NativeWrap: Ad Hoc Smartphone Application Creation for End Users

Adwait Nadkarni
NC State University
Raleigh, North Carolina, USA
anadkarni@ncsu.edu

Vasant Tendulkar
NC State University
Raleigh, North Carolina, USA
tendulkar@ncsu.edu

William Enck
NC State University
Raleigh, North Carolina, USA
enck@cs.ncsu.edu

ABSTRACT

Smartphones have become a primary form of computing. As a result, nearly every consumer, company, and organization provides an "app" for the popular smartphone platforms. Many of these apps are little more than a WebView widget that renders downloaded HTML and JavaScript content. In this paper, we argue that separating Web applications into separate OS principals has valuable security and privacy advantages. However, in the current smartphone application ecosystem, many such apps are fraught with privacy concerns. To this end, we propose *NativeWrap* as an alternative model for security and privacy conscious consumers to access Web content. NativeWrap "wraps" the domain for given URL into a native platform app, applying best practices for security configuration. We describe the design of a prototype of NativeWrap for the Android platform and test compatibility on the top 250 Alexa Websites. By using NativeWrap, third-party developers are removed from platform code, and users are placed in control of privacy sensitive operation.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*

Keywords

Web browsers; Mobile applications; Smartphone security

1. INTRODUCTION

Smartphones are now commonplace in much of the developed world, and their popularity continues to rise. A key feature of smartphones is the wide variety of available third-party applications, commonly known as "apps." Users can find apps to enhance nearly any daily activity and provide entertainment during idle periods. Indeed, the official application markets for Android and iOS both contain over 700,000 applications [55, 4].

Privacy is a significant problem for smartphone consumers. In the past several years, a number of research groups have identified widespread privacy concerns with smartphone apps in both Android [18, 32, 19, 26, 25, 51] and iOS [15, 30]. Popular media investigators such as the Wall Street Journal have made similar independent findings [47]. Smartphone apps leak a range of privacy sensitive information, from seemingly innocent phone identifiers to geographic location to entire address books. Researchers often speculate that such data is collected and sold to data brokers that perform analytics for selling advertisements. Regardless of the actual use, it is clear that privacy sensitive data is being leaked by smartphone apps, often without user consent or information.

The current state of the smartphone application ecosystem leaves privacy conscious consumers with a dilemma: either use the app while being aware of the privacy risks, or do not install the app. Many privacy conscious consumers (including the authors) occasionally decide that an application's benefit outweighs its privacy risks. While recent research has proposed fine-grained privacy controls, none are likely to go mainstream. Solutions that modify the OS to allow finer-grained permission control [9, 43, 58], return fake values [6, 58, 32], or limit network connections with sensitive values [18, 32] require significant technical expertise to build and install the custom OS for a specific device. Furthermore, these research prototypes have not undergone rigorous testing, nor are they frequently updated to new OS versions that contain new features and security patches. More recently, an array of solutions have proposed adding inline reference monitors to applications [57, 37, 12, 31] rather than modifying the OS. Unfortunately, statically modifying an application package either results in a painful install process for the user, or requires an online trusted third-party to host modified apps (which to date does not exist). Finally, all of these solutions risk breaking applications in unknown ways, as developers frequently assume permissions are granted if the app is installed.

Privacy conscious consumers sometimes have third choice: use a mobile Website in the phone's Web browser. Many applications are simply a convenient way to access a popular Website from a mobile device. Increasingly, Website owners are developing and maintaining mobile versions of their websites, often with an "`m.`" or "`mobile.`" domain prefix. Frequently, the mobile Website functions very similar to the mobile app. However, there are security and privacy drawbacks to accessing the app through its corresponding mobile Website. First, authentication tokens are stored in the Web browser's cookie store, which has a larger attack surface than if they are stored in an app's private data storage. Second, the shared cookie store allows advertisers and social networking sites to track users [14].

In this paper, we propose *NativeWrap* as a new alternative model for privacy conscious consumers to use Web-based applications on smartphones. NativeWrap balances the security and privacy risks of using the smartphone application and the phone's Web browser.

WiSec'14, July 23–25, 2014, Oxford, UK.

ACM 978-1-4503-2972-9/14/07$15.00.
http://dx.doi.org/10.1145/2627393.2627412.

When a user is visiting a Website in the phone's browser that she would like to run as a native app, she "shares" the URL with NativeWrap. NativeWrap then "wraps" the URL into a native platform app while configuring best-practice security options. In effect, NativeWrap removes the third-party developer from the platform code, placing the user in control.

Specifically, NativeWrap provides the following properties:

- *Isolated Cookie Store*: Web browsers have one cookie store and mediate access based on the same origin policy (SOP). Unfortunately, SOP is insufficient to prevent privacy loss when the same advertisement firm (e.g., DoubleClick) is used on many Websites. SOP also does not prevent large social networking sites (e.g., Facebook) from identifying user browser habits by simply encouraging Website owners to include social networking integration [14]. NativeWrap prevents such privacy loss by ensuring a separate cookie store for each wrapped Website. It also prevents a compromised browser from leaking authentication cookies for multiple Websites.

- *Phishing Prevention*: Phishing attacks are successful when the user clicks on a link and is fooled into entering sensitive information into a fake Website. On smartphones, phishing attacks are aided by Web browsers that remove the address bar to maximize the viewing area [22]. By using a native platform app, the user can be trained to always use the phone's application launcher to access security sensitive services (e.g., banking). NativeWrap provides the native platform app experience to any Website. It also pins the wrapped Website to a specific domain to ensure embedded elements (e.g., ads) do not redirect the user to a malicious site.

- *Correct SSL configuration*: Recent research has identified widespread misconfiguration of SSL in smartphone apps [20, 24]. NativeWrap not only ensures proper SSL verification, but it also can pin the Website to a certificate authority to remove dependence on a large root CA list. Furthermore, NativeWrap adapts HTTPS Everywhere [16] to optionally allow the user to force SSL within the wrapped Website [34].

- *Limited, User-controlled Permissions*: Developers of native mobile applications frequently include extra functionality that impinges on user privacy. NativeWrap defaults to Internet-only permission, with the ability for the user to add several common functional permissions when wrapping the Website.

Our Contribution: The primary contribution of this paper is the proposal of a new conceptual approach for privacy concerned consumers to access Web content from smartphones and mobile devices. We provide a prototype implementation for Android and note that the approach could be adopted by other platforms if it was integrated into the platform OS. We survey 12,500 applications from the Google Play Store to demonstrate the need for NativeWrap. Finally, we test the compatibility of NativeWrap with the Alexa top 250 websites.

The remainder of this paper proceeds as follows. Section 2 motivates NativeWrap. Section 3 describes the NativeWrap design. Section 4 details its implementation. Section 5 evaluates NativeWrap compatibility. Section 6 discusses deployment strategies. Section 7 describes related work. Section 8 concludes.

2. MOTIVATION

Before describing NativeWrap, we must first understand how and why many applications are developed. We begin with a short history of mobile application development while defining several key terms used throughout the paper. We then provide a survey of mobile apps from the Google Play Store to better characterize the significance of the problem.

2.1 Background

The first feature-enhanced mobile phones provided an Internet connection and a Web browser. Early users visited the same Websites as provided for personal computers; however, it quickly became clear that mobile versions of these Websites were required to cater to the small display sizes on mobile phones. These Websites, commonly known as *mobile WebApps* (or simply *WebApps*) are front ends developed specifically to suit the display and user interface aesthetics of mobile phones, and can be accessed by nearly any smartphone with a Web browser.

As mobile phone platforms with native application environments emerged, developers began porting WebApp functionality to the popular platforms. These native applications (or *native apps* for short) are platform-specific, and are hosted on application markets such as the Google Play Store or the Apple App Store, from which users discover, download, and install them to their devices.

Native apps possess the ability to closely interact with the user and use the phone's hardware features such as accelerometers and GPS receivers to provide a rich user experience. As the usefulness of native apps grew, so did their popularity, ultimately leading users to frequently choose a native app over visiting the corresponding WebApp in the phone's Web browser. In turn, more and more companies and organizations felt compelled to provide native app versions of their Websites to stay up-to-date and maintain company image.

Developing and maintaining native apps requires significant resources. First, the application must be developed for each popular platform. Android and iOS use vastly different programming languages and design abstractions. Second, native app updates must occur via the platform's application market, which can include timely review processes (e.g., iOS) or at minimum user annoyance when apps are updated frequently. As a consequence, hybrid applications began to emerge. These hybrid applications are essentially WebApps "wrapped" in a "WebView" class within a native app. Both Android and iOS provide WebView primitives, therefore only a very small amount of code needs to be written for each platform, and updates only need to occur at the Web server. Toolkits such as PhoneGap simplify this process even further by providing a common template. To simplify discussion, this paper terms these hybrid applications as *WebView apps*.

There are both security and privacy benefits and drawbacks to WebView apps verses using WebApps in the Web browser. On the positive side, WebView apps are treated as security principles within their native platforms. This separation provides extra protection of user credentials and other sensitive data. WebView apps can also deter phishing. Once a user downloads a native app (e.g., a banking app), she becomes implicitly trained to access the service through the phone's launcher, and potentially less likely to be fooled by a link in an Email. Finally, WebView apps have separate cookie storage, which limits cross-site privacy concerns. For example, if a user is logged into Facebook in the Web browser, whenever the users visits a Website with a Facebook "like button," Facebook is notified. In contrast, if the user accesses Facebook via a native or WebView app, the user's authenticated Facebook cookies are not present in the Web browser. Similar privacy concerns with Website advertisements are also mitigated.

WebView apps also have security and privacy drawbacks. WebView apps are generally relatively simple and their core function-

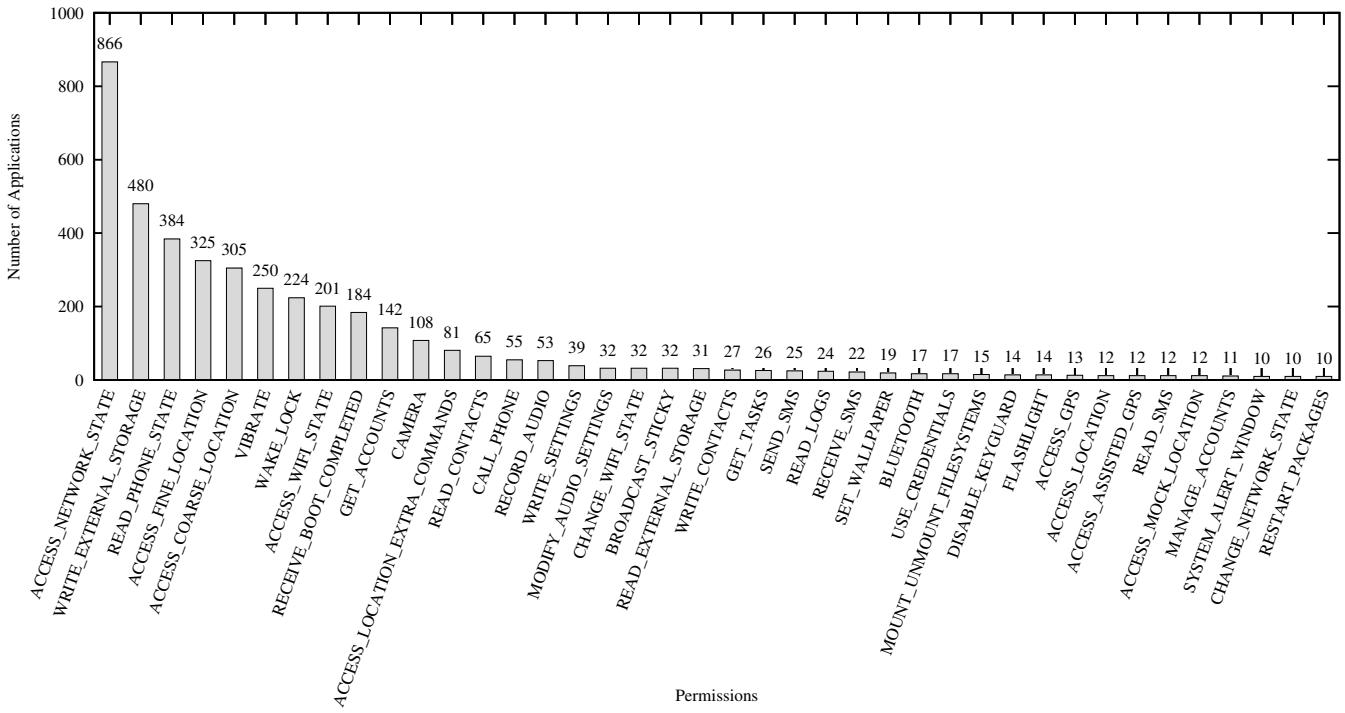


Figure 1: Permissions frequently requested by WebView apps (only $>$ 10 shown)

Table 1: Application Survey Results

Characteristic	# of Apps	Percentage
Total Apps	12,500	100.00%
Apps that use WebViews	10,165	81.32%
WebView Apps	1,066	8.52%
Potentially over-privileged WebView Apps	999	7.99%

ality requires little more than permission to access the Internet. However, WebView apps often contain extra permissions. Many recent studies [18, 15, 32, 19] have identified privacy leaks of geographic location and phone identifiers, often by advertisement libraries [26]. Finally, WebView apps with extra permissions can potentially do more harm if exploited [21].

2.2 Application Survey

NativeWrap is an alternative to any mobile Website or native app that has a mobile Website. However, our primary target is to replace WebView applications, as they are little more than a WebView widget rendering the mobile version of a Website. To estimate a lower bound on the need for NativeWrap, we performed a survey of popular Android applications. Specifically, we sought to better understand *(1) what percentage of apps are WebView apps?* and *(2) what is the permission request profile of WebView apps?*

Our survey includes the top 500 free applications from each of the 25 application categories on the Google Play Store, as of January 2013. We excluded game and widget categories, as they are rarely full screen WebView apps. We disassembled the applications using baksmali [50] and extracted the `AndroidManifest.xml` file for each app using AXMLPrinter2 [2]. We then used lightweight static code analysis heuristics to classify the apps (described below). Our survey results are summarized in Table 1.

Counting WebView apps: We identified WebView apps in two steps. First, we used grep on the dissembled code to identify all applications that create or initialize WebView objects with URLs. We found that roughly 81% of applications used WebViews. However, upon closer inspection of randomly chosen applications, we found many apps use WebViews for extra functionality such as displaying company policies or advertisements. To estimate the lower bound of WebView apps, we identified the applications that use WebViews within the file that contains its main activity class. The main activity is specified in the application's `AndroidManifest.xml` and defines the first activity component started when the application is launched. If an app uses a WebView in its main activity class, it is highly likely that WebViews are core to the app's functionality. However, we stress that this is a lower bound, because developers may place WebView initializers in other classes called by the main activity class. This second search strategy identified 1,066 apps, or 8.52% of our sample set, which is significant enough a percentage of applications to be concerned about.

Permission use by WebView apps: Having identified a lower bound on the percentage of WebView apps, we turned to their security and privacy implications. Ideally a WebView app should only require the INTERNET permission. However, we found that nearly all of the identified WebView apps (≈93%) required more permissions. These applications are called "potentially over-privileged WebView apps" in Table 1. Users installing these WebView apps have no way to deny specific undesired permissions.

The WebView apps requested a total of 436 unique extra permissions, of which 333 were custom permissions declared by the applications themselves. Figure 1 further breaks down the frequency of popularly requested permissions. The figure shows that most WebView apps request ACCESS_NETWORK_STATE, which is used to determine if the phone has a data connection, and can be used to differentiate cellular and WiFi connections. Addition-

ally, WRITE_EXTERNAL_STORAGE is reasonable for WebView apps storing caches on the SDcard. However, Figure 1 shows a wide variety of privacy and security relevant permissions. We note that the phone state and location permissions are the next highest requested permissions. These results clearly indicate WebView apps present privacy concerns.

Stowaway [21]: To characterize how many requested permissions are actually used by WebView app code, we analyzed 50 randomly selected applications with Stowaway [21]. We only analyzed 50 applications because Stowaway is not a stand-alone application and required us to manually upload the applications to a Website. The Stowaway results are useful as they help describe the potential for increased damage if the WebView app is compromised (e.g., due to a vulnerability in WebKit). We found that half of the 50 apps requested permissions that are never used. This result indicates that NativeWrap can also help increase application security.

2.3 Threat model

A fundamental premise behind our work is that both apps and mobile Websites have advantages and disadvantages with respect to security and privacy. Our NativeWrap solution is designed to leverage the advantages of each while removing the disadvantages.

Mobile applications are written by potentially untrusted third-party developers. Recent studies have clearly demonstrated that many legitimate (i.e., non-malware) apps leak privacy sensitive values such as phone identifiers, location, and address books [18, 32]. Often, these privacy leaks are a result of advertising and other non-required functionality. We seek to eliminate privacy loss due to non-required functionality.

Accessing mobile Websites through the device's Web browser also has security and privacy threats. We summarize these threats as follows.

Cross-site Attacks: WebApps contain Web elements from different origins. These elements can store cookies with the Web browser, and are frequently aware of the WebApp they are embedded within. By storing and retrieving cookies, the owners of these elements can track user's browsing habits. For example, consider a user logged into Facebook. Whenever the user visits a Website that embeds a Facebook "like button," Facebook is notified that the user visited the page, even if the user does not click the button [14]. Further investigations found that logging out of Facebook is not enough [11, 49]. To regain privacy, the user must clear the cookie store. Similar privacy concerns arise with Web advertisements that store cookies, i.e., a privacy concern DoubleClick is infamously known for. Browser state, including a range of browser cache methods, can be used to track the user [35]. By having per-WebApp cookie stores and state, NativeWrap significantly mitigates, if not removes, such privacy threats.

Phishing: Phishing attacks commonly trick users into clicking on URLs that direct them to a Website pretending to be the original (e.g., a bank Website). Web browsers on smartphones often make this easier, because the browser hides the address bar to maximize the page viewing area [22]. An example of such an attack is "Tabnabbing" [46], wherein the attacker loads a fake page resembling some recently used website's login page into a browser tab that has been open, but inactive for a while. If the user is convinced the page is authentic, she may enter her credentials. NativeWrap seeks to mitigate such attacks by always clearly displaying the WebView app's name. NativeWrap further pins the WebView app to a domain to ensure phishing does not inadvertently originate from the domain, e.g., via advertisements that hijack the screen [1].

Browser Compromise: Upon compromising the Web browser, an attacker potentially gains access to all of the user's cookies, including those that are used for authentication. The compromise could also result in a Man-in-the-Browser attack [29], wherein the compromised browser logs all user activity and input. NativeWrap mitigates these threats by treating each WebApp as a different security principal in the host operating system. This includes separate cookie stores and separate runtime principals for each WebApp. We note that newer Web browser architectures such as Chrome for Android also provide defenses against such attacks. A more detailed comparison is provided in Section 7.

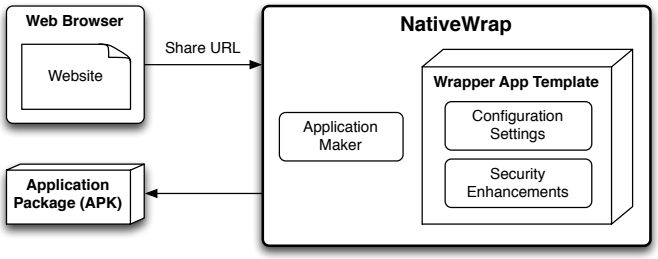


Figure 2: The NativeWrap Architecture

3. NATIVEWRAP DESIGN

NativeWrap provides an alternate model for accessing Web-based content by providing a balance between installing a third-party application and using the phone's Web browser. NativeWrap seamlessly allows end users to create safe and privacy friendly applications for any Website. To do this, the user must first visit the desired Website in the phone's Web browser. Once loaded, the user selects the "share" action that is often used to share a URL with messaging and social networking applications. When the user shares the URL, NativeWrap is available as a share target. Once NativeWrap receives a URL, it presents configuration screen to the user. NativeWrap uses the URL to specify best practices defaults (e.g., forcing SSL, CA pinning). Once the configuration is confirmed, NativeWrap parameterizes a pre-made WebView wrapper template and installs the newly created application package. This architecture is shown in Figure 2.

The remainder of this section describes the objectives and design of NativeWrap. We note that while many parts of the discussion are Android specific, NativeWrap is more general. We use Android where necessary to provide simplified and concrete discussion. Android also allows us to build and distribute a working NativeWrap prototype. We did not consider the other smartphone platforms for the prototype, because they cannot install applications without distributing them through the official application market. However, this need not necessarily be a limitation of NativeWrap. Other smartphone platforms (e.g., iOS) could easily include NativeWrap as part of the OS and provide it the ability to install the created applications.

3.1 Design Objectives

The primary objective of NativeWrap is to provide the user with a secure alternative to using WebView apps provided by third parties or accessing a WebApp via the browser. As such, NativeWrap seeks to achieve the following design objectives.

1. Regulated permission set: The WebView app should operate with the bare minimum privileges, i.e., network access. If additional privilege is required (e.g., to access external storage to upload photographs), the user should be provided the option to grant it. However, only network access should be enabled by default, and

the WebView app should operate correctly with only network access (with the exception of the function requiring more privilege).

2. Separate WebApp-specific resources: In the browser, WebApps share a cookie store, bookmarks, and history. If the browser is compromised, the authentication cookies of all WebApps may be compromised. Furthermore, the same origin policy is insufficient to prevent privacy loss when a cookie provider is included as a page element on many Websites. Therefore, NativeWrap seeks to ensure separation of these resources. The resources should be specific to the WebApp; other WebApps should not be loaded into the original WebApp's container.

3. Application-specific SSL configuration: Web browsers must support the SSL needs of all Websites. In contrast, a NativeWrap app needs only to support the SSL needs of one Website. This feature must be leveraged to ensure the best possible SSL configuration for the app, including pinning the app to a CA certificate and forcing SSL if possible.

4. Execution of trustworthy code: The created WebView app should be free from known vulnerabilities and execute in a predictable manner. It should also prevent malicious arbitrary code from executing, and should be resistant to confused deputy attacks.

3.2 Design Elements

We fulfill these design objectives on Android in four parts: a secure configurable wrapper, domain pinning, SSL pinning, and forcing HTTPS where possible.

3.2.1 Secure Configurable Wrapper

In order to keep the resources of WebApps isolated, we wrap WebApps into native Android applications. Each Android application has a unique Linux UID making it a unique security principal. Therefore, native Android apps cannot access the private storages of other apps. By using this separation, we ensure protection for resources such as the cookie stores, saved passwords, etc.

Our native application template is actually an Android application built using a WebView as its primary layout view. The WebView is configured to display the WebApp associated with the URL supplied by the user. An alternate approach would have been modifying the default Android Open Source Project (AOSP) browser to support a single WebApp. After briefly considering this option, we determined that refactoring the browser app was a complex and error prone process that may leave unknown vulnerabilities. Therefore, we opted for a clean design.

We configure our wrapper template to request only the INTERNET permission. While studying WebApps, we recognized that some Websites allow users to upload files (e.g., photographs). WebViews can be programmed to relay file upload events to the Android OS. This feature will require the READ_EXTERNAL_STORAGE permission in future Android releases. Therefore, NativeWrap offers the user the option to add this permission while configuring the wrapper. Furthermore, the wrapper template is configured to only upload a file via the Android OS. Hence, the resulting app cannot directly access the external storage without the user's knowledge.

We note that NativeWrap could be too restrictive for some applications that genuinely require certain permissions (e.g., location) to execute their primary functionality. Our goal behind NativeWrap is to put the user in control, and such optional permissions can be added to NativeWrap's implementation if necessary.

3.2.2 Domain Pinning

The wrapper template is a native Android app that ensures that other native applications do not have access to the private resources of the WebApp wrapped in the template. To describe domain pinning, we call this wrapped WebApp the "primary WebApp" and the corresponding URL the "primary URL." Domain pinning only affects the primary URL and not resources referenced by that page.

If the user navigates outside the primary WebApp, she may be exposed to phishing or cross-site attacks. These attacks often rely on the browser's ability to load multiple WebApps, which then share the same resources such as cookie stores, history and bookmarks. To prevent these attacks, we make the wrapper WebApp-specific by configuring the WebView to only work with the primary domain. Requests outside this domain are forwarded to the phone's default Web browser. To ensure the user is aware of this transition, we always display the name of the WebView App at the top of the screen. We also display a non-intrusive toast message when transitioning to the Web browser.

NativeWrap identifies the domain for the primary WebApp from the URL specified by the user. During our experimentation with initial versions of NativeWrap, we found that the full domain is not always appropriate. For example, `www.bestbuy.com` redirects to `www-ssl.bestbuy.com` for user login. Therefore, pinning the WebApp to `www.bestbuy.com` will not allow the user to log in, because the authentication cookies will be stored in the phone's browser. In this case, it is better to pin the WebView to `bestbuy.com` and allow all subdomains.

Pinning the WebApp to the second-level domain (e.g., `bestbuy.com`) is not always appropriate. For example, if the user is wrapping `foo.blogspot.com`, `blogspot.com` is too broad. However, we anecdotally observed that pinning the third-level domain is required significantly less frequently than the second-level domain. Therefore, we use the second-level domain as the default configuration, but also display the third-level domain as a clear option. We believe the cases when the third-level domain is needed will be obvious to most users.

Our experimentation with NativeWrap also uncovered redirection to other second-level domains. For example, `blogspot.com` redirects to `accounts.google.com` for authentication. Many Websites use third-parties such as Google and Facebook to authenticate. To address third-party authentication services, we suggest a whitelist solution. There are a relatively small number of authentication providers, which can be easily enumerated within the template. Furthermore, these domains generally are not the source of phishing attacks. Our current implementation only includes `accounts.google.com` and `facebook.com`, but additional entries can be easily added.

We note that including Facebook as trusted domain does not introduce privacy concerns unless the user actually logs into the WebApp via Facebook. In this case, Facebook may be notified of page visits within the primary WebApp if those pages contain Facebook like buttons.

3.2.3 SSL Pinning

Recent CA compromises have confirmed worst fears about the flaws of the CA model. An attack on Comodo in March 2011 resulted in it issuing 9 fake certificates for Websites including Google, Microsoft and Skype [40]. DigiNotar was compromised several months later [38], with the attacker(s) being able to issue over 500 fraudulent certificates, including a wildcard certificate for Google.

Fake SSL certificates are not limited to adversarial CA compromises. Nation states and other governing bodies can also force CAs to issue fake certificates. According to the Electronic Frontier Foundation's SSL Observatory, there are about 650-odd organizations that function as CAs [53]. An Android version ships 100s of such trusted CA certificates in its KeyStore, 140 for Android 4.2 [17]. If any one of these CAs is compromised, a fake SSL cer-

tificate for any Website can be created, allowing the holder of the fake certificate to perform DNS redirection or MITM attacks.

In the wake of the CA compromises and growing cyber-political tension, researchers have given increased attention to the CA model. Convergence [42] is a promising solution resulting from this discourse. Convergence is based on the idea of "trust agility," where the user chooses a set of notaries to validate certificates, and multiple notaries can be added or removed as needed. Notaries situated in different geographic areas can further reduce the possibility of an attacker fooling all notaries. One option is to include a Convergence module into NativeWrap. This would need to be coupled with defining an initial set of notaries, as well as allowing the user to configure the notary template used for all newly created applications. However, we currently use a simpler, and perhaps more appropriate mechanism: SSL CA pinning.

Creating WebApp-specific native applications makes NativeWrap suitable for using SSL CA pinning. Individual WebApps commonly only use one CA, therefore, it becomes possible to pin a root CA certificate to a particular wrapper application. SSL CA pinning significantly reduces the attack surface for many WebApps. For example, since Google uses Equifax as a CA, a compromise of Comodo would not affect the created WebView app. In fact, many third-party developers have begun using SSL pinning for their native apps. Unfortunately, doing so has proved to be error prone [20].

NativeWrap uses a first-use approach to acquire the CA certificate for the WebApp loaded in the native wrapper, i.e. we extract the CA certificate associated with the URL passed to NativeWrap. We then configure a TrustManager for the WebView class that only allows that root CA for SSL verification. We note that this approach is less flexible than Convergence, as the WebApp may wish to change its CA, which would require the WebView app to be recreated. This is not a problem for WebView apps created by third-parties, as they could simply distribute an updated version in the application market. The first-use approach is also subject to compromise during acquisition of the CA certificate used for the pinning. Finally, WebApps that use multiple CAs may nondeterministically fail. However, we did not experience any such problems during our compatibility study described in Section 5.1.

3.2.4 *Force HTTPS*

Many Websites provide both HTTP and HTTPS versions of their content. Unfortunately, URL references in content do not always use the HTTPS version of a URL when the user is visiting the HTTPS version of the site. ForceHTTPS [34] is a solution that allows Website owners to configure the site to inform the browser that HTTPS should be used for all connections. However, to take advantage of ForceHTTPS, the user must be aware that an HTTPS version of the site is available. For example, Google Search provides both HTTP and HTTPS versions, and until only recently, the user would need to type "`https://`" to visit the HTTPS version. To take advantage of the optional HTTPS versions of Websites, the Electronic Frontier Foundation (EFF) created the HTTPS Everywhere project [16]. This project provides an extension for Firefox and Chrome that consults a regular expression based rule set identifying Websites that have an HTTPS version. Users using the extension can ensure that they visit the HTTPS version of a Website whenever possible, without the need to type "`https://`".

We have incorporated the HTTPS Everywhere concept into NativeWrap. When the user shares a URL with NativeWrap, NativeWrap consults the HTTPS Everywhere rule set to determine if an HTTPS version of the Website is available. If so, the NativeWrap configuration template includes a "ForceHTTPS" checkbox, with the value selected by default.

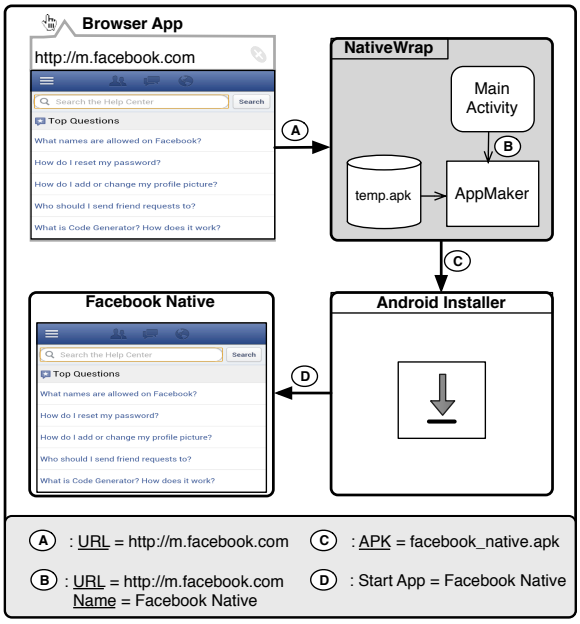


Figure 3: NativeWrap Implementation: Wrapping the *Facebook* WebApp to create the *Facebook Native* application.

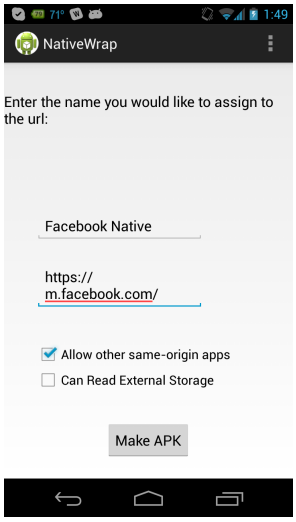
If the user creates the app with the ForceHTTPS option enabled, the matched rule is included in the created WebView app. When the user uses the app, the rule is matched against every visited URL, substituting the HTTPS version whenever possible. Packaging a single rule works, since the wrapper is pinned to a single domain. This also works if the user selects the option to pin the wrapper to pin to the domain of the origin (e.g., `*.google.com` instead of `images.google.com`). In this case, the rule for `*.google.com` is applied, covering all its sub-domains.

We know that there are multiple ways to maintain the HTTPS Everywhere rule set. One option is to hard-code the rule set into the NativeWrap app, and update it by distributing a new version through the application market. However, this method is slow and potentially annoying for users. Therefore, NativeWrap currently retrieves the ruleset by making a secure connection to our remote server, where the rules are stored and regularly updated as soon as the EFF git repository is updated.

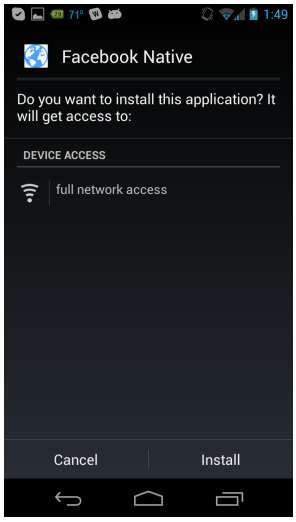
4. IMPLEMENTATION

In this section, we describe the implementation of NativeWrap for the Android OS. We describe the basic flow of events that takes place when a URL is native-wrapped. The core NativeWrap logic is implemented as an Android application that can be installed on any Android phone. The application includes a wrapper template that is in and of itself an Android APK package. An example execution using Facebook is shown in Figure 3. The source code for NativeWrap can be found at `http://research.csc.ncsu.edu/security/nativewrap/`.

1. Sharing the URL: The process begins with the user visiting the target URL in the phone's Web browser. Web browsers commonly have a "share" function that calls `startActivity` with an intent addressed to the ACTION_SEND action string and a data field containing the URL string of the current page. When Android resolves ACTION_SEND, multiple targets are available, therefore it opens a chooser dialog that allows the user to choose the target. NativeWrap defines an intent filter for ACTION_SEND on its main



(a) NativeWrap Configuration Screen



(b) Android Installer

Figure 4: Configuration and installation of a *Facebook Native* app.

Table 2: HTML5 Compatibility Score Comparison

Feature (Max points.)	Google Chrome	NativeWrap
Parsing rules (10)	10	10
Elements (30)	25	25
Forms (110)	106	106
Microdata (5)	0	0
Location and Orientation (20)	20	20
Output (10)	5	5
Input (20)	13	3
User Interaction (25)	20	20
Performance (25)	25	20
Security (40)	28	28
History and Navigation (10)	10	10
Communication (35)	35	35
Video (35)	35	35
Audio (30)	25	20
Peer To Peer (15)	15	0
2D Graphics (25)	19	19
3D Graphics (25)	20	0
Animation (5)	5	5
Web Applications (20)	16	15
Storage (30)	28	28
Files (10)	10	10
Other (20)	14	14
Total (555)	**484**	**428**

activity. As such, NativeWrap is started automatically by Android, and there is no need for a persistent service.

2. Customizing the Wrapper: Once NativeWrap receives the intent, it extracts the URL and populates the configuration template with defaults, as described in Section 3. At this point, the user can modify the URL, the pinned domain, specify an application name, enable additional permissions, etc., as shown in Figure 4(a). The user then chooses to "Make the APK", which sends the customized parameters to the *AppMaker*, which is a private activity component.

3. The AppMaker: When AppMaker receives the customized parameters, it copies the default wrapper APK file to `temp.apk`. This APK is already configured to support SSL pinning, domain pinning, and some usability features to support a maximum number of web applications. It is also designed to retrieve the URL from an XML file in the `/assets` directory within the APK.

The *AppMaker* first extracts the `AndroidManifest.xml` from `temp.apk`. We parse and modify the manifest file using AXML [5], as it is in a binary XML format. We change the package name to "`com.nativewrap.wrapped<SERIAL>`", where "`<SERIAL>`" is a 32 bit integer serial number that is incremented to avoid repeating package names. NativeWrap does not use part of the URL as the package name to allow the user to make multiple WebView apps for the same domain, e.g., with different security settings.

AppMaker changes the package name only in the manifest file. It does not rebuild the application. To ensure that the application executes correctly, we use the full classname of activity components specified in the manifest. Using the default relative class names attempts to call a nonexistent class, since the package name in the manifest no longer matches the prefix on the Java classes.

Next, AppMaker modifies the `label` attribute of the main activity. This is the activity started by the phone's application launcher, and changing its `label` to the application name specified by user ensures the user can easily find the WebView app in the list of icons and in the settings menus. Additionally, if the user chooses external storage read access, AppMaker adds a `<uses-permission>` specification for READ_EXTERNAL_STORAGE.

Finally, AppMaker creates a new XML file for the Website URL, adds that file and the modified manifest file to `temp.apk`. The resulting package is signed with a prespecified key and renamed to `<application-name>.apk`. In order to install the `.apk`, the installer must be able to read the file. The most obvious place to store the `.apk` is the SDcard, which is effectively readable by all applications. However, the SDcard is also effectively writable by all applications. If the `.apk` is writable, a malicious application may exploit a race condition by modifying the file before it is installed. To avoid this race condition, we place the `.apk` in the root of NativeWrap's `/data` directory and make the file world readable. Passing the full file path to the installer allows the package to be installed.

4. Installing the APK: Once the APK is created, AppMaker sends an intent message to the system with the full path to the APK to initiate its installation. As shown in Figure 4(b), this intent invokes the Android's installer, which presents the user with a screen to install the application. Once the user approves the permission list, the WebView app is available in the phone's application launcher.

5. EVALUATION

We begin the evaluation by comparing the HTML5 compatibility of NativeWrap with Google Chrome for Android, studying how NativeWrap affects the compatibility of WebApps. Then, we describe two case studies to demonstrate the functionality and security benefits of NativeWrap.

5.1 Compatibility

We test NativeWrap compatibility in two ways. First we test raw HTML5 compatibility using a standard benchmark. We then manually evaluate the top 250 Alexa Websites.

5.1.1 HTML5 Compatibility Test

We performed a compatibility test for HTML5 support using `html5test.com`, on a Nexus 4 running Android 4.4.2. This test evaluates a Web browser on how well it supports the upcoming HTML5 standard, and generates a cumulative score chart for each aspect examined. We also compare the NativeWrap results with Chrome for Android (available for Android 4.0 and later).

Table 2 gives a comparison of Chrome for Android's and NativeWrap's wrapper's performance in the HTML5 compatibility test. We also note that NativeWrap performed exactly as well as the stock Android 4.4 browser, and much better than the reported values for the stock Android 4.0 browser (272 points as per `html5test.com`[1]), which confirms our choice to build the wrapper from scratch rather than refactoring the AOSP browser.

Our wrapper, and in turn the Android WebKit, only partially supports some HTML5 elements, while it does not support features like Microdata, 3D graphics, and peer to peer. However, we do support most other aspects of the standard, including form elements, essential parsing rules, audio, and video. NativeWrap's wrapper generally scores similar to Google Chrome for most of the features. Chrome scores better only in the input (access to webcam), audio (Web Audit API), peer to peer (WebRTC and Data Channel), 3D Graphics (WebGL 3D graphics), and Performance (Shared Workers), and Web applications (custom search providers) categories.

5.1.2 Alexa Top 250 study

To further verify our results, we manually tested NativeWrap for compatibility with the top 250 Websites in the world (filtering the duplicates, such as `google.in` and `google.cn`) from Alexa.com as of April 2013. Note that we skip websites in foreign languages that require login, therefore we actually consider the top testable 250 Websites. It is worth mentioning that as of September 2011, 34 of the top 100 websites had already converted to HTML5 [41]. Even by a conservative estimate, the number is likely to have gone higher since. We used a Samsung Galaxy Nexus phone running Android version 4.2.2 for this experiment and the case studies described in Section 5.2.

We made a native-wrapped application for each of these websites, and simultaneously tested the Website in Chrome for Android version 25. We tested the hypertext content as well as interactive multimedia content such as HTML5 audio and video tags, and also the intra-website navigation. None of the websites crash or exhibit broken functionality during our tests, with some minor exceptions[2], that exhibit similar behavior on the AOSP Browser as well due to HTML5 incompatibilities of the Webkit API. We infer the following from our results:

1) *Websites conservatively use HTML5 features*, using the ones commonly supported by most available browsers. For example, a developer would want to consider the Android 2.3 browser, which is still on about 46% of all Android devices as of February 2013 [27] and scores a modest 200 points on the compatibility test.

2) *Websites detect browser compatibility* and present only compatible features. Websites could also redirect the user to a HTML4 version, though we did not observe any redirection on our native-wrappers, possibly because it is compatible with most required HTML5 features that most websites currently use.

3) *Websites handle errors* and exceptions silently and transparently from the user, especially when they are related to HTML5, which is still not supported completely by most browsers.

4) *NativeWrap supports HTML4 content* well, and is completely compatible with websites that still work on HTML4.

5.2 Case Studies

5.2.1 Slick Deals

The Slick Deals WebApp keeps the user updated with the latest information on deals and offers on various products and services. The Android app for Slick Deals is a WebView application, and does not use the native Android User Interface to a great extent. It is a fairly popular application installed in around 100,000 - 500,000 devices, with a four star ranking on the Google Play store. The app loads a WebView with the web address of the mobile WebApp, i.e., `http://m.slickdeals.net`.

Slick Deals was one of the over-privileged applications obtained from our application survey described in Section 2. An analysis with Stowaway detected that the app requests the Android location permissions (both coarse and fine location), but does not use any API that require these permissions. Even if it did call API that requested location, its purpose of displaying online deals would not justify the need for location information.

We created a new Slick Deals app using NativeWrap for this case study. The Slick Deals mobile website worked just as well on the new app as it did in the browser. At the same time, the original Slick Deals app did not offer any more functionality than the native wrapped app, apart from a different font and color combination, but was in fact vulnerable to activity hijacking attacks when scanned with ComDroid [8].

5.2.2 Facebook for Android

Facebook tops the Alexa rankings as the most visited website worldwide as of April 2013. The Facebook app is also the most popular free Android app based on the number of installs from the Google Play Store, somewhere between 100-500 million as of April 2013. Based on the sheer number of users whose privacy depends on Facebook, it is an ideal candidate for a case study.

We compared three methods of accessing Facebook from an Android device: 1) the Facebook WebApp accessed via the phone's Web browser shown in Figure 5(a), 2) the Facebook for Android native app (version 3.1) shown in Figure 5(b), and 3) the native-wrapped version of the Facebook app shown in Figure 5(c). We evaluate each approach on two main factors: *usability* which measures the convenience and features offered to the user, and *security* which is based on the vulnerabilities in the approach, possible attack surfaces and potential privacy violations.

Accessing Facebook via the browser: As described in Section 2.3, using the Facebook app in the Web browser exposes the user to various privacy and security problems, for e.g., the Facebook 'like' button privacy issue or phishing attacks like the 'tabnabbing'. The other two approaches do not face such problems as they are directly installed as independent native applications on the smartphone, and have their own separate resources.

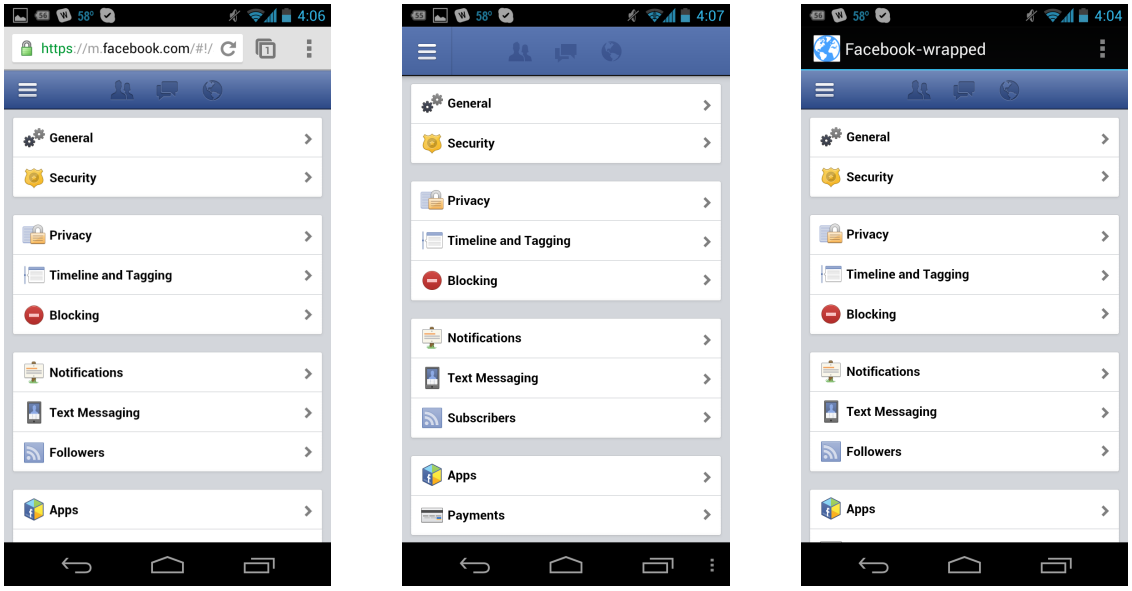
The browser based approach also lacks the convenience of using a native app, as the user has to go through an additional step, i.e., the Web browser. The other two approaches provide dedicated apps for Facebook, and the native Facebook for Android app also utilizes some of the smartphone's resources and UI elements to provide a more immersive experience. Therefore, the Web browser-based approach clearly does not measure up to other two approaches, both in terms of usability and security. Hence, we now only focus on the remaining two approaches.

Facebook for Android vs. Facebook-wrapped: For this evaluation, we created a native-wrapped Facebook application with the URL `m.facebook.com`. We call it "Facebook-wrapped". We compare both the approaches on the basis of usability and security.

Facebook-wrapped and the Facebook for Android app are identical in terms of performing all of the core Facebook functionality, such as browsing pages and profiles, liking and sharing objects, uploading pictures, managing the user's account and privacy settings, etc. Facebook-wrapped lacks three primary features that Facebook for Android provides: 1) Android notifications, 2) contacts inte-

[1]Results accessed May 14, 2014.

[2]*Dailymotion* plays only the audio part of a video clip occasionally.



(a) Facebook WebApp in Chrome browser

(b) Facebook for Android app

(c) Facebook-wrapped

Figure 5: Facebook Privacy Settings page.

gration, and 3) geo-location checkin. However, users willing to sacrifice these features can benefit from privacy advantages.

Both the Facebook for Android and the Facebook-wrapped app are installed as native applications, and hence are not affected by the threats faced by the Web browser-based approach. The Facebook-wrapped app can perform all of the core Facebook functionality. Therefore, ideally, Facebook for Android should also not require more than the Internet permission. This is not the case, because Facebook for Android has many value-add features such as taking pictures and geo-location check-in. However, Facebook for Android also requests a number of non-obvious permissions. For example, it can access call logs, contacts, and recently added a permission allowing it to track what applications the user is currently running [45]. While there are likely reasonable justifications for all of Facebook for Android's permission requests based on various integration features, the functionality is not required by all users.

Facebook-wrapped on the other hand does not require any special privilege other than network access and the permission to read external storage (API 17 onwards, optional). The primary observed drawback was the inability to use geo-location check in. However, we view Facebook-wrapped as a privacy friendly alternative to Facebook for Android. Users interested in these privacy benefits are less likely to use the location feature.

6. DISCUSSION

When designing NativeWrap, we debated between bundling it with a custom Android and creating a stand-alone third party application that can be downloaded from Google Play. Clearly, a stand-alone third party application is more desirable and will reach a wider audience. Unfortunately, this deployment approach requires the user to modify the "Unknown Sources" application side-loading security setting. That is, the user has to choose to allow apps from unknown sources to install on the phone. Considering that most users are not security experts, allowing side-loading of apps from unknown sources may make the user vulnerable to attacks by malicious applications. Expert users can reduce their vulnerability time frame by checking the option immediately before using NativeWrap, and unchecking it immediately afterwards. Testing showed that "Unknown Sources" was the only Android security option that needed to be disabled. NativeWrap was successfully tested with the "Verify Apps" feature activated.

The "Unknown Sources" limitation can be eliminated by making NativeWrap part of the Android OS. For example, NativeWrap could be deployed as a pre-installed system application and configured with the `ApplicationInfo.FLAG_PRIVILEGED` set in the package manager service. Doing so would inform the system package installer that NativeWrap install requests are not from an unknown source.

7. RELATED WORK

Web browser hardening: Web browsers are the central aspect of our Internet use. Anupam et al. analyzed JavaScript and VBScript based attacks on the Web application data in 1998 [3], and their work was one of the first to note how operating systems security primitives (e.g., 'ACL' [23], 'capabilities' [56, 39, 48]) apply to the multi-application environment in the browser. Since then, many approaches based on standard OS primitives have been proposed for enhancing the browser's security.

Tahoma [10] treats Web applications as first class objects, and uses virtual machines (VM) to isolate Web applications from each other and the browser from the underlying operating system. Each Web application instance starts in a new VM and has its own virtual disk space, screen, input devices, etc. A key difference with respect to NativeWrap is that Tahoma allows the Web application to specify the domains that will run in its VM instance in a manifest file. Delegating the browser configuration (domains to pin, security enhancements, etc.) to the Web application exposes the user to cross-site attacks and to some extent, phishing attacks described in Section 2.3. App Isolation [7] similarly allows web developers to configure domain pinning, and to optionally select isolated storage.

The OP Browser [28] splits the browser design into distinct function-specific components (e.g., webpage, storage, user interface) and makes the communication between these subsystems explicit, trusting the underlying operating system and the Java Virtual Machine (JVM) to maintain isolation between components. Such a model makes browser compromise difficult to achieve through exploits in individual subsystems, and provides strong isolation guarantees. Although OP Browser starts Web applications in new instances (processes), it still has a common cookie store for all Web application instances in the *storage* component. Although the reference monitor will follow the same origin policy, the common cookie store will lead to privacy issues such as the Facebook 'like' button problem. Instead of simply starting a new process, NativeWrap leverages the UID based separation provided by the underlying Android OS and ensures complete isolation between wrappers.

Google's Chrome for Android also leverages the UID based sandboxing provided by the Android OS. Every new browser 'tab' is started in a new principal instance, i.e., a process, and every such process has a different UID. This allows Chrome to regulate permissions allocated to each such principal, and provides isolation with respect to resources and data for each principal. A major limitation of the Chrome for Android browser is that is puts content from various origins in the same tab, i.e., in the same principal instance, meaning that the privileges allocated to a tab may still be accessible to the content from a different origin than the main content of the tab, leading to cross site attacks.

The Gazelle Web browser [54] recognizes the need for isolating Web application principals into separate instances. Content from different domains, even if accessed in the same tab or embedded in the same webpage, is put in separate principal instances. Therefore, Gazelle prevents embedded content of one principal executing code in another principal's context. In spite of such protections, principals in Chrome as well as Gazelle share common resources like cookie stores, which can result in privacy problems, some of which are described in this paper. The fundamental reason behind this difference is that NativeWrap's wrapper provides a single Web application environment, while Chrome for Android, Gazelle, OP browser and other similar approaches [52, 33, 36, 7] attempt to achieve complete app-specific isolation in a multi-app environment.

Privacy violations by native apps: Most Web browsers available today are vulnerable to many of the attacks described in our threat model (Section 2.3). Google's Chrome for Android is relatively resistant to browser compromise due to its UID based sandboxing, but is still vulnerable to phishing and cross-site attacks. Native WebView apps defined in Section 2.1 by default do not share browser state and cookie stores, and hence are not vulnerable to cross-site or browser phishing attacks. Nevertheless, native WebView apps that are over-privileged cause privacy concerns [18, 32, 19, 26, 25, 51].

There are different strategies for preventing privacy violations by such applications. Aurasium [57] repackages Android apps to make them policy compliant and to prevent privilege escalation attacks. A similar approach is taken by Dr. Android [37] and RetroSkeleton [12]. TISSA [58] allows the user to manage the private information granted to the app both during and after installation. It also has a provision to supply applications fake information. Apex [43] retrofits the Android package installer to install an application with custom policies. TaintDroid [18] uses taint tracking to alert the user when an application tries to export private data off the device. AppFence [32] and MockDroid [6] give the user a choice to provide fake information to apps that demand private data. In case the user needs to divulge information, AppFence prohibits the receiving app from exporting the data off the device.

Modifying an application package or its functionality may cause an application to break. Therefore, NativeWrap instead takes the control out of the hands of the developer, and packages a reliable template according to the security settings configured by the user.

Other WebApp wrappers: PhoneGap [44] allows developers to create native wrappers for HTML5 WebApps, and also provides JavaScript API to access the phone's resources. Thus, PhoneGap-based applications can potentially be just as privacy invasive as other native applications. PhoneGap is also only used by developers to wrap their HTML5 apps in native wrappers, and cannot be used by the user without the source code for the HTML5 app.

Finally, close in implementation, but drastically different in motivation, is the Fluid app [13]. Fluid is designed to create a native version of any Website for Mac OS X for user convenience. NativeWrap is designed specifically to address the security and privacy needs of smartphone users and is proposed as an alternate model for accessing Web content on smartphones. As such, Fluid does not provide the best practices security configuration provided by NativeWrap, nor does it provide the basic facility, i.e., a separate cookie store per wrapped WebApp in its free version.

8. CONCLUSION

Third-party native applications have become the *de facto* way for users to access Web content on smartphones. In this paper, we argued that native applications offer many security and privacy benefits over accessing the Web content using the phone's Web browser. Unfortunately, many of the native applications provided by third-parties contain privacy concerns in and of themselves. To resolve this tension, we proposed NativeWrap as an alternative approach for smartphone users to access Web content. NativeWrap "wraps" a given URL into a native application and applies security best practices configuration. In doing so, NativeWrap removes third-party developers from platform code and places users in control of privacy sensitive operation.

Acknowledgements

This work was funded in part by the National Security Agency, and NSF grants CNS-1222680 and CNS-1253346. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. We would also like to thank Tsung-Hsuan Ho, Ashwin Shashidharan, our shepherd Aurélien Francillon, and the anonymous reviewers for their valuable feedback during the writing of this paper.

9. REFERENCES

[1] C. Amrutkar, K. Singh, A. Verma, and P. Traynor. VulnerableMe: Measuring Systemic Weaknesses in Mobile Browser Security. In *Proceedings of the International Conference on Information Systems Security (ICISS)*, 2012.

[2] android4me - J2ME port of Google's Android. `https://code.google.com/p/android4me/`. Accessed August 2012.

[3] V. Anupam and A. Mayer. Security of web browser scripting languages: vulnerabilities, attacks, and remedies. In *Proceedings of the 7th USENIX Security Symposium*, pages 187–200, 1998.

[4] Apple. Apple Updates iOS to 6.1, Mar. 2013. `http://www.apple.com/pr/library/2013/01/28Apple-Updates-iOS-to-6-1.html`.

[5] axml - Read write android binary xml files. https://code.google.com/p/axml/. Accessed January 2013.

[6] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011.

[7] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App Isolation: Get the Security of Multiple Browsers with Just One. In *Proceedings of the 18th ACM conference on Computer and communications security. ACM*, 2011.

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.

[9] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-Related Policy Enforcement for Android. In *Proceedings of the 13th Information Security Conference (ISC)*, Oct. 2010.

[10] R. S. Cox, J. G. Hanson, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *2006 IEEE Symposium on Security and Privacy*, pages 15–pp, 2006.

[11] N. Cubrilovic. Logging out of Facebook is not enough. http://www.nikcub.com/posts/logging-out-of-facebook-is-not-enough, 2011.

[12] B. Davis and H. Chen. RetroSkeleton: Retrofitting Android Apps. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.

[13] T. Ditchendorf. Turn Your Favorite Web Apps into Real Mac Apps. http://fluidapp.com/about/, 2012. Accessed May 5, 2013.

[14] A. Efrati. 'Like' Button Follows Web Users. http://online.wsj.com/article/SB10001424052748704281504576329441432995616.html?mod=WSJ_Tech_LEADTop, 2011.

[15] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, Feb. 2011.

[16] Electronic Frontier Foundation. HTTPS Everywhere. https://www.eff.org/https-everywhere. Accessed April 2013.

[17] N. Elenkov. Certificate pinning in Android 4.2. http://nelenkov.blogspot.com/2012/12/certificate-pinning-in-android-42.html, 2012.

[18] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[19] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[20] S. Fahl, M. Harbach, T. Muders, L. Baumgartner, B. Freisleben, and M. Smith. Why eve and mallory love android: an analysis of android SSL (in)security. In *Proceedings of the 2012 ACM conference on Computer and communications security(CCS)*, 2012.

[21] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[22] A. P. Felt and D. Wagner. Phishing on Mobile Devices. In *Proceedings of the Workshop on Web 2.0 Security and Privacy (W2SP)*, 2011.

[23] G. Fernandez and L. Allen. Extending the Unix Protection Model with Access Control Lists. In *Proceedings of the USENIX Summer Symposium*, pages 119–132, 1988.

[24] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 38–49, 2012.

[25] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Trust and Trustworthy Computing, Lecture Notes in Computer Science Volume 7344*, 2012.

[26] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2012.

[27] D. Graziano. Jelly Bean's market share is up but Gingerbread just won't die. http://bgr.com/2013/02/08/android-version-distribution-february-2013-316698/, 2013. Accessed April 2013.

[28] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[29] P. Guhring. Concepts against Man-in-the-Browser Attacks. http://www.cacert.at/svn/sourcerer/CAcert/SecureClient.pdf. Accessed December 2012.

[30] J. Han, Q. Yan, D. Gao, J. Zhou, and R. Deng. Comparing Mobile Privacy Protection through Cross-Platform Applications. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[31] H. Hao, V. Singh, and W. Du. On the Effectiveness of API-Level Access Control Using Bytecode Rewriting in Android. In *Proceedings of the ACM SIGSAC Symposium on Information Computer and Communications Security (ASIACCS)*, 2013.

[32] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[33] L.-S. Huang, Z. Weinberg, C. Evans, and C. Jackson. Protecting browsers from cross-origin CSS attacks. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 619–629, 2010.

[34] C. Jackson and A. Barth. ForceHTTPS: Protecting High-Security Web Sites from Network Attacks. In *Proceedings of the 17th International ACM Conference on World Wide Web*, 2008.

[35] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th international conference on World Wide Web*, pages 733–744. ACM, 2006.

[36] K. Jayaraman, W. Du, B. Rajagopalan, and S. J.Chapin. ESCUDO: A Fine-Grained Protection Model for Web Browsers. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS)*, pages 231–240, 2010.

[37] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-Grained Permissions in Android Applications. In *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[38] D. Kaplan. DigiNotar breach fallout widens as more details emerge. http://www.scmagazine.com/diginotar-breach-fallout-widens-as-more-details-emerge/article/211349/, 2011.

[39] P. A. Karger and A. J. Herbert. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1984.

[40] W. Leonhard. Weaknesses in SSL certification exposed by Comodo security breach. https://www.infoworld.com/t/authentication/weaknesses-in-ssl-certification-exposed-comodo-security-breach-593, 2011.

[41] K. Maine. Percentage of Web sites Using HTML5. http://www.binvisions.com/articles/how-many-percentage-web-sites-using-html5/, 2011. Accessed April 2013.

[42] Moxie Marlinspike. Convergence. http://convergence.io/. Accessed March 2013.

[43] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of ASIACCS*, 2010.

[44] PhoneGap. http://phonegap.com/about/, 2012. Accessed May 5, 2013.

[45] E. Protalinski. Facebook's Android app can now retrieve data about what apps you use. http://thenextweb.com/facebook/2013/04/13/facebooks-android-app-can-now-retrieve-data-about-what-apps-you-use/, 2013.

[46] A. Raskin. Tabnabbing: A new type of phishing attack. http://www.azarask.in/blog/post/a-new-type-of-phishing-attack/, 2010.

[47] Scott Thurm and Yukari Iwatani Kane. Your Apps Are Watching You. http://online.wsj.com/article/SB10001424052748704694004576020083703574602.html.

[48] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.

[49] B. Slawski. Facebook Patent Application Describes Receiving Data from Logged-Out Users to Target Ads. http://www.seobythesea.com/2011/09/facebook-patent-application-target-ads/, 2011.

[50] smali - An Assembler/Disassembler for Android's dex Format. https://code.google.com/p/smali/. Accessed April 2013.

[51] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *IEEE Mobile Security Technologies (MoST)*, 2012.

[52] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[53] The Electronic Frontier Foundation. EFF SSL Observatory. https://www.eff.org/observatory. Accessed October 2012.

[54] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principle OS Construction of the Gazelle Web Browser. In *Proceedings of the USENIX Security Symposium*, 2009.

[55] B. Womack. Google Says 700,000 Applications Available for Android. Bloomberg Businessweek, Oct. 2012. http://www.businessweek.com/news/2012-10-29/google-says-700-000-applications-available-for-android-devices.

[56] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating Systems. *Communications of the ACM*, 17(6), June 1974.

[57] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the USENIX Security Symposium*, 2012.

[58] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST)*, June 2011.