AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone

Wenhao Li, Haibo Li, Haibo Chen, Yubin Xia
Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University

ABSTRACT

Mobile advertisement (ad for short) is a major financial pillar for developers to provide free mobile apps. However, it is frequently thwarted by ad fraud, where rogue code tricks *ad providers* by forging ad display or user clicks, or both. With the mobile ad market growing drastically (e.g., from $8.76 billion in 2012 to $17.96 billion in 2013), it is vitally important to provide a verifiable mobile ad framework to detect and prevent ad frauds. Unfortunately, this is notoriously hard as mobile ads usually run in an execution environment with a huge TCB.

This paper proposes a verifiable mobile ad framework called AdAttester, based on ARM's TrustZone technology. AdAttester provides two novel security primitives, namely *unforgeable clicks* and *verifiable display*. The two primitives attest that ad-related operations (e.g., user clicks) are initiated by the end user (instead of a bot) and that the ad is displayed intact and timely. AdAttester leverages the secure world of TrustZone to implement these two primitives to collect proofs, which are piggybacked on ad requests to ad providers for attestation. AdAttester is non-intrusive to mobile users and can be incrementally deployed in existing ad ecosystem. A prototype of AdAttester is implemented for Android running on a Samsung Exynos 4412 board. Evaluation using 182 typical mobile apps with ad frauds shows that AdAttester can accurately distinguish ad fraud from legitimate ad operations, yet incurs small performance overhead and little impact on user experience.

1. INTRODUCTION

Mobile ad is a key pillar to the mobile app ecosystem: developers provide free mobile apps with embedded ad libraries from *ad providers* to users and get paid by the ad provider for the ads displayed and/or clicked. A recent report shows that the mobile ad market revenue increases from $8.76 billion to $17.96 billion and is predicted to reach $31.45 billion in 2014 [19].

Unfortunately, this huge-revenue ecosystem is severely thwarted by ad frauds, where rouge code tricks the ad provider with forged display (also called *impression* [26]) or clicks. For example, recent studies pointed out that ads in mobile apps are plagued by various types of frauds: mobile ad providers are estimated to lose nearly 1 billion dollars in 2013 due to these frauds and around one third of mobile ad clicks may constitute click-spam [18]. The most recent research study [33] shows that, one of the largest click fraud botnets, called ZeroAccess, induces advertising losses on the order of $100,000 per day. Ad frauds can typically be characterized into two types [26]: (1) *Bot-driven frauds* employ bot networks to initiate forged ad impressions and clicks; (2) *Interaction frauds* manipulate visual layouts of ads to trigger ad impressions and unaware clicks from the end users.

Because of the urgent need to detecting mobile ad frauds, prior approaches have made important first steps by using an offline-based approach [16, 26]. Specifically, they trigger the execution of mobile apps in a controlled environment to observe deviated behavior to detect ad frauds. However, such approaches are limited to *interaction frauds* whose behavior can be triggered by automatic testing; it is hard for them to detect *bot-driven frauds* because botnets are compromised bots (thus real devices) that have been installed with the ad fraudulent applications, which could be instructed to make arbitrary network requests stealthily to users (such as ZeroAccess [33]). Further, such approaches suffer from both relatively high false positives and false negatives because the lack of reliable proofs to ad providers for attestation.

Reliably and securely detecting ad fraud raises two important questions: 1) *display integrity*: how could the ad providers know that their ads are displayed in the users' screen without their advertising policies being violated? 2) *user action authenticity*: how do the ad providers know that an ad request is from a user click but not a bot? Unfortunately, answering the two questions in current mobile platforms are notoriously hard due to the complex software stack that is potentially controlled by malicious botnets and/or rouge app code, which can forge arbitrary actions on mobile ads without users' awareness, making it hard (if not impossible) for ad providers to distinguish such forged actions from legitimate ones in a reliable way.

In this paper, we propose AdAttester, a system that reliably detects and prevents well-known ad frauds. The key insight of AdAttester is to provide attestable proofs to ad providers. To achieve this, we provide two novel security primitives:

- *Unforgeable clicks*: A user action, such as an ad click, could not be forged by malicious code that sends ad click network requests silently without users' awareness. Thus, ad providers can reliably detect the forged ad clicks.

- *Verifiable display*: If an ad's content is displayed on the mobile screen, a verifiable proof of the display content and the displaying duration will be generated and sent to the ad provider for verification.

MobiSys'15, May 18–22, 2015, Florence, Italy.

http://dx.doi.org/10.1145/2742647.2742676.

AdAttester uses a trusted extension of hardware, namely ARM TrustZone [10], to implement these two primitives, which avoids the need to trust the mobile software stack. AdAttester leverages the split execution mode in TrustZone and assigns specific peripheral devices to be accessible by the secure world only. Specifically, AdAttester puts the touch and display device drivers to the secure world, so as to securely get the ad click event and read the display content. To ensure that the user actions are unforgeable, AdAttester will sign each touch input whenever the touch positions lies within the ad view area, then send the signed touch inputs and the *eigenvalue* of the clicked ad area in the display screen to the ad provider (i.e., ad server) together with the ad request. Providing verifiable display is done by checking the ad display content periodically and calculating the *eigenvalue* of the display content. These eigenvalues will also be sent to the ad server together with the ad request; the ad providers will compare the requested eigenvalues with the correct ones that are calculated at the time of releasing the ads.

With these two security primitives, the bot-driven frauds and interaction frauds can be detected reliably. The ad providers may define some policies to determine whether a sequence of ad requests conform the predefined policies and whether such ad requests should be paid.

To deploy AdAttester, mobile users need to install a small trusted ad attestation application in secure world and AdAttester provides a list of ad attestation APIs in normal world OS for ad providers to adopt to their ad SDKs. Note that the trusted ad attestation application needs to be certified by trusted parties and unauthorized code such as code provided by ad providers could not run in secure world, and the TCB of AdAttester is small: only the hardware and software running in secure world of TrustZone.

We have implemented AdAttester in a Samsung Exynos 4412 board. We show that AdAttester can effectively detect all types of mobile ad frauds found in [16]. Evaluation shows that AdAttester incurs small performance overhead and little impact on user experience.

In summary, we make the following contributions:

- Two novel security primitives: *unforgeable clicks* and *verifiable display*, which provide attestable proofs of user interaction and can be deployed to effectively detect and prevent mobile ad frauds.
- The design and implementation of a verifiable mobile advertising framework named AdAttester, the first system to detect and prevent well-known mobile ad frauds online on the real hardware with small overhead.
- A comprehensive evaluation on performance and using a set of mobile apps with ad frauds to confirm the effectiveness and efficiency.

The rest of this paper is organized as follows. The next section describes some background information on mobile ads, ad frauds and ARM TrustZone. Section 3 describes the threat model, design goal and an overview of AdAttester. Section 4 presents the two primitives, followed by how they are leveraged for detecting ad frauds (section 5). Section 6 describes the security and deployment analysis. Section 7 describes some implementation issues using ARM TrustZone. Section 8 and 9 evaluate the effectiveness, performance and disruption of AdAttester. Section 10 discusses potential limitations of AdAttester and possible future work. Finally, section 11 relates AdAttester to state-of-the-art and section 12 concludes this paper.

2. BACKGROUND AND MOTIVATION

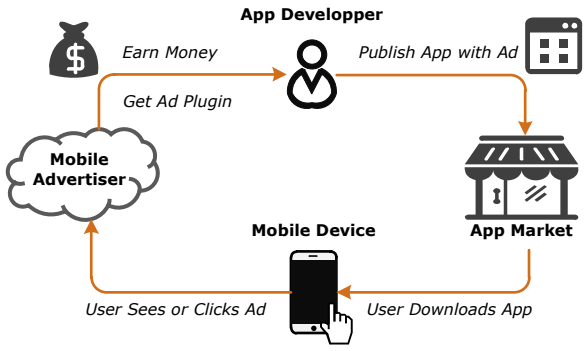

Figure 1: The ecosystem of mobile advertising

2.1 Mobile App Advertising and Ad Frauds

Mobile ads are the key to the ecosystem of mobile app development: developers design and implement mobile apps and distribute them to users for free; instead they get paid when users click or view ads, which are fetched from ad providers through the ad libraries embedded in mobile apps, as shown in Figure 1. Specifically, to embed ads in an app, an app publisher (developer) first registers with an ad provider, who provides the developer with a publisher ID and an ad library which will be embedded in her apps. Then the developer includes this ad library in her apps and allocates some screen real estate to the ads. The ad library is responsible for fetching, displaying and handling ads when the app is running. The app developer then uploads the apps to some app markets (e.g., Google Play) for users to download and use. App developer is paid by the ad provider based on the number of ad clicks or impressions, or both.

To make sure ads are delivered and displayed non-intrusively, ad providers usually have strict guidelines on how ads should be used and displayed in apps; otherwise the ad providers may refuse to pay for the ads. For example, AdMob's guidelines and policies clearly state that developers are strictly prohibited from using any means to inflate impressions or clicks artificially, and the number of visible ads per page must not exceed one [7].

App developers, however, have the incentives to violate those policies in order to earn more money, which are called violation frauds. Such frauds could be intentional or unintentional. In general, the types of ad frauds can be characterized into two categories [26]:

Bot-driven frauds: These frauds employ bot networks to initiate forged ad impressions and clicks. The bot networks could be compromised mobile devices, PCs or desktops, similar to the traditional web advertising bot networks. Violators may first carefully analyze the correct sequence of how a paid ad could be made successfully, and then distribute an automated tool to the bot network to process the ad requests secretly. Bot-driven frauds widely exist; a conservative estimation concludes that 2% to 40% of ad traffics are botnet traffic and the accumulated monthly figure could be $6.2 billion in web advertising [3, 5]. These frauds are now targeting at mobile ads [4].

Interaction-driven frauds: These frauds include *placement fraud* that ads are placed in violation of the ad guidelines or even use a display view to cover the ad view so as to receive impression fee while not affecting user experience. In general, this can be done in multiple ways, including resizing ads to too small to read, hiding them behind other display elements or placing them outside the display screen. There could also be *unaware fraud* that an app sends ad requests in the background or forges ad click requests

without user's consciousness. Evaluation by Crussell et al. [16] over 130,339 apps shows that 12,421 apps have background impressions and 21 apps fabricate user clicks. Liu et al. [26] show that 2% to 54% apps have certain types of placement frauds.

Although ad providers have guidelines to restrict the behavior of developers, there are few technical ways to guarantee that the guidelines are not violated. Currently, an ad server will analyze all the received clicks to identify suspicious ones that might be forged click, using some heuristic methods such as pattern matching or anomaly detection. However, these methods are known to have high false positives as well as false negatives, since they are based on inferring, which is intrinsically not accurate.

2.2 ARM TrustZone

Split CPU Mode Execution: The lack of a secure execution environment in mobile devices motivated the design of ARM TrustZone. Though TrustZone was designed more than 10 years ago, it is manufactured for the mass market only until recent years, due to the urgent need of a foundation for mobile security. Briefly speaking, ARM TrustZone is a security extension introduced to achieve strong isolation between a security-sensitive execution environment (secure world) and a commodity environment (normal world). This is done by dividing access permissions of the processor, memory and peripherals between the two worlds. The secure world can access all states of normal world but not vice-versa. As shown in Figure 2, there is a higher privileged mode called TrustZone secure monitor mode that is responsible for switching between the two worlds by either executing the *SMC* instruction or receiving interrupts.

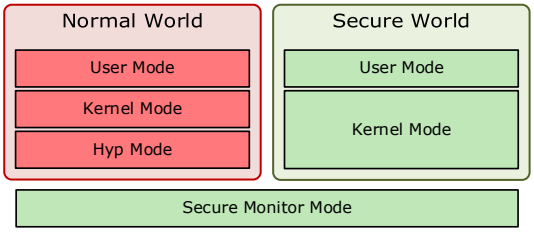

Figure 2: Split CPU mode with TrustZone support

Memory and Peripheral Protection: TrustZone supports memory partitioning between the two worlds. The DRAM could be partitioned into several memory regions by TrustZone Address Space Controller (TZASC), each of which can be configured to be used in either worlds or both. By default, secure world applications can access normal world memory but not vice-versa. System peripherals such as touch input or display controller could be configured as secure by TrustZone Protection Controller (TZPC) to ensure these peripherals could only be accessed in the secure world. Currently most of mobile phones run their OSes in the normal world, designate touch input and display controller as non-secure peripherals. By setting the touch input and display controller as secure peripherals using TZPC, we can have a secure input and display that are isolated from the normal world OS. Besides, DMA is also world-awareness and a normal world DMA that transfers data to or from secure memory will be denied.

Secure OS and Trusted Application: To leverage the benefits from ARM TrustZone, many mobile vendors nowadays, including Samsung, Apple and Huawei, deploy a secure OS (i.e., Trusted Execution Environment [2]) in the secure world, which runs in parallel with the commodity OS such as Android and iOS in the normal world in their mobile devices. They also provide some SDKs for some trustworthy third-parties to deploy their secure services called Trusted Applications (TA) running in the secure world. Typical Trusted Application in today's mobile devices are Digital Rights Management (DRM) and mobile secure payment. Note that TAs running in secure world are certified by trusted parties and unauthorized code such as code provided by ad providers will not be permitted to run in the secure world of TrustZone.

With this trend of deploying secure OS and TA on mobile devices in industry, we observe that it is practical to run an ad service as a TA to provide more secure and verifiable information about the validity of ad operations. Hence, AdAttester mainly runs as a TA in the secure world.

3. THREAT MODEL AND OVERVIEW

3.1 Goals

The main goal of AdAttester is to provide secure online ad attestation to effectively detect mobile ad frauds. Unlike prior offline-based analysis, AdAttester aims to provide a framework to enhance the mobile ad ecosystem by making mobile ad related operations attestable, even facing the challenges that the commodity mobile software stack is not trustworthy. Further, AdAttester is designed to provide verifiable attestation to detect both *bot-driven* and *interaction-driven* frauds. These two types of frauds constitute the most common ways by ad frauds nowadays and there is still no effective way to reliably detect them on-the-fly. Note that AdAttester is currently not designed with the capability to detect some app-specific policies like how many ads may be placed in a single page.

Besides providing secure online attestation to detect ad frauds, there are four extra requirements to make AdAttester practical.

- *Non-intrusiveness:* AdAttester must attest the requests on behalf of user's actions automatically without users' involvement and awareness. Thus, ad providers, developers and users have the incentives to use and deploy AdAttester.

- *Incremental deployable:* For a new ad fraud detection framework to be deployable, AdAttester should be compatible to existing ad deployment ecosystem such that existing ad delivery and accounting framework can still run under AdAttester.

- *Privacy-preserving:* AdAttester should preserve the existing properties of privacy semantics and should not leak out more privacy information than existing ad solutions.

- *Small TCB:* To make the process of ad attestation secure, the TCB of AdAttester must be small enough.

3.2 Threat Model and Assumptions

AdAttester assumes the mobile device is equipped with ARM TrustZone (TrustZone is already widely deployed in mobile devices) and the device manufacturer implements TrustZone features correctly without hardware security flaws. Further, AdAttester assumes that the device contains a persistent per-device public key pair, which constitutes a unique device identity. This identity is used to uniquely identify the ad proofs from a mobile device.

AdAttester trusts the hardware of mobile devices including the TrustZone extension and thus precludes physical attacks that may read the protected memory and tamper with TrustZone. Like other systems leveraging TrustZone, AdAttester trusts the software stack running inside the secure world. AdAttester strives to keep the TCB

of the code running in the secure world small, which makes it reasonable to trust the secure world as a whole. In contrast, the entire OS and its applications in the normal world are not trusted and are susceptible to be compromised. This is because the software stack for the normal world, which usually contains a full-fledged Linux, an application framework and/or a Java virtual machine, comprises millions of lines of code. Attackers who compromise the OS could launch various software attacks and modify any code of the OS or even the mobile ad apps. Since software running in secure world may leak some side-channel traces to attackers controlling the normal world, we will discuss how to mitigate the potential risk of leaking device private key through side-channels attacks in section 6, though there are no such kind of attacks yet.

In the current scheme, AdAttester assumes the ad provider (mobile advertiser) is trustworthy and will not be compromised by attackers. This assumption can be relaxed by requiring a secure processor or a trusted hypervisor [41, 42, 45] in the ad server (like ObliviAd [11]).

3.3 Overview

Figure 3 illustrates an overview of how AdAttester works. On the client side, when a user clicks an ad, AdAttester will generate a proof, which contains eigenvalues of the displayed ad view that the user clicks and a nonce that comes from ad server to avoid replay attack. For show-ad, instead of sending messages per click, the mobile device will collect proofs in a random interval, e.g., every five seconds, which also contains eigenvalues of ad image and nonce. The proof will be signed using the private key of the mobile device and sent to the ad server for attestation.

The ad server maintains a per-ad repository, which contains the related ad information, like eigenvalues of some image or video ads, and valid ad positions and sizes. A new module, named AdVerifier, is introduced to check the proof of each click and impression. Once receiving a message, the AdVerifier first checks its signature, nonce and proofs. If all checks pass, the AdVerifier will deliver the click message to the ad server for further processing.

To make the proof reliable, AdAttester must ensure the following conditions. First, the click is actually from user, instead of being forged. Second, the eigenvalues are actually calculated from the screen data, instead of some data not shown to users. This is guaranteed by two primitives, namely *verifiable display* and *unforgeable clicks*, which are enabled by AdAttester's non-bypassable use of TrustZone (section 4).

Figure 4 further shows the overall architecture of AdAttester. Commodity operating system (e.g., Android) runs in the normal world and a secure OS runs in the secure world. In AdAttester, the touch screen and display devices are only controlled by the secure OS to provide a reliable source of user input and display information. The trusted TZAttester, which runs in the secure world as a Trusted Application, is deployed to generate attestation blob. An untrusted AdAttester Service runs as a system service in the untrusted OS framework. The AdAttester Service cooperates with the TZAttester and provides ad libraries in applications with interfaces of getting attestation data whenever the ad libraries need to send ad requests and the application runs in the foreground. The TrustZone driver (TZ Driver in the figure) in the normal world provides APIs for AdAttester Service to call the TZAttester in the secure world.

4. AdAttester PRIMITIVES

This section first describes the necessary information to be collected for the two primitives and then describe how such information is collected securely.

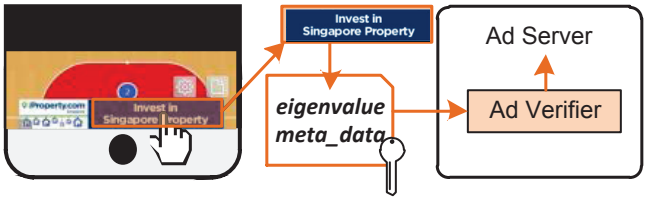

Figure 3: An overview of how AdAttester works. When a user viewing and/or clicking a mobile ad, AdAttester will generate a signed proof describing the impressions and/or user clicks. The Ad server managed by the ad provider will attest this ad operation by comparing the proof with a per-ad repository

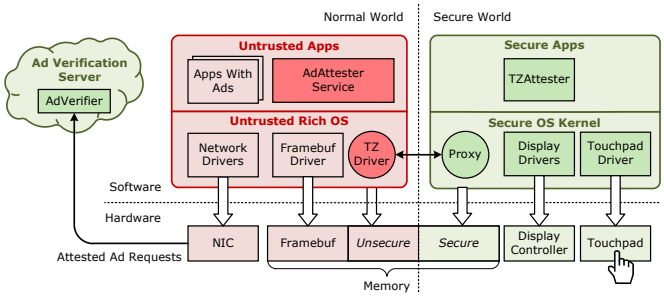

Figure 4: AdAttester architecture. AdAttester consists of three parts, two in the local mobile device and one in the remote ad server. The local mobile side consists of (1) a trusted TZAttester running in secure world and (2) an AdAttester Service running in the OS framework in normal world. The remote ad server has (3) a trusted AdVerifier that verifies the ad requests

4.1 Primitives

As the primary means for an ad provider to decide how much to pay for a developer is through counting the times of impressions and clicks, AdAttester provides two primitives, namely *verifiable display* and *unforgeable clicks*. The *verifiable display* ensures that a verifiable proof of ad impression (i.e., display) in the mobile screen and its duration will be generated and sent to the ad provider for attestation. Similarly, *unforgeable clicks* ensures that a proof is generated to attest that a click to a specific ad is initiated by a user, instead of malware such as a bot. These two primitives serve as the key building blocks to build a secure online ad attestation.

The message format of these two primitives is shown in Figure 5. A signed proof message for ad impression contains a sequence of hash values of the displayed ad, a nonce which is used to defend

Attested Message Format

impression_blob = {
displayed ad hashes, nonce, interval,
duration, ad_info, device_info
}
signed_ad_impression = {
K_{priv}{SHA(**impression_blob**)},
impression_blob, certified K_{pub}, meta_data
}

click_blob = {
clicked ad hash, nonce, ad_info, device_info
}
signed_ad_click = {
K_{priv}{SHA(**click_blob**)}, **click_blob**,
certified K_{pub}, meta_data
}

Figure 5: AdAttester primitives message format

message reuse attack (called replay attack), display timestamps describing the duration of ad display and interval of sampling, ad information including ad ID, displayed ad size and position in the screen, device information such as screen size and resolution, some other ad library-specific metadata and a certified public key. Similarly, a signed proof message for an ad click contains the hash value of the displayed ad around the click point, a nonce, ad information, device information, metadata and certified public key.

4.2 Obtaining Verifiable Ad Proofs

Generating verifiable ad proofs means that AdAttester must have a way to determine the origin of a user's click and what was displayed on the mobile screen. AdAttester achieves this by leveraging the secure OS in the secure world to control the input and display devices. This is achieved by assigning the input and display devices to be only accessible by the secure world. The secure OS provides touch input driver and secure configuration of display controller handler so that TZAttester in the secure world can get reliable information of input and display. AdAttester uses both the trusted TZAttester in the secure world and the untrusted AdAttester Service in the normal world to cooperatively collect necessary proofs for attestation. Note that the developers may control the untrusted part of AdAttester to refuse to cooperate with TZAttester, which, however, may cause a valid ad impression/click not to be attested correctly.

The first proof can be reliably obtained by collecting a user click from the secure input and signing it with a private key. A straightforward way to obtain display information is to get a snapshot of the displayed ad and send it to the ad server for attestation. This method, however, may leak user's privacy and violate our privacy-preserving requirement. To preserve user's privacy while providing verifiable display, we do not directly send the raw display data to ad server, but calculate and send the eigenvalue of the display data (i.e., data hash), which is a one-way operation and server can hardly reverse it to get the original data.

The display device driver puts the framebuffer region to the untrusted memory used by the normal world software. Hence, existing display libraries can still write display data directly to the framebuffer without trapping to the secure world, which will not greatly decrease performance. As TZAttester in the secure world could control the framebuffer, it can also get reliable display information, for example, the framebuffer region address. This is important because if the display controller is controlled by the normal world software, the normal world software could deceive TZAttester into getting a wrong framebuffer region and thus attesting on an incorrect display region.

The touch input is only accessible in the secure world, and the touch input driver is implemented in the secure world. Hence, whenever an input interrupt arrives, the touch input driver will first get the input data (i.e., input position on the screen), then the input data will be saved by TZAttester before being sent to the untrusted OS in the normal world. On getting the input data, TZAttester will check whether: (1) an ad is currently displayed; (2) the touch position locates within the ad display. If both conditions are satisfied, TZAttester will calculate the eigenvalue of the ad view region from the framebuffer. The information for checking whether an ad is currently displayed and its exact display region is provided by AdAttester Service in the untrusted OS. Note that if the information is incorrect, then the signed advertising information could be easily detected by the remote AdVerifier. The eigenvalue of the clicked ad view together with some other information such as a message nonce and device resolution, will be signed using the device private key and thus a piece of verifiable advertising information is generated. Besides, to attest ads (such as video ads) that need to attest for a period, a secure timer is used to periodically attest the display data. One possibly bypassable case is that an adversary knowing the interval can display the ad only when AdAttester is about to attest the displayed ad. The ad may be displayed short enough so that a user will not even notice it, yet from AdAttester's point of view, the ad was displayed continuously. To defend against this case, AdAttester chooses a random start time to attest with a random interval check or attest consecutively for several times in one interval period.

To retain good performance, AdAttester Service works cooperatively with TZAttester to provide necessary information so that TZAttester could decide whether an attestation proof should be generated. AdAttester Service tracks the view system of the untrusted OS and will notify TZAttester with some necessary information, including the position of the ad view whenever an ad view is displayed, hidden or resized.

4.2.1 Tracking Ad View

As a mobile ad will change it views during the execution of mobile apps, AdAttester needs to track the ad views to correctly generate proofs.

AdAttester instruments the view system of the untrusted OS to track the status of ad views. This step is necessary to reduce performance impact incurred by AdAttester; otherwise, TZAttester needs to attest the display view every time a touch input event generates. To identify ad views, ad libraries need to register ad view elements to the view system and AdAttester Service. If an ad view is shown on screen, updated or hidden from screen, the view system will invoke the notification method of AdAttester Service, which in turn will notify TZAttester in the secure world.

4.2.2 Identifying Ad Display Region

To calculate the eigenvalue of the ad view, TZAttester needs to identify and read the display content. Directly reading this data from framebuffer is feasible because the secure display driver sets the address of the framebuffer, which could not be forged by attackers in our threat model. To get the correct display data, the untrusted AdAttester Service must tell TZAttester the exact position in the display screen, otherwise an incorrect eigenvalue will be calculated, which will be detected by the remote AdVerifier.

In the hardware abstraction layer, the display device driver in secure OS allocates twice the screen size of memory and uses one as a back buffer while other as the primary surface. The drawing is done on the back buffer while the content of the primary surface is used to display the current screen content. Page flipping is used so that when the drawing is complete on the back buffer, the back buffer becomes the primary surface and the old primary surface becomes the back buffer. Since the driver is controlled by the secure world, TZAttester could get the currently primary surface and retrieve the ad content using the provided ad position information.

4.2.3 Calculating Eigenvalue

There are many algorithms to calculate eigenvalues for an image (and video). Example include SIFT [29] and SURF [12], which can tolerate complex transformations like rotation and scale down and have also recently been parallelized [15, 20]. Basically, they represent an image as a set of 64-bit or 128-bit vectors and compare their similarities by comparing the bit vectors. Originally, we used SIFT to calculate eigenvalue for attestation. However, the generated eigenvalue is sometimes too big (may even exceed the original image size), which may consume too much network bandwidth. We observed that for images in mobile ad views, it is

unlikely that an image would be rotated (though it may be scaled up/down). Hence, a scale-resistant algorithm is enough for our purpose. Hence, we use the perceptual hash algorithm [24] to reduce the eigenvalue size to 64 bit. This significantly reduce the size compared to SIFT, but can still accurately calculate the similarity of images.

5. SECURE ONLINE AD ATTESTATION

This section describes how the two primitives are used to construct the secure online ad attestation by slightly adjusting the ad libraries using APIs provided by AdAttester. Note that AdAttester is completely transparent to mobile apps.

5.1 Attesting Ads

Mobile ads have several types, including *banner ad*, *rectangle ad* and *interstitial ad*. Banner ads use a small portion of the screen to tempt users to "click through" to a richer, full-screen experience such as a website or app store page. They can have a variety of file formats, from static JPEG and GIF banners to simple text and in-banner video ads. Rectangle ads are similar to banner ads, except that they usually have a larger height than banner ads occupying a larger portion of screen. Interstitial ads, on the other hand, immediately present rich HTML5 experiences or "web apps" at natural app transition points such as launch, video pre-roll or game level load. Besides directly showing ads on screen, some ads are displayed on the notification bar to tempt users into a higher click rate.

We classify these kinds of ads into two categories: image ads (i.e., text and static image) and video ads (i.e., video media). Table 1 shows the overall APIs provided by AdAttester to ad libraries. In the following, we describe how to attest such ads using these APIs.

Adopt AdAttester to Ad SDK. Adopting AdAttester to existing ad SDKs requires minor changes on impression and click tracking. The first change is to notify the trusted TZAttester when an impression ad is shown and hidden by calling one of the APIs: *adattester_track_videoad*, *adattester_track_imagead* and *adattester_track_update*. The second change is to call the trusted TZAttester to generate attestation when a click is triggered or an impression is finished by calling one of the two APIs: *adattester_attest_impression* and *adattester_click* and add the attested blob to ad request. When an app comes to background, AdAttester Service will not accept any ad attestation requests from that app because in that case, a malicious app or ad could forge a faked ad attestation request on behalf of the foreground app, which in turn, may leak out some sensitive information such as the area a user clicks.

Ad Impression Attestation: To attest impression-based paid ads (e.g., the video ads), the *adattester_track_videoad*, *adattester_track_imagead* and *adattester_track_update* APIs are provided to keep tracking of the giving ad views. TZAttester keeps a secure timer and will calculate the signature (i.e., the eigenvalue) of the displayed ad view at a random interval. These APIs will return a track ID so that subsequent calls (namely the *adattester_track_update* that updates an ad which is being tracked, *adattester_attest_impression* that generates impression attestation and *atttester_attest_click* that generates click attestation) could find the right tracked ad. When the impression time expires and an attestation is needed, the normal world software will call function *adattester_attest_impression* and provide necessary ad information to do an attestation.

Ad Click Attestation: Click-based paid ads, on the other hand, are slightly different from impression-based ones. TZAttester tracks status of currently visible ad views with the help of the untrusted AdAttester Service in the normal world, which will in turn notify TZAttester by invoking the create, update and delete ads API. When a user click triggers ad events, the normal world AdAttester Service will call function *adattester_attest_click* to attest the user click. When the API *adattester_attest_click* is invoked, it will check whether the attested ad is currently being tracked and will put the tracked data into the attested blob to handle the situation when a user clicks an impression ad (i.e., a video ad) if so.

5.2 Ad Attestation Proofs

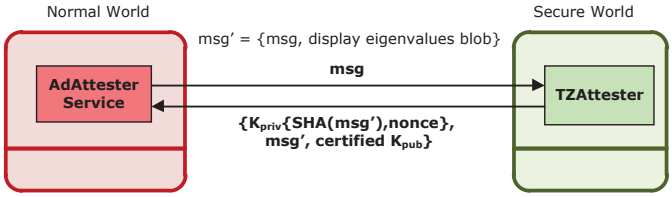

Figure 7: The attestation process of AdAttester

Device Keys. An AdAttester-enabled device contains a persistent per-device public key pair, which constitutes a unique device identity. The per-device public key pair widely exists in TrustZone-enabled mobile devices nowadays. At manufacturing time, the key pair is flashed into secure fuses (write-once persistent memory) using eFuse technology and is accessible only in the secure world. For privacy, the device generates an attestation public key pair (called attestation identity keys or AIKs), which is used to sign data inside attestation blob and will be certified by the trusted private CAs after confirming that the key is generated by TrustZone-enabled devices. Another alternative design is to use group signatures [13] to improve anonymity.

Attestation Structure. Figure 7 shows message format of the attestation proofs. An attestation message contains several parts: a hash of the attested message which is the combination of an application-specific data blob and the blob of the corresponding display data, a nonce that is used to check freshness of the request to protect against replay attacks, a certificate of the device attestation identity key. The nonce is stored in the ad server, increased and synchronized between the ad server and a mobile device when an ad session starts.

5.3 Detecting Fraud Behavior

Detecting ad frauds consists of three phases (Figure 6): a *feature extraction phase*, in which an ad preprocessing tool generates and saves an ad signature database; an *online checking phase*, which extracts and checks the legality of ad requests which may be signed; a *log analysis phase*, which uses existing advertising analytics tools to further detect frauds.

Feature Extraction. The feature extraction phase takes a set of ad images or videos as input source and splits the input source into a list of images. For each image, we calculate the eigenvalue (section 4.2.3). For a video, we generate a list of images and their corresponding eigenvalues to be used for proof.

Online Checking. When an ad request comes, the ad server will first check whether the request is attested. If the request not is attested, the legacy fraud detection and request processing modules will process that request, otherwise the AdVerifier will check the attested request. The AdVerifier will verify the certificate of the AIK to assure that the AIK comes from a TrustZone-enabled device by traversing the public-key chain enclosed in the attestation blob and use the certified public key to decrypt the nonce and the

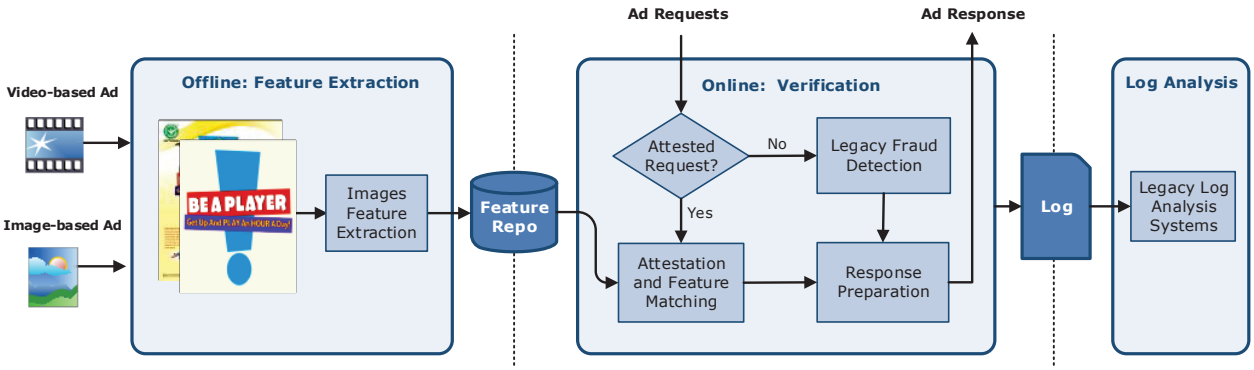

Figure 6: The ad fraud checking framework of AdAttester

Table 1: APIs provided by AdAttester to ad libraries

Command and parameters	Description
adattester_register_adview(ad_view)	Register an ad view to AdAttester Service for view tracking
adattester_create_ad(ad_info)	Notification to AdAttester Service when an ad view is created
adattester_update_ad(ad_info)	Notification to AdAttester Service when an ad view is updated
adattester_delete_ad(ad_info)	Notification to AdAttester Service when an ad view is destroyed or hidden
adattester_track_videoad(ad_info, seconds)	Tell AdAttester Service to attest a video ad for a period of time
adattester_track_imagead(ad_info, seconds)	Tell AdAttester Service to attest an image ad for a period of time
adattester_track_update(track_id, ad_info, new_seconds)	Update tracking of a tracking ad
adattester_attest_impression(track_id, ad_info, device_info, metadata)	Call AdAttester Service to attest an impression-triggered ad packet
adattester_attest_click(ad_info, device_info, metadata)	Call AdAttester Service to attest a click-triggered ad packet

payload hash value. Since the certified public key will consume a relatively large portion of ad network traffic, the ad server will maintain an ID to certified public key mapping so that client only needs to transfer its public key for the first time and uses the ID got from ad server for the rest of requests. The integrity of the message payload is checked by recalculating the hash value of the payload and comparing it with the decrypted one. The AdVerifier will retrieve the image feature data according to the ad ID from the ad request and compare the data with the ad request one. If the difference between the pre-stored image feature and the ad request one is within a predefined deviation (currently we set the maximum similarity distance value to 10 using perceptual hash algorithm), we could confirm the ad request comes from a legitimate user action from a real device. At last, the ad request will be processed by the request processing module and an ad response will be returned to the client.

Log Analysis. During the *online checking phase*, the necessary information (i.e., ad request payload and client IP) will be logged for further analysis. This phase is necessary even for the attested ad requests because some fraudsters may employ teams of people to click on ads using TrustZone-enabled devices, which could not be directly detected by AdAttester and AdAttester relies on existing fraud checking approaches to detecting this kind of frauds.

6. SECURITY ANALYSIS AND DEPLOYMENT

Attacks On Untrusted AdAttester Service: Though AdAttester relies on untrusted AdAttester Service running on the normal world to collect necessary information for proofs, AdAttester still ensures that the ad impressions and clicks cannot be forged. If AdAttester Service refuses to corporate with AdAttester, such as DOS attacks, the generated proofs will not be attested by AdVerifier. This essentially violates the sole purpose of mobile ad developers and attackers, who would try their best to avoid this.

Side-channel Attacks on Device Keys: Side-channel attacks that target on device keys is a potential threat to the security of AdAttester and even the whole TrustZone platform. Existing approaches such as Prime-and-probe [40], Flush-and-reload [43] may be also feasible to stealing the device keys, though to our best knowledge, there are no such successful attacks on TrustZone. Defensive methods against such attacks include: using lock-down cache lines for key calculation in secure world, using hardware encrypt/decrypt engines on the SoC (most of mobile devices are equipped with this feature), flushes sensitive cache on world-switch or using cache or SoC internal RAM for private key computation [21, 32].

Potential Abuses: Given the capabilities AdAttester could peek the device input and screen through TrustZone's eyes, AdAttester needs to mitigate the potentially possible abuses that could be gained by either malicious ads or applications. For screen display data, since AdAttester calculates the eigenvalue of the displayed ads and transfers the 64-bit-long eigenvalue to normal world AdAttester Service upon an attested call, this eigenvalue could be leaked to attackers. However, the algorithm for generating the eigenvalue is a one-way operation and attackers could get little valuable information from it. For input data, although AdAttester does not send the touch position outside the secure world, there are two scenarios to be discussed based on different threat model: one is that the normal world is completely compromised and the other is that only the ads or applications are malicious.

In the first scenario, because the touch input data will eventually be delivered to the normal world, attackers could easily get the input data even without AdAttester. In the second scenario, a malicious ad or app may ask TZAttester to attest non-existent ads so that after requesting for attesting ads, the malicious ad or app could know the regions where a user clicks. However, this is not feasible because AdAttester Service in the untrusted OS will keep track of ad view states and if an ad asks for tracking a non-display ad or asks for tracking when the application is not in the foreground, AdAttester Service will refuse to serve it. AdAttester may suffer from deputy or mediation attacks [14] which also exist in current mobile ads, AdAttester doesn't make it easier or leak out more privacy information than existing solutions and this kind of attacks could be prevented using ad separation (see Section 11).

Deployability and Incentives: Although AdAttester requires slight modification to existing mobile OS and ad libraries, ad providers and users as well as vendors could have the incentives to use and deploy, and the deployment effort of AdAttester for each party is small, the solution, on the other hand, could benefit the whole ecosystem of mobile advertising. (1) For ad providers, they have the incentives to deploy AdVerifier in their ad server on top of their existing ad infrastructures and adopt the APIs for attesting ads to their mobile ad SDKs which only requires several lines of code modification because it could significant prevent mobile ad frauds and thus reduce their loss. (2) Although mobile vendors are required to modify normal world OS to track ad views, the modification is largely independent and only requires adding hundreds of Java code. Besides, mobile vendors today are always optimistic to provide new features in their products, especially giving that mobile advertisement is now a major income for application developers. (3) For mobile users, they are required to install TZAttester as a trusted application in the secure OS, which is easy to deploy. By deploying TZAttester, mobile users benefit from costing network consumption due to stealthy ad requests reduction and from potential risks when a bot steals ad requests on behalf of the user. Besides, AdAttester could bring some new business scenarios where mobile users could benefit directly. For example, content providers could pay for the traffic generated by mobile users watching their ads, promotions or visiting their websites and AdAttester could provide trusted proofs to content providers.

7. IMPLEMENTATION

We have implemented the mobile side of AdAttester on a TrustZone-enabled development board called Samsung Exynos 4412, which is equipped ARM Cortex-A9 processor at the frequency of 1.4GHz. The system running in the normal world is Android Ice Cream Sandwich (Android 4.0 version), with Linux kernel version 3.0.2. The secure OS running in the secure world is T6 [9], a secure OS for ARM with TrustZone support. The trusted TZAttester runs as a Trusted Application in T6. We implemented the server side (the AdVerifier of AdAttester) on a 2-core machine equipped with Intel Xeon CPU E3-1230 V2 at 3.30GHz, which runs Ubuntu 12.04.

Device Configuration We use a development board instead of a mobile phone to implement and evaluate AdAttester as we need to get the authority from the phone manufactures to sign the TZAttester as a Trusted Application. Through the development board, we have full control of the secure world and thus can deploy our own secure OS and the TZAttester. One limitation with our board is that it does not have fused a public key pair and disallows us from flashing one into the secure fuses. We address this problem by hard-coding the public key pair in the secure OS image and envision a future in which device key is certified by trusted CAs instead of the vendors themselves.

System Boot Devices with TrustZone support will start in the secure world and run a secure world firmware after executing a vender-specific bootloader. Originally, in Samsung SoC, the secure world firmware (called *tzsw*) is proprietary and close-sourced. We replace it with a secure bootloader to control the secure world software. The secure bootloader will load the secure OS image and check the signature of the image using its embedded public key to ensure the integrity before executing T6. T6 will setup the secure environment, including partitioning the secure and non-secure memory in TZASC (reserving 16MB for the secure world used by T6 and leaving others for the normal world), setting the touch input and display controller peripheral as secure peripherals in TZPC. Note that interrupts that belongs to the touch input and display controller will be set as secure interrupt and thus will be handled by T6. After configuration, T6 will switch to the normal world to execute the normal world bootloader and boot Android.

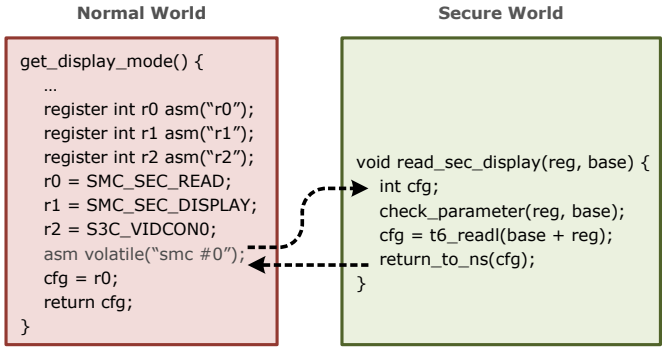

Figure 8: World switch when accessing display controller. A read or write operation on secure registers is changed to a crossing world call

Secure Input and Reliable Display Touch screen peripheral is protected from the normal world by setting the TZPC at startup and its driver is implemented in the secure world. A shared memory region between worlds is created for exchanging input data. We created a kernel workqueue, which waits on a variable and copies the input data in the shared memory region to the input buffer (using linked list) of the Linux input core (input subsystem). When the touch input driver in the secure world receives a completed sequence of input data (ABS_X, ABS_Y, ABS_PRESSURE, etc.), it will copy the data to the shared buffer and notify the normal world kernel workqueue by setting its waited variable.

Display controller is also set to be a secure peripheral in TZPC at system startup. Because the purpose of setting the display controller as secure is to get a reliable display content, which could not be faked by the normal world software. Specifically, TZAttester needs to get the exact display data from display framebuffer and if the display controller is controlled by untrusted software, TZAttester may get a faked framebuffer. To achieve this while keeping the modification of original code as small as possible, we modify the display controller driver in the normal world explicitly by changing the access of the peripheral registers to an SMC call (see Figure 8). To determine the exact address of the framebuffer, TZAttester could directly read the corresponding registers of the display controller in the secure world. The data format read from the framebuffer is RGB and TZAttester will convert it into jpeg format and use average hash algorithm [24] to calculate the eigenvalue of the displayed data.

TZAttester Execution AdAttester tracks all ad views in the Android view system layer and will notify the trusted TZAttester in secure world when an ad view is shown, resized and hidden. To omit some unnecessary crossing world notifications, changes of those ad views that are covered at least half of their size, either unintentional or on purpose (e.g., placement fraud), will not be notified to trusted TZAttester.

AdAttester Service is implemented as a privileged system service of Android that passively waits for attestation requests from ad libraries in applications. Ad libraries need to be modified to use attestation interface, but their modifications are quite small. The only change is to call AdAttester Service using the provided APIs to obtain an attestation blob when tracking ad actions and sending ad requests.

AdVerifier Implementation The AdVerifier of AdAttester locates at the ad server to process ad requests. The feature extraction phase module is a separated program implemented in Python. It takes images or videos as inputs, splits the images or videos into separated image frames and uses an image feature extraction algorithm to generate ad image features. Note that the image feature extraction algorithm is the same as the one implemented in mobile trusted TZAttester.

The online checking phase module is implemented in PHP code and we use nginx as its HTTP server. When an ad request comes, it will first check whether the request is signed. If so, it will decrypt the packet and check the validity of the nonce value. Here is the validating process on nonce value: each ad response from ad server contains a server timestamp (i.e., 2014-12-05-23-18-18), the mobile client incrementally generates the nonce value based on this timestamp. The server checks the nonce value assuring it is strictly larger than the old one it previously receives which is stored in the server but does not exceed the one based on current timestamp. If the nonce is valid, it will get the ad view display image feature from the packet, then retrieve the image feature data according to the ad ID from the ad request and compare the data with the ad request one.

8. AD FRAUD DETECTION EVALUATION

This section evaluates the effectiveness of AdAttester by the ad fraud apps from MAdFraud [16], which were crawled from several major app markets and filtered from 130,339 apps. As some of the ads in the fraudulent apps cannot be executed in our phones and we only instrument one version of each Ad SDK, we only use 182 out of 1,000 typical ad fraud apps from MAdFraud, as shown in Table 2

8.1 Dataset and Evaluation Methods

8.1.1 Ad SDK Instrumentation

Since the ad libraries need to be modified to use AdAttester, we need to instrument those ad libraries to invoke the APIs provided by AdAttester. Table 2 shows the ad SDKs we instrumented and their corresponding number of tested ad fraud apps. Admob and Doubleclick are grouped into 'Admob' as they appear together at most of time and apps that contain more than one ad SDKs are counted several times. Ad SDKs are either close-sourced or open-sourced. For the open-sourced ones, we instrument the source code by calling AdAttester Service and adding the attested blob to the ad request before the SDK wants to send ad request and then generate a jar package. Then we replace the existing ad package with our new one and repackage the apps. For the close-sourced ones, we manually decompile and analyze ad SDKs to instrument the necessary code. Here we show how we instrument the SDKs by explaining a close-sourced ad SDK named Admob and an open-sourced ad SDK named Mopub.

Table 2: Top instrumented ad SDKs. *Total*: number of fraud apps in total, *Version*: instrumented SDK version, *Tested Num*: number of tested apps

SDK Name	*Total*	*Percentage*	*Version*	*Tested Num*
Google Ads	791	79.1%	6.0.1	48
Millennialmedia	155	15.5%	4.2.6	21
Inmobi	95	9.5%	3.6.x	23
Airpush	64	6.4%	4.0.2	33
Mobfox	60	6.0%	1.4	12
Mobclix	46	4.6%	4.0.x	10
Cauly	40	4.0%	1.2.5	8
Smaato	38	3.8%	2.5.2	13
Mdotm	29	2.9%	2.0.2	3
Waps	28	2.8%	1.5.x	14
Inneractive	27	2.7%	3.1.7	9
Adlantis	26	2.6%	1.3.7	5
Domob	24	2.4%	1.5	12
Adwo	19	1.9%	2.5.1	5
Youmi	18	1.8%	3.0	10
Mopub	17	1.7%	1.9.x	5
Nend	15	1.5%	1.2.1	7
Vpon	14	1.4%	3.0.3	5
Wiyun	13	1.3%	1.2.3	2
Chartboost	10	1.0%	3.1.5	2
Guohead	8	0.8%	1.6.8	3
Applovin	8	0.8%	5.0.0	4
Appmedia	7	0.7%	1.1.0	3
Wooboo	7	0.7%	1.2	3
Casee	5	0.5%	2.7	3
Adsmogo	2	0.2%	1.2.8	1

Admob: Admob SDK is a closed-source ad library owned by Google and is now offered through Google Play services. Google uses code obfuscation techniques to protect its SDK library, which increases the difficulty of analysis tremendously. By careful instrumentations, we locate the exact location (i.e., *AdActivity.smali*) where an URL is opened as a response to the click event. We insert code there to call the system service to get attested blob and append it to the URL as parameter. Impression tracking, however, is more complex because we need to instrument several methods distributed in different files, namely internal/d.smali, internal/c.smali and internal/g.smali.

MoPub: MoPub SDK [6] is an open-source ad library owned by MoPub Inc. and is used as an ad mediation network provider. It is easy to instrument with the availability of source code and official documents. We download version v3.2 from Github and modify the following files. First, we add a function getAttesterBlob() in MoPubView to get attesterBlob by calling AdAttester Service. Then, we modify adClicked() in MoPubView to call getAttesterBlob() and store the attesterBlob value in AdConfiguration, which contains the configuration information of MoPub SDK. Finally, in registerClick(), which is located in AdViewController.java and starts a new thread to handle the click event(naming starting a new Activity, such as browser or APP market), we update the mClickThroughUrl using the attesterBlob in AdConfiguration. Thus, the attesterBlob is added to the requesting URL. The impression tracking is easy because there is an isolated file called *ImpressionTrackingManager.java* that manages the impression tracking and we just need to instrument this file.

For each ad SDK, we only instrumented one version of it. Among the instrumented ones, some ad fraud apps are useless and discarded due to at least one of the three problems: (1) they cannot display ad views in our board, probably because their fraud behavior has already been discovered by ad providers; (2) they can-

Table 3: Fraudulent behavior among the tested apps

Fraud Type	Fraud Apps
Frequent Ad Requests	154
Ads Outside Screen	23
Ads Covered/Hidden	63
Ads Too Small	9
Unaware Click	0

not run due to incompatibility in Android SDK version; (3) some apps have self-integrity check and refuses to start after being instrumented. The right two columns of Table 2 show the SDK versions we instrumented and the number of fraud apps we successfully ran.

We discarded a few ad SDKs including Madhouse, Adhub and Energysource because most of apps that embedded these SDKs also include some other ad SDKs which we have instrumented. We manually double checked that the instrumentation for each app worked correctly.

8.1.2 Online And Offline Analysis through Proxy

Since we do not have the original ad data, we set up a proxy server to redirect all ad network packets of our device so that we could get the ad data for testing the effectiveness of AdAttester. Our requirement is that this test environment should not require any modification to our ad framework AdAttester and existing ad ecosystem. To meet this requirement, we use *iptables* to configure network of target ad apps in our test device and redirect all ad network packets to our own server for analysis by creating a daemon with root privileged to execute the *iptables* scripts. The remote AdVerifier components are deployed in the proxy server. The proxy server will receive all the ad network packets and send them to the AdVerifier, which will validate ad frauds of those requests and write the result and behavior into a file for further analysis. Then it will modify the ad packets by removing the attestation blob that added by TZAttester in the mobile client and send them to the real ad server.

Each fraudulent app was run twice, each for around 90 seconds to 2 minutes. In the first run, we open the app and traverse all pages to find pages that contain ads without clicking them. In the second run, we reopen the app and click every visible ad we could see. In both runs, we use an Android tool called *hierarchyviewer* to see detail structure of hierarchy view of the app and manually find any placement frauds and/or unaware user click frauds. After the two rounds, we analyze the ad request log in our proxy server. Together with the potential placement frauds, we compare the results derived from AdVerifier with our manual analysis to determine if they were accurately detected.

We also simulated a bot-like ad fraud by writing a script in the test board to simulate clicks. Not surprisingly, as these bot-driven clicks are not signed by AdAttester and thus can by easily identified by the AdVerifier in the ad server.

8.2 Ad Fraud Findings

Fraudulent Behavior: Table 3 shows the fraudulent behavior among our tested apps. Most of the apps send ad requests frequently (at an interval of 3s to 20s), even when there are no ads visible on the screen. However, most of these frequent ad requests are not signed by TZAttester and thus will not be paid. We then carefully read the developer manual of several ad SDKs and find many of them provide an API to periodically fetch ads from ad server. We conclude that most of these frequent fetching ad requests are not intentional frauds, probably due to the carelessness of misusing the provided API or app developers want to have different ad content displayed to users to have a higher click rate. Besides the frequent ad requests, a large portion of ad requests contain impression frauds. The fraudulent behavior includes: using a floating and user-friendly element (such as an image) to cover displayed ad, putting an ad outside of the screen and a user needs to scroll up to see the ad, setting the height of an ad to zero to hide the ad from the user. These impression requests are signed by our TZAttester and considered fraudulent by our AdVerifier.

Table 4: Benchmark scores using AnTuTu

AuTuTu Test	Original Android	AdAttester Android	Overhead
Total Score	16322	16191	0.80%
2D Graphic	2280	2268	0.53%
3D Graphic	4580	4561	0.41%

Interestingly, we find some apps put several ad views from different ad SDKs to the same position and these ad libraries send impression almost simultaneously but at most one of them is treated legal by our AdVerifier. We also find several apps resize ad views to too small to be recognized (since we don't know how small a view will be considered fraudulent by ad providers, we treat any ad views smaller than 32*32 as illegal in our test). However, in our tested applications we did not find apps that have unaware click frauds and all clicks the ad AdVerifier has checked are triggered by our clicks and legally signed by TZAttester.

False Positives: Among over 600 legal ad clicks, 13 of them are considered illegal by the ad AdVerifier. Further analysis shows all the 13 clicks are banner ad clicks and happened when banner ads are changing their image content to another ad content with animation. When the content of an ad region is changing from one ad image to another in progress with animation, the eigenvalue of the ad region is quite different with all possible values we have in our database and thus is considered invalid ad request. To eliminate this kind of false positives, we suggest ad SDK providers to disable click event dispatching when a banner ad is changing its ad image with animation or more aggressively, they could disable ad animation directly during changing content to another ad image. It should be noted that all these false positives occurred only when an ad view is changing its content from one ad to another and video ads (i.e., video media and GIF image) will not have any false positive because all the possible eigenvalues of video ads are already calculated and included during offline feature extraction phase.

There is also another possible source of false positive. If a user happens to click an ad during the ad's self-animation, a wrong eigenvalue might be generated. A trivial solution is to simply disable self-animation, which is not general enough. We propose another solution called *Double-Snapshot* to detect and handle such rare cases. During offline feature extraction phase, we calculate eigenvalue of all the possible animating frame that an ad image may have and discard some eigenvalues that could violate the ad guidelines (i.e., 80% of ad image is hidden). When a user clicks the ad, TZAttester will get two snapshots: the first one on the touch-down event and the second on the touch-up event. TZAttester then detects "click in the middle of self-animation" by the two snapshots. If that is the case, TZAttester will use the eigenvalues of animation set in AdVerifier to verify the legality of the ad click.

9. PERFORMANCE EVALUATION

9.1 Ad View Tracking Overhead

We evaluate the performance impact of tracking ad views on applications. We use a popular Android benchmarking tool named AnTuTu [8], which runs a series of tests and provides a score re-

Table 5: Input latency comparison

Original Android	Android with AdAttester
66.92 ms	66.94 ms

port. We run the benchmark 10 times, each time with a reboot to eliminate impact caused by other factors (i.e., different system workload), then calculate the average score. Table 4 shows the performance data (higher score is better). Most of the impact from AdAttester is in the view system of Android graphics and other parts such as SDCard, memory access, float operation and integer operation are not affected by AdAttester. Thus, we omit those scores from the final results. From the result, it can be seen that AdAttester imposes almost no overhead on 2D graphics (0.53%) and 3D graphics (0.41%). This is because the access of display controller (read and write operation) is mostly done at system starts and will not be frequently updated in latter usage.

9.2 Input Event Handling Latency

The input latency is calculated by the execution time from the moment an input interrupt comes to the point that the input data is received by applications. Specifically, the input latency is the interval between an input interrupt which is generated by a user's touch input arrives at the interrupt handler in *ft5x0x_ts_interrupt()* and the touch input data is received by the applications through touch handler called *onTouchEvent()* in Android app. The extra input latency introduced by AdAttester includes two world switches, some ad states checking and shared data copy. World switch from secure world to normal world costs 1858 cycles and switches from normal world to secure world costs 2041 cycles. We measured the input latency of AdAttester and the latency running in the original Android for 50 times separately and calculated the averaged value. Evaluation result in Table 5 shows that there is almost no overhead incurred by putting input device driver to secure world.

9.3 Ad Network Traffic

We evaluate the increase on ad network traffic using 20 apps embedded with different ad SDKs from the top list in Google Play. First, we run these apps for 2 minutes normally without clicking ads. Second, we run them for 2 minutes and click every ad the app holds. Table 6 shows the results. The network traffic increase is less than 3.3%, which is very small compared to the original ad network traffic because the only traffic increase is from tracking impressions and clicks, which is quite small comparing to the overall traffic. During tracking ad impressions and clicks, the additional network traffic for one attested packet is never larger than 35% and mostly less than 20%.

9.4 AdVerifier Response Latency

AdVerifier processes ad attestation requests and we measured the introduced performance slowdown by simulating 100 to 1,500 concurrent user requests using Apache Benchmark [1]. The test was done in a local network environment and we used SHA1 hash algorithm to calculate the hash of message blob and 1024-bit modulus for RSA signature. The number of public keys stored in the server is 500,00. Figure 9 shows the service latency (in ms) results with and without AdVerifier. The average latency increased is 0.04 ms, which should be an acceptable overhead.

9.5 Other Evaluation

Time Cost to Generate Attestation Blob in AdAttester: Generating an attestation blob is computing-intensive and we used

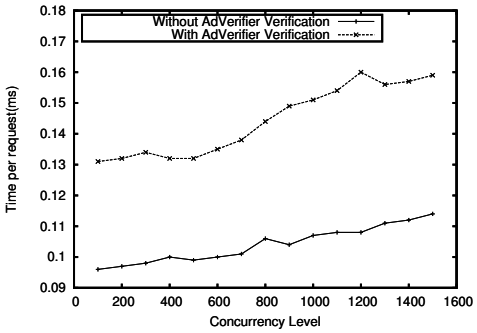

Figure 9: AdVerifier response latency compared with ad service without AdVerifier in a 2-core server. We used SHA1 for hashing and 1024-bit modulus for RSA signature

Table 7: Duration for getting an attestation blob

RSA Key Size (byte)	Duration (ms)
512	0.68
1024	3.73
2048	25.56

SHA1 hashing algorithm and RSA signing with different size of keys for generating attestation blob. The hashing time is negligible compared with RSA signing. Table 7 shows the evaluation result. AdAttester uses an asynchronous notification for generating attestation blob and ad libraries could use batching to attest ad requests if this overhead matters. The above results were got using software-only encryptions and this duration could be further reduced using hardware accelerated RSA encryption available in SoC, which will be our future improvement work.

Implementation Complexity and TCB Size: The TCB of AdAttester in the mobile client is the software running in the secure world. The secure OS, T6, contains about 6,000 LoC. TZAttester is less than 600 LoC. The secure input driver and display peripheral operation handling contain about 800 LoC. The TCB size in total is less than 7,400 LoC, which is very small. In the normal world, the implementation of tracking ad views in Android view system contains 164 source lines of java code distributed in two files and AdAttester Service has 325 source lines of Java code. In the remote AdVerifier, the feature extraction program contains 234 source lines of Python code and AdVerifier online checking contains 467 source lines of PHP code in our prototype.

10. DISCUSSIONS AND FUTURE WORK

Relaxing Threat Model of AdAttester by Trusting OS: While AdAttester could securely fight against malicious application developers and botnets, it requires the availability of per-device key, which may prevent AdAttester from being widely adopted. The main purpose of using device key is to prove that the request comes from a real mobile device which is hard to be compromised (using TrustZone) instead of a rogue device. Directly relaxing our threat model by trusting the OS and putting the secure TZAttester to normal OS could solve this problem, but comes at cost: it may become a little hard to defect frauds from botnets. This problem could be mitigated by using software-based attestation [36] [38], which will be our future work to explore.

Attest Ads to Developers: Currently, AdAttester mainly consider the case of mobile ad frauds by the developers and botnets. One interesting question is how to attest ad impressions and clicks

Table 6: Network traffic increase

App Name	Ad SDK	Total /bytes	Impression /bytes	Click /bytes	Impression Increase /bytes, percentage	Click Increase /bytes, percentage	Overall Increase /bytes, percentage
Jumper	Inmobi Mobclix Smaato	63277	3728	1991	1280, 34.3%	128, 6.4%	1408, 2.2%
English Norwegian	Google Ads	17245	1314	6071	320, 24.4%	256, 42.8%	576, 3.3%
Solitaire	Inmobi Google Ads Millennialmedia	42495	3989	4378	960, 24.1%	128, 19.7%	1088, 2.6%
Big Snake	Mopub Airpush Google Ads Millennialmedia	25398	2760	876	640, 23.2%	128, 14.3%	768, 3.0%
Despicable Me Skills	Google Ads	11957	1235	1573	160, 13.0%	128, 14.3%	288, 2.4%
Electric Bliss Wallpapers	Inneractive Google Ads Millennialmedia	49220	2357	1140	480, 13.0%	128, 14.3%	608, 1.2%
Zimane Kurdi	Google Ads	37144	2357	4024	480, 20.4%	128, 7.1%	608, 1.6%
Alien Destroyer	Google Ads Airpush	35188	2358	6343	480, 20.4%	256, 17.2%	736, 2.1%
Indian Newspapers	Inneractive Google Ads Millennialmedia	97292	7973	1137	1600, 20.1%	128, 9.4%	1728, 1.8%
Quiz des Communestet	Google Ads	59806	4829	2111	960, 19.9%	128, 11.6%	1088, 1.8%

to the developers. Though many ad providers are relatively reputed, this research question is still relevant since the ad providers may also have the incentives to deceive the developers in order to pay less. Considering mutual distrust between ad providers and developers, AdAttester could be further extended to send collected proofs (could be summary stats) to developers such that they can use the proofs to attest the ad providers if developers found significant deviation in their revenues. We leave this as our future work.

Multiple Ads on One Page: Currently, as AdAttester relies on the commodity software stack to know which ad is current on screen, AdAttester cannot detect mobile ads that violate the ad policy of the ad provider, which usually only allow one ad on one page to minimize impact on users' experiences. Fortunately, this may still be distinguished by the ad providers though statistical analysis, which we will study in our future work.

11. RELATED WORK

AdAttester is motivated by the urgent need of a verifiable approach for ad impression and clicks; it differs from prior work in that it is the first to provide verifiable mobile ad attestation to reliably detect mobile ad frauds online.

Mobile Ad Separation There are several systems focusing on restricting the privilege of advertising code in the mobile client. AdSplit [37] puts the advertisement into a separate activity and passes all requests from the activity onto a newly introduced advertisement service. AdDroid [34] encapsulates the ad libraries into Android framework and introduces a new advertising API to allow applications to show ads without requesting privacy-sensitive permissions. AFrame [46] isolates ad libraries by covering not only the process and permission isolation, but also the display and input isolation.

Ad Privacy Preserving. Another line of research on advertisement is to protect users' private information. Privad [22] provides a faster and more private advertising framework by using a client to locally serve ad and introducing a dealer server to separate ad requests in web advertising. Adnostic [39] proposes client-side software that provides target ads to a user without compromising user privacy. ObliviAd [11] uses secure hardware (i.e., secure processor) in the ad server for private information retrieval aiming at providing two privacy goals: profile privacy and profile unlinkability. These systems are complementary to AdAttester and may further improve the privacy of AdAttester.

Web Advertising Fraud. Ad fraud has been studied extensively for years in online web advertising. Daswani [17] gives an overview on techniques of web ad frauds; Miller et al. [31] summarize techniques and innovations of today's clickbots. Prior work on detecting bot-driven click frauds mainly analyzes query logs [44] in search engine to aggregate ad traffic across IP addresses [30],

or through complex analysis from peer-to-peer measurements and command-and-control telemetry [33]. NAB [23] uses TPM to attest user actions by analyzing mouse and keyboard activity to identify and certify human-generated activity, thus filter out bot-driven clicks and spam. However, NAB doesn't address the problem of ad verification and the approach they took to deal with bot-driven clicks isn't feasible for mobile devices.

Mobile Advertising Fraud. Though ad frauds have been relatively well-studied in desktop environments, there are very few research studies on mobile ad frauds. Existing approaches mainly focus on offline testing to detect ad frauds. DECAF [26] characterizes ad fraud in mobile apps and proposes offline techniques to detect display fraud on Windows-based mobile platforms by using automated testing. The techniques they used are analyzing the status of ad UI offline to determine whether ads are hidden, obfuscated or stacked. MAdFraud [16] studies mobile ad fraud perpetrated by Android apps and identifies two kinds of fraudulent behavior: requesting ads in the background and clicking on ads without user interaction. They further developed an analysis tool to automatically trigger and expose ad fraud in Android emulators. However, the intrinsic limitation of offline testing on coverage and the lack of a reliable way to distinguish benign from fraud ads make it hard for such approaches to detecting sophisticated means of doing frauds, especially bot-driven frauds. In contrast, the verifiable nature of AdAttester makes it cryptographically reliable to identify ad frauds.

Attestation Using TrustZone. There are several systems that use TrustZone to provide attested data or actions. VeriUI [27] runs a Linux in TrustZone secure world to provide an attested login for users, which demonstrates that a responsive UI could be provided with a small attack surface. The attested login augments user credentials with a certificate describing the software and hardware that handled the credentials. TrustedSensors [28] uses TrustZone to attest sensors data and provides two sensor abstractions, namely sensor attestation and sensor seal. SIMlets [35] splits the bill for mobile data using TrustZone and allows content providers to pay for the traffic generated by mobile users visiting their websites or using their services. TrustUI [25] uses TrustZone to provde trusted paths between mobile user and mobile device as well as between mobile device and remote service without trusting the device drivers.

12. CONCLUSION

This paper argued that the lack of verifiable ad attestation proofs is a key obstacle to defend against mobile ad frauds. To this end, this paper described AdAttester, a system that effectively detects and prevents well-known ad frauds. AdAttester is enabled by two primitives, namely *verifiable display* and *unforgeable clicks*, which securely attests to the ad sever whether an impression or a click is actually delivered to or conducted by a real user. The two primitives have been implemented using ARM TrustZone to exclude the commodity software stack out of the TCB. Based on these two primitives, AdAttester successfully checked the legality of every ad click and impression request by determining whether the ad request violates predefined rules. We have implemented AdAttester on a Samsung Exynos 4412 board, which runs Android as the mobile operating systems. Evaluations with a set of mobile ad frauds that use all well-known ad SDKs confirmed that AdAttester can reliably detect ad frauds, while incurring small performance overhead and little impact on user experience.

Acknowledgments

We thank our shepherd Landon Cox and the anonymous reviewers for their insightful comments, Prof. Hao Chen for making their mobile ad fraud dataset available to us, our group member Liang Liang for helping us evaluate the performance impact of AdAttester. This work is supported in part by a research grant from Huawei Technologies, Inc., the National Natural Science Foundation of China (No. 61303011), the Program for New Century Excellent Talents in University, Ministry of Education of China (No. ZXZY037003), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (No. TS0220103006), the Shanghai Science and Technology Development Fund for high-tech achievement translation (No. 14511100902), a research grant from Intel and the Singapore NRF (CREATE E2S2).

13. REFERENCES

[1] Apache http server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] Trusted execution environment of globalplatform. `http://www.globalplatform.org/specificationsdevice.asp`.

[3] Bots are hot, but publishers and advertisers are cold to combating the situation. `http://www.adexchanger.com/online-advertising/bots-are-hot-but-publishers-and-advertisers-are-cold-to-combating-the-situation/`, 2013.

[4] Bots mobilize. `http://www.dmnews.com/bots-mobilize/article/291566/`, 2013.

[5] Bots win! nonhuman ad impressions still selling like hotcakes. `http://www.adexchanger.com/online-advertising/bots-win-non-human-ads-still-selling-like-hotcakes/`, 2013.

[6] Mopub android sdk. `https://github.com/mopub/mopub-android-sdk`, 2013.

[7] Admob publisher guidelines and policies. `https://support.google.com/admob/answer/2753860?hl=en`, 2014.

[8] Antutu benchmark. `https://play.google.com/store/apps/details?id=com.google.android.stardroid&hl=en`, 2014.

[9] T6, a secure os and tee for mobile devices. `http://trustkernel.org/`, 2015.

[10] T. Alves and D. Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4), 2004.

[11] M. Backes, A. Kate, M. Maffei, and K. Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 257–271. IEEE, 2012.

[12] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In *Computer Vision–ECCV 2006*, pages 404–417. Springer, 2006.

[13] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145. ACM, 2004.

[14] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.

[15] P. Chen, D. Yang, W. Zhang, Y. Li, B. Zang, and H. Chen. Adaptive pipeline parallelism for image feature extraction algorithms. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 299–308. IEEE, 2012.

[16] J. Crussell, R. Stevens, and H. Chen. Madfraud: investigating ad fraud in android applications. In *Proceedings of the 12th*

annual international conference on Mobile systems, applications, and services, pages 123–134. ACM, 2014.

[17] N. Daswani, C. Mysen, V. Rao, S. Weis, K. Gharachorloo, and S. Ghosemajumder. Online advertising fraud. *Crimeware: understanding new attacks and defenses*, 2008.

[18] V. Dave, S. Guha, and Y. Zhang. Measuring and fingerprinting click-spam in ad networks. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 175–186. ACM, 2012.

[19] emakerter. Driven by facebook and google, mobile ad market soars 105http://www.emarketer.com/Article/Driven-by-Facebook-Google-Mobile-Ad-Market-Soars-10537-2013/1010690, 2014.

[20] Z. Fang, D. Yang, W. Zhang, H. Chen, and B. Zang. A comprehensive analysis and parallelization of an image retrieval algorithm. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 154–164. IEEE, 2011.

[21] L. Guan, J. Lin, B. Luo, and J. Jing. Copker: Computing with private keys without ram. 2014.

[22] S. Guha, B. Cheng, and P. Francis. Privad: Practical privacy in online advertising. In *NSDI*, 2011.

[23] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-bot (nab): Improving service availability in the face of botnet attacks. 2009.

[24] N. Krawetz. Perceptual hash algorithm: the average hash algorithm. http://www.hackerfactor.com/blog/?/archives/432-Looks-Like-It.html.

[25] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 8. ACM, 2014.

[26] B. Liu, S. Nath, R. Govindan, and J. Liu. Decaf: detecting and characterizing ad fraud in mobile apps. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 57–70. USENIX Association, 2014.

[27] D. Liu and L. P. Cox. Veriui: Attested login for mobile devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, page 7. ACM, 2014.

[28] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 365–378. ACM, 2012.

[29] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[30] A. Metwally, D. Agrawal, and A. El Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. In *Proceedings of the 16th international conference on World Wide Web*, pages 241–250. ACM, 2007.

[31] B. Miller, P. Pearce, C. Grier, C. Kreibich, and V. Paxson. Whatąŕs clicking what? techniques and innovations of todayąŕs clickbots. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 164–183. Springer, 2011.

[32] T. Müller, F. C. Freiling, and A. Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, pages 17–17, 2011.

[33] P. Pearce, V. Dave, C. Grier, K. Levchenko, S. Guha, D. McCoy, V. Paxson, S. Savage, and G. M. Voelker. Characterizing large-scale click fraud in zeroaccess. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 141–152, New York, NY, USA, 2014. ACM.

[34] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72. ACM, 2012.

[35] H. Raj, S. Saroiu, A. Wolman, and J. Padhye. Splitting the bill for mobile data with simlets. In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, page 1. ACM, 2013.

[36] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272–282. IEEE, 2004.

[37] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*, pages 553–567, 2012.

[38] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 249–264. ACM, 2011.

[39] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *NDSS*, 2010.

[40] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[41] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 430–444. IEEE, 2013.

[42] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 246–257. IEEE, 2013.

[43] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, Aug. 2014. USENIX Association.

[44] F. Yu, Y. Xie, and Q. Ke. Sbotminer: large scale search bot detection. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 421–430. ACM, 2010.

[45] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, 2011.

[46] X. Zhang, A. Ahlawat, and W. Du. Aframe: isolating advertisements from mobile applications in android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 9–18. ACM, 2013.