid Encryption Key, GEP-IV mapping table, and Access Control List are also stored in the same way.

At early stage of GuarDroid booting process, the system will look for the SecPhrase. If it cannot find it, it will reboot the system into the SecPhrase-Setup image. Since the check of the presence of the file does not require any user interaction, it can happen in very early stage of boot process as soon as file system is ready so no third party app can interrupt. We modified init.rc, a script written in Android Init Language [9] that guides the set up of Android system and executes the check as soon as the system image get mounted. If the check passes, the system continues to boot the main system image for user use.

The SecPhrase is cleared every time the system shuts down. Therefore, if the user forgets her SecPhrase or wants to reset it, she simply needs to reboot her phone.

C. GuarDroid Safe Input

GuarDroid Safe Input (GSI, shown in Figure 6) is a trusted customized input method built into the system image that accepts the password input from users and completes the trusted path. It will automatically pop up when the user would enter a password using the standard password input box, i.e., when typing in EditText with inputType = textPassword in Android. The user enters her password to GSI, which encrypts the password and returns the GuarDroid Encrypted Password (GEP) to the app.

Our design leverages the SecPhrase we set up in the previous section and displays it on the UI of GSI. The user should only enter her password when the SecPhrase is presented. In order to remind the user to check the SecPhrase, our design requires the user to click the SecPhrase before they can enter anything. Section IV reports on a preliminary experiment on the design.

To effectively identify and restore GEP in network traffic, GEP will be constructed as: G{*Enc(Password, Key)*} where G{...} is a pattern we used to fast locate the possible GEP in data stream. In this formula Enc(d, k) means to encrypt data d with key k. In our implementation, we use AES/CFB8/NoPadding first and then encode the result with Base64. AES/CFB8/NoPadding is a stream encryption using 8 bits as a block so there is no overhead of extra letters. Encoded with Base64, the output is all printable ASCII characters. This is necessary because in many password input implementation, unprintable binary bytes might produce errors. The *Key* is a 16-byte key managed by the system generated randomly when the system is first set. A 16-byte IV required by our chosen algorithm is randomly generated at the time of encryption. A table of mapping from Enc(Password, Key) to IV is stored in the Trusted Storage. In this way, 1/3 overhead over the original length of password from Base64 and a 3-byte identifier G{...} are added to the password, which we assume is acceptable in most apps (Section III-A reports more on the compatibility of our design).

Our keyboard is a simple keyboard implementation based on the SoftKeyboard example from the Android SDK. Since it will only be turned on when entering a password by the system automatically and restored to the previous input method after finishing the input, this simple input method should suffice because it will not impact the user experience elsewhere. Word prediction and auto-correct are undesirable features when entering the password.

D. Outgoing Data Processing

After receiving the password, the app will usually send the password to the corresponding application server. It may save a local copy, encrypted by its own algorithm or not, but in the end, it needs to send the password to the application server to authenticate the user to the service. Since the password received by the app is encrypted by our GuarDroid system, if it's sent to the application server, it will be meaningless.

The way we deal with this problem is to install a local proxy that decrypts GEPs. All outgoing data is redirected to this proxy and processed before being sent to the network. The redirection is done by using iptables commands.

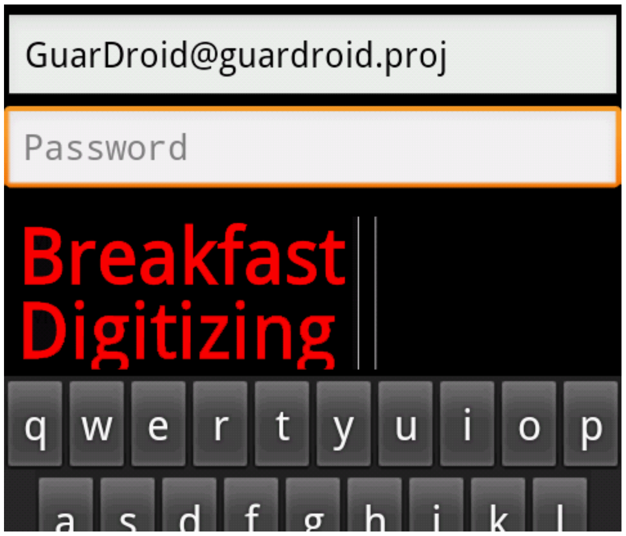
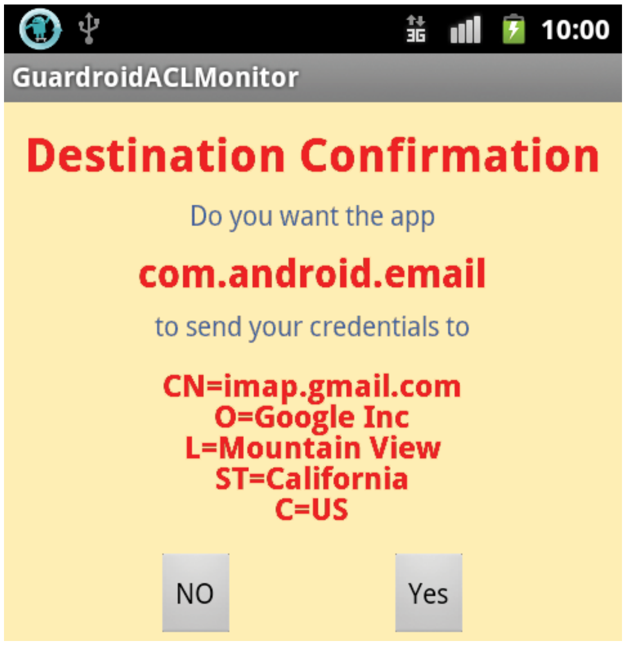


Fig. 6. GuarDroid Safe Input

The proxy looks for patterns that match G{...} to identify the possible GEPs. Then it looks at the GEP-IV table in the Trusted Storage to get the appropriate IV and decrypt the intercepted GEP with the GuarDroid Key. The GEP-IV table is maintained as a self-balanced binary search tree using GEP as sorting key for fast index to lower the overhead. For false positive, the look up will usually fail and the packet will be sent out untouched although there is still very small chance that a string in other network traffic happens to match a GEP. The rate can be lowered by increasing the length of GEP. After these, the proxy will replace it in a protocol aware manner (meta-data also needs to be updated per protocol like HTTPS, SMTP, etc).

Since we are dealing with user passwords, we need to support TLS/SSL. By default, when using TLS/SSL, the secure channel is established between the app directly to the application server. Therefore, our proxy can only see encrypted data and will be unable to analyze and decrypt the GEP. In order to analyze such TLS/SSL connections, our local proxy needs to act as an SSL proxy which generates site certificates on the fly. As shown in Figure 3, since the GuarDroid Proxy is a system app and its root CA used to sign those generated certificates is also built into the system as part of the trusted base, a user-level app will not be able to access the memory of the GuarDroid Proxy, nor can it forge the certificates signed by the GuarDroid Proxy's root CA. The SSL secure channel is now between this proxy and the application server. Since this local proxy is part of the trusted base in our GuarDroid system, this completes the trusted path from the human user to the application server, through the proxy.

We only consider data transmission over TLS/SSL. This does not mean that our GuarDroid system cannot work with apps that transmit data directly over plain TCP. This can be achieved by applying the same procedure on plain TCP data and even easier because we do not need to use the SSL proxy technique for plain TCP data. But GuarDroid in this case



Destination Confirmation

Do you want the app

com.android.email

to send your credentials to

CN=imap.gmail.com
O=Google Inc
L=Mountain View
ST=California
C=US

Fig. 7. UI of Destination Confirmation

will have some limitation. One important limitation is that we will not have trustworthy destination confirmation with TCP because of the nature of TCP that plain TCP data can be easily spoofed or intercepted over the network.

We implement this local proxy, GuarDroid Proxy, based on Sandrop project [26] and Webscarab project [22] and extend it so that it can find and decrypt the GEP before sending data to the server.

E. Destination Confirmation

After handling the outgoing data, we also need to ensure that the password goes to the right place. Otherwise, a malicious app can just send the password to their own server and get the clear text. Since the SSL secure channel is between the application server and the GuarDroid Proxy, the GuarDroid Proxy examines the certificate chain of the destination server, shows the user the details of the destination server's certificate, and lets the user decide if the data should be sent to this server. See Figure 7 for an example.

This process is required only for outgoing network traffic containing a GEP. If the root certificate is not trusted, a serious warning will be shown to the user. To avoid frequent user confirmation, a set of ACL files are stored in Trusted Storage that cache these decisions. It is unnecessary for the Destination Confirmation and ACL Manager dialogue to display SecPhrase because writing to ACL files needs system privilege and it is useless for a malicious party to just mimic the UI of these system components.

One might wondering since we have destination confirmation, we know where the passwords go, then why we need all the rest complicated parts of GuarDroid. The reason for that is with GuarDroid's protection on password, we will only need destination confirmation when the password needs to be decrypted and transmitted over the network. It is okay to have other network traffics (like Ads, statistical report to app developer, and other third party library network traffic) without confirmation as long as it does not ask for the password. On the other hand, if we do not have the support of GuarDroid, we need to taint tracking the password which is very complicated with all the possible transforms a malicious app can apply to the clear text of password. As a result, we would need a destination confirmation for every destination which is confusing and annoying.

III. Evaluation

We evaluated the compatibility of our system with existing apps and its the performance overhead.

A. Compatibility

Although we preserve the external behavior of the app so the application server will see no difference, our design assumes the app does not depend on the actual password in any way, other than sending it to the app server. We tested several popular apps to verify this assumption. Google accounts linked to Android phone use a special protocol for login. Our system supports this protocol. We also choose the top 20 free apps from Google Play under categories of "Communication" and "Finance" (accessed on Feb 21, 2013). Table I shows the results. We did not encounter any compatibility problems with the "Communication" apps (left half of the table). Two of them login using the Google account linked to Android OS which does not ask for password input, and six of them do not require login or authenticate phone with phone number plus access code in a following SMS, so although there are no compatibility problems with these apps, our mechanism does not provide any enhanced security for them. The remaining 12 apps do support entry of a password, and GuarDroid provides an end-to-end trusted path for these apps.

For the 20 "Finance" apps, two do not require any password and two ask for an SSN as login credential, which is entered using standard password input but restricting the input to be exactly 9 numbers. We were not able to test one app (PNC Mobile) because it asks for a valid username before letting us input password. GuarDroid provides enhanced security for the remaining 11 apps. We found that 3 of them check the length of password on the client app: Navy Federal (password length must be between 4-8) [21], Wells Fargo (6-14) [29], and PayPal (8-20). Although we think it is appalling for modern online services (especially financial ones) to set a low maximum length limit for passwords, these constraints cause problems for our encrypted passwords since they must satisfy these length constraints. One general solution would be to ensure that the GEP has the same length as the user-entered password. Given the need to include an identifying tag in the GEP, though, this would not be possible. One solution will be maintaining an identifier-password mapping in Trusted Storage and use a matching-length tag in place of the GEP. The security model will be the same however, the reason we choose encryption over this password table in GuarDroid prototype is

TABLE I
App Compatibility Test Result

App Name	Compatibility Notes	App Name	Compatibility Notes
Facebook Messenger	Yes	Chase Mobile	Yes
Skype	Yes	Bank of America	Yes
Yahoo! Mail	Yes	Wells Fargo Mobile	Password must be 6-14 chars
Kik Messenger	Yes	IRS2Go	Authenticate with SSN
GO SMS Pro	Yes	PayPal	Password must be 8-20 chars
WhatsApp Messenger	Authenticate with SMS	MyTaxRefund	Authenticate with SSN
Voxer Walkie-Talkie	Yes	Capital One Mobile	Yes
Chrome	Android Account	Mint	Yes
Antivirus Security	No Login	H & R Block	Yes
Google Voice	Android Account	USAA Mobile	Yes
Firefox Browser	No Login	TurboTax SnapTax	Yes
Viber	Authenticate with SMS	H & R Block 1040EZ	Yes
Handcent SMS	Yes	GEICO App	Yes
Yahoo! Messenger	Yes	Tip N Split Tip Calc	No Login
Dolphin Browser	Yes	U.S. Bank	Yes
Pinger	Yes	PNC Mobile	Not tested
Mr. Number-Block	No Login	Navy Federal Credit	Password must be 4-8 chars
Portable Wi-Fi hotspot	No Login	American Express US	Yes
Hotmail	Yes	Citi Mobile (SM)	Yes
Text Me!	Yes	Discover Mobile	Yes

that with encryption approach, we can still keep third party apps' "remember password" function working without risking storing all clear text of password in the phone.

B. Performance

Our implementation uses a client proxy, which involves significant overhead. However, this is an artifact of our implementation and would not be necessary if GuarDroid were built into the OS. We also evaluate the intrinsic costs associated with scanning traffic for GEPs and rewriting them.

Proxy. We used an app from SpeedTest.net to test the latency and bandwidth to two servers with and without GuarDroid Proxy. DC Server has smaller latency than Chicago Server. We tested 10 times on both servers and the average results are shown in Table II. The bandwidth is only minimally affected by the GuarDroid Proxy. The upload speed is 17.0% slower for DC Server and 21.8% slower for Chicago Server. For both servers, the overhead of latency is consistently around 50ms, essentially all of which is due to the need for a client-side proxy.

GEP decryption. To test the overhead of GEP decryption, we take 5000 strings and encrypt them as GEPs. We also setup a local HTTPS server to minimize the influence of the network. We sent these 5000 strings, one per POST request, to the server. We tested two groups: with the original proxy before our modifications and with the GuarDroid proxy that decrypts every GEP in POST requests. As the result, it takes 280ms on average to send a string over the original proxy to our server while 288ms via GuarDroid Proxy. We can see that the overhead of finding GEP and decrypting is only about 3%.

To answer how frequently a packet that looks like a GEP appears in normal network traffic, we monitored the network traffic on one of the authors computer. Since we only need to look for GEP in outgoing data, we only monitored outgoing links. We recorded 5.5 billion bytes of outgoing data in 3 days

TABLE II
Proxy Latency and Bandwidth Test

	DC Server		Chicago Server	
	No Proxy	Proxy	No Proxy	Proxy
Latency (ms)	13.8	63.2	68.7	131.2
Download Speed (kbps)	6698.5	6732.8	5830.4	6059.3
Upload Speed (kbps)	7294.9	6053.7	7573.2	5923.1

and found 2710 GEP pattern matches. This is a rate of about 4.95×10^{-7}. The GEP pattern uses three bytes and this rate is lower than three random printable ASCIIs ($95^{-3} \approx 1.13 \times 10^{-6}$). When such packets appear, it only cause GuarDroid extra overhead to check if they are real GEP instead of break the network traffic and the rate is low enough that the proxy won't take too much resources.

From the previous evaluation, we can see that the main overhead is the latency of the proxy. The latency may have some impact on apps that establish many small connections and require high interactiveness like Telnet or real-time online games. However, apps insensitive to latency or apps that can mortgage the overhead of establishing connection by using a connection to transmit large amount of data or setup connections concurrently like watching videos online won't feel significant difference. We believe if we fully integrate the versatile third party proxy into the OS to eliminate the extra trip from and to the kernel and put all the proxy in native code, we can further reduce the overhead. Given the fact that the bandwidth is not significantly impacted, the low overhead of GEP detection and decryption, and the rarity of possible GEP packets in network traffic, the performance of our system would not be a big concern.

IV. Feasibility Study

The integrity of our system depends on users making correct security decisions based on the interfaces presented to them by the smartphone. We conducted a small, preliminary human subject feasibility study to evaluate the effectiveness of our design as a proof of concept. The study focus on evaluating if our design prevents users from entering their password into a fake GSI, which can display anything on the screen but not the SecPhrase. We did not test the human behavior assumption behind the destination confirmation.

The result shows challenges in training users to look for SecPhrase before input password and suggests the need for better UI design.

A. Study Setup

We developed a user-level application that mocks the UI of real GuarDroid Safe Input as shown in Figure 6 but does not actually provide any protection for the propose of easier deployment for the user study.

The participants would use their phone as usual. We replaced the default email client on their phone with our own so that our mock GSI will be triggered when they uses email app and the mock GuarDroid Safe Input will also launch fake attacks randomly. These attacks will do no real harm to the participants. However, they will try to trick the participants by displaying a fake password input box with some probability.

We tried to recruit as many participants as possible and make the set up of study as easy as possible. We set up a website, post information on Craigslist, university and city. We installed our test system on the participants' mobile devices and provided them with either a document or in-person demonstration of how the system works. To motivate people join our study, every confirmed registered participants would get $10.00 for just installing our user study apps. To simulate their motivation of protecting their password, each participant will get $1.00 bonus per successful login up to $40.00 but be penalized $5.00 from the bonus for each login that leaks password to the simulated attack. However, due to the resource and time constraints, our user study requires the user to use their own Android phones with version 2.2 or higher, replaces their original email client, possibly one of their most frequently used app, with our modified one which requires password input much more frequently in order to gain enough amount of data in reasonable period of time and they need to use this app for couple of weeks. This is annoying to many people that they may want to uninstall it. Also, for some logistical reasons, we could not accept remote participants and we tried to recruit from outside computer science major in order to minimum the bias. The combination of these requirements of having certain device, prolonged study and local participants not from computer science major only, causes that we ended up with only 4 persons with valid data. All of these 4 participants are not computer science majors and one of them is not from the University. Some of them took the one month study and logged frequently during the study and others took the study for a longer time. The longest one has sent data for 31 weeks.

To simulate the rarity of attacks we lower the probability of the appearance of the simulated attacks and make sure there are no simulated attacks in at least 10 initial logins. Otherwise if the participants see too many attacks, they might be just used to look for attacks which is often neglected in real world.

The mock GuarDroid Safe Input behaves as described in Section II-C. It requires the user to click the SecPhrase before they can proceed to enter the password. The only difference is that there is a FRAUD bottom on the soft keyboard so when users think there is an attack, they can report it by clicking the FRAUD bottom. The mock GSI will then refresh itself so the next input might or might not be an attack.

There are three types of fake attacks:

1) **No-SecPhrase**: Displays neither SecPhrase nor any other error information.
2) **Wrong-SecPhrase**: Shows wrong SecPhrase.
3) **Service-Error**: Shows "SecPhrase Service Error" message.

We recorded how many times each participant attempts to login or provides her password to the attack input which means the intended protection failed.

We do not record any sensitive information including password from the participants. Statistical data from each user is transmitted to our server periodically, but in a way that does not leak any other information about the user's activities. We obtained IRB approval[2] for this user study. More details about the setup of feasibility study can be found at http://GuarDroid.net/index.php/usability-test/how-to-setup-study-environment/.

B. Study Result

Although we did this feasibility user study as a proof of concept, we were not able to perform a larger user study with adequately representative number of participants due to resource and time constrains.

Table III shows the results. We observed 359 finished logins including normal ones and compromised ones. Among them 57 are attacks (19 are No-SecPhrase, 25 are Wrong-SecPhrase, and 13 are Service-Error). 16 of the 57 attacks succeeded. 15 of the 16 succeeded attacks are No-SecPhrase and only one from Service-Error. All of 25 Wrong-SecPhrase attempts are detected. This indicates that when a SecPhrase is shown to the users, they remembered to check its correctness.

Although the preliminary user study implies that users have difficulties in checking for security indicators in our UI design, we believe one reason for this result could be that the behavior of No-SecPhrase just looks like all other normal password input box in current Android system. The participants are so used to the default Android UI with no SecPhrase that they sometimes forget to check or get confused this attack with other login UIs of normal apps they have on their phones. Our user study suggests the need of a better UI and maybe a longer training phase.

V. Related Work

App Source Control. Controlling what apps can be loaded onto the device is one possible solution to the fraud app problem such as a closed app store model (like iOS). This has

[2]UVa IRB-SBS Protocol Number: 2011-0273-00

TABLE III
FEASIBILITY STUDY RESULT

	# Logins	Simulated Attacks	No-SecPhrase		Wrong-SecPhrase		Service-Error	
			Fraud Detected	Total	Fraud Detected	Total	Fraud Detected	Total
Participant 1	142	27	3	12	10	10	5	5
Participant 2	136	15	0	5	7	7	3	3
Participant 3	40	8	0	1	5	5	1	2
Participant 4	41	7	1	1	3	3	3	3

the drawback of restricting what app can do and relying on a central market but even such a closed model cannot always detect malicious applications. For example, N. Seriot shows how malicious applications could pass the mandatory Apple Store review unnoticed and harvest data through officially sanctioned Apple APIs [28] and Marianne Schultz demonstrated how a tethering app camouflaged as a flashlight [27] passed the Apple Store review.

Verifying the author of the apps is another approach. For example, the author of the client app for Bank of America should be a verified Bank of America account and if we trust the verification made by the app market, we trust the app. However, this eliminates the opportunities of benign third party apps that provide more convenience to the user. For instance, users may want to use one application to manage several different bank accounts. Mint [18] and Android Banking by J. Peiffer [23] are two examples which can manage more than half a dozen bank accounts in one app. No single bank could provide this application. Another example is social networking apps which have more third party apps that user may prefer over the official one (take Twitter as an example). The users of these apps today will worry about if their bank accounts passwords will be stolen. With our system, the users can be confident that their passwords will be protected because it's isolated from the apps.

Trusted Path. Several works [13, 15]–[17, 31] attempt to construct trusted paths. In Windows versions with NT kernels, the combination of Ctrl+Alt+Del, known as Secure Attention Sequence, is reserved by the system to establish a trusted path from user to the OS [17]. Similar to this idea, SpoofKiller from M. Jakobsson and H. Siadati [13] built a trusted path between the user and the OS on smartphone by pressing the power button which generates an interrupt to the system. They maintain a white-list of trusted sites. When the power button is pressed, the system checks if the focused web page is on the whitelist. If it is, the password entry proceeds; if not, the app is terminated. Their solution needs the user to remember to press the power button whenever they try to input a password which is very counter-intuitive. Further, maintaining a whitelist of trusted sites is difficult and SpoofKiller only targets the browser phishing on smartphones which is a small fraction of the general problem of smartphone app phishing we target. Although they did claim the same principles apply to smartphone apps, it is much more difficult to infer the destination of data to be sent in app than in browser using their method.

Flicker [15] leverages the TPM on new PC CPUs as a dynamic root of trust to execute sensitive code fully isolated from the untrusted OS and apps. It demonstrates how small the trusted base can be. However, it fails to link this trusted path to the human user. Bumpy [16] extends Flicker by including the keyboard to the trusted base and triggers the trusted execution by pressing reserved keys on the keyboard. To complete the trusted path to the user, a smartphone as a dedicated trusted monitor is used for the user to understand the transition between protected and unprotected input. The same idea applies to other projects that use smartphone as an independent device to verify and authorize the transaction on the computer [14, 24]. Our solution does not require a dedicated device as a trusted monitor.

Origin Attestation. QUIRE by Michael Dietz et al. [5] provides a way to verify the caller of IPC requests in Android. It annotates the whole call chain to authenticate the origin of an IPC request. This can be applied to solve the impersonation problem in our paper by ensuring any input data originates from the system touch screen event and is fresh. However, to provide call chain verification over the network, QUIRE needs RPC attestation by assigning a unique certificate signed by a trusted certification authority and its corresponding key to every phone. The application server should be aware of this and ready to do the check. This requires co-operation from the server and phone manufacture. It is infeasible to modify thousands of application servers and assign such a key to billions of existing phones. GuarDroid can provide a similar level of protection on impersonation with server and phone manufacture co-operation, while still providing useful protection without any modification to existing phones or servers.

Program Analysis. More generally, to protect sensitive data from untrusted apps on smartphones, traditional static and dynamic analysis are applied. For example, PiOS [6] uses static analysis to detect the leakage of private data on iOS. It constructs a call graph from the app's binary and checks for an execution path from a privacy source to a sink such as network. However, this model suffers from limited coverage and cannot accurately solve our problem because if we take user input as privacy source, it ends up in network which seems legitimate and the destination of the network traffic may not be available until run-time. It also depends on a central organization such as official market to do the analysis for them. TaintDroid [7], on the other hand, uses dynamic taint analysis to detect sensitive information leakage. It has four granularity of taint propagation: variable-, method-, message-, and file-levels. A malicious app can apply all sorts of transforms on the sensitive input so variable-level taint propagation is needed. With this, the performance penalty will be high and we might end up with a lot of false positive of leaking information.

Web Phishing. OAuth [11, 12] and OpenID [25] try to solve web phishing problem by providing authentication without exposing login credentials. This won't prevent smartphone app phishing because when the third party tries to access some private information from the service provider, it still needs to redirect the user to log onto the service provider's page to obtain the key and the malicious app still can fake this UI by implementing this in WebView where no security indicator can be found.

VI. Conclusion

GuarDroid is designed to protect a user's password from being stolen by fraudulent smartphone apps without requiring any modification to existing apps or application servers. We believe end to end trusted paths will provide strong security benefits for end users, but establishing such paths also requires cooperation from users. Our small preliminary usability study reveals the challenges in training users to recognize security-critical interfaces, and points to the need for well-designed interfaces and training use habits.

Availability

The modified Android kernel including the GuarDroid safe input mechanisms, as well as our experimental email client, is available under an open source license from http://GuardRoid.net.

Acknowledgments

This work was supported by grants from the National Science Foundation, Google, and a MURI award from the Air Force Office of Scientific Research. The authors thank Alf Weaver for helpful comments on the work.

References

[1] Anonymous. ADB Setuid Exhaustion Attack. http://c-skills.blogspot.com/2008/01/evilness-of-setuidgetuid.html.

[2] Irfan Asrar. Will Your Next TV Manual Ask You to Run a Scan Instead of Adjusting the Antenna? http://www.symantec.com/connect/blogs/will-your-next-tv-manual-ask-you-run-scan-instead-adjusting-antenna, October 2011.

[3] Liang Cai and Hao Chen. On the Practicality of Motion-Based Keystroke Inference Attack. In *Fifth International Conference on Trust and Trustworthy Computing*, 2012.

[4] CyanogenMod. CyanogenMod 7. http://www.cyanogenmod.org/.

[5] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. *USENIX Security Symposium*, 2011.

[6] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *18th Network and Distributed System Security Symposium*, 2011.

[7] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: an Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[8] Adrienne Porter Felt and David Wagner. Phishing on mobile devices. In *Web 2.0 Security and Privacy*, 2011.

[9] Google. Android source tree. https://github.com/android/platform_system_core/blob/master/init/readme.txt, 2012.

[10] Google. Recovery System - Android Developers. http://developer.android.com/reference/android/os/RecoverySystem.html, 2012.

[11] E. Hammer-Lahav. The OAuth 1.0 Protocol. RFC 5849 (Informational), April 2010. Obsoleted by RFC 6749.

[12] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), October 2012.

[13] Markus Jakobsson and Hossein Siadati. SpoofKiller: You Can Teach People How to Pay, but Not How to Pay Attention. In *Workshop on Socio-Technical Aspects in Security and Trust*, 2012.

[14] Mohammad Mannan and P. van Oorschot. Using a Personal Device to Strengthen Password Authentication from an Untrusted Computer. In *11th Financial Cryptography and Data Security*, 2007.

[15] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *ACM European Conference in Computer Systems (EuroSys)*, April 2008.

[16] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. Safe Passage for Passwords and Other Sensitive Data. In *Network and Distributed Systems Security*, February 2009.

[17] Microsoft. How Interactive Logon Works. http://technet.microsoft.com/en-us/library/cc780332(v=ws.10).aspx, January 2009.

[18] Mint.com. Mint - Personal Finance. https://www.mint.com/, 2012.

[19] Ryan Naraine. Remote-controlled Android Malware Stealing Banking Credentials. http://www.zdnet.com/blog/security/remote-controlled-android-malware-stealing-banking-credentials/10804, March 2012.

[20] National Vulnerability Database. Udev Before 1.4.1 Does Not Verify Whether a NETLINK Message Originates from Kernel Space, Which Allows Local Users to Gain Privileges by Sending a NETLINK Message from User Space. CVE-2009-1185, August 2009.

[21] Navy Federal Credit Union. Password Confidentiality. https://www.navyfederal.org/account-management/.

[22] OWASP. OWASP Web Scarab Project. https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project.

[23] Jeff Peiffer. Android Banking. https://market.android.com/details?id=com.jpeiffer.banking.chase, 2011.

[24] T. Pullar-Strecker. NZ Bank Adds Security Online. https://www.rsasecurity.com/products/securid/whitepapers/PHISH_WP_0904.pdf, 2004.

[25] David Recordon and Drummond Reed. OpenID 2.0: a Platform for User-Centric Identity Management. In *Second ACM Workshop on Digital Identity Management*, 2006.

[26] Sandrop. Sandrop - Secure Android Proxy. http://code.google.com/p/sandrop/.

[27] Marianne Schultz. Handy Light: Tethering App Camouflaged as Flashlight. http://appshopper.com/blog/2010/07/20/handy-light-tethering-app-camouflaged-as-flashlight/, 2010.

[28] Nicolas Seriot. iPhone Privacy. In *Black Hat DC*, 2010.

[29] Wells Fargo and Company. Online Banking Enrollment Questions. https://www.wellsfargo.com/help/faqs/enroll_faqs.

[30] Zhi Xu, Kun Bai, and Sencun Zhu. TapLogger: Inferring User Inputs on Smartphone Touchscreens Using On-Board Motion Sensors. In *Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[31] Zishuang (Eileen) Ye, Sean Smith, and Denise Anthony. Trusted Paths for Browsers. *ACM Transactions on Information and System Security*, May 2005.