Fraus: Launching Cost-efficient and Scalable Mobile Click Fraud Has Never Been So Easy

Elliott Wen
The University of Auckland
jq.elliott.wen@gmail.com

Jiannong Cao, Jiaxing Shen, Xuefeng Liu
The Hong Kong Polytechnic University
csjcao, jxshen@comp.polyu.edu.hk

Abstract—**Mobile click fraud is a type of attack where an adversary deceptively generates click events on mobile applications in pursuit of revenue. Conventionally, the attack is carried out by automating a massive number of physical devices. However, purchasing the devices incur substantial costs. A cheaper alternative to the physical devices is emulators. However, existing emulators are inefficient and vastly blocked due to their immense resource demand and defective device signatures. In this paper, we propose Fraus[1], a cost-efficient and scalable approach to conduct large-scale click fraud using device emulators. Fraus maintains a low resource profile by circumventing graphics emulation and applying lazy-loading techniques on system components. Besides, Fraus provides a seemingly authentic device signature and disguises itself as a legitimate device by fully emulating the missing hardware components including WiFi interfaces and cellular modems. To facilitate the management of numerous emulator instances, Fraus also offers a distributed management system, which is scalable and fault-tolerant. We evaluate the performance of Fraus by mocking attacks against the top 300 applications from the Google Play store. The results demonstrate that Fraus has high system stability and application compatibility. It also significantly reduces CPU usage and memory footprint up to 90% and 60% respectively compared with the existing emulators.**

I. Introduction

Nowadays, Mobile Click Fraud Attack (MCFA) has become a frequent topic of cyber security experts [1]. In such an attack, malicious individuals repeatedly generate click events on a mobile application with the intention of increasing revenues or personal influence. The common examples include boosting product ratings or increasing the 'like' number in social media pages. According to [2], the attack is causing a substantial damage of $16.7 billion on mobile application economy in 2017.

To launch a MCFA, a simple approach referred to as Click Farms [3] is widely adopted. Figure 1 demonstrates the general set-up of such a farm where thousands of mobile devices are deployed and automated by computers to generate click events. Since the click actions are originated from the physical devices, they all appear to be triggered by real users without sophisticated examinations [4]. It makes this approach fairly effective for most applications.

However, despite the effectiveness, setting up such a farm incurs a substantial cost. The major expenses is for purchasing the massive number of devices. Meanwhile, the subsequent deployment and maintenance tasks also involve huge labor costs. These drawbacks naturally inspire us to investigate the potential of replacing the physical devices with device emulators. Device emulators are in essence virtual machines that are mainly designed for developers to swiftly prototype mobile applications without using hardware devices. Compared with physical devices, emulators are significantly more cost-effective because they can be massively deployed in cloud platforms or local servers in a cheap price. For instance, the average price for a low-end smartphone with one-year lifespan is approximately 215 USD [6], while the money can enable us to host 6 emulator instances without optimization in the cloud for a same timespan [7]. Besides, deploying and maintaining emulators is less labor intensive owning to existing automated management systems [8].

Though the idea of launching click fraud with emulators seems quite straightforward, practical implementation entails substantial challenges. First, existing device emulators are CPU-intensive and memory-intensive. It likely results in poor performance and application failure due to resource constraints. Moreover, the high resource demand also decreases the number of emulator instances that can be run concurrently, which limits the power of the attack. Second, the existing emulators are easily blocked by various mobile applications due to their defective device profiles. A profile serves as a unique device signature, usually consisting of properties such as WiFi MAC addresses or phone numbers that are hardware-specific. Since the corresponding hardware components are not emulated, the property values remain empty. Therefore, by scrutinizing the defective device profile, an application can easily identify and block the emulator. Finally, it is challenging to deploy and manage numerous emulator instances in a large-scale attack, because existing emulator management systems are centralized and they fail to cope with the scale-out and fault tolerance issues.

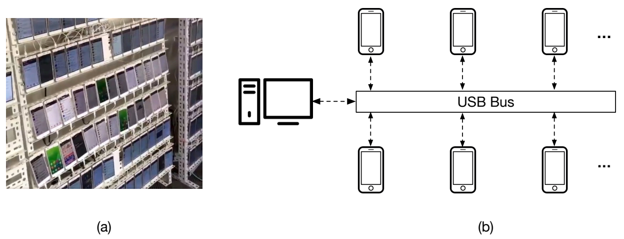

Fig. 1: (a) A photo inside a real-life click farm [5]. (b) The general set up of a click farm, where phones are powered up and connected to a central server via USB connections.

[1]In Roman mythology, Fraus was the goddess or personification of treachery and fraud.

In this paper, we propose Fraus, a novel approach that enables individuals to launch MCFA using device emulators. Specifically, Fraus significantly reduces CPU usage by circumventing the compute-intensive graphics emulation. It also maintains a small memory footprint by applying the lazy-loading and deduplication techniques on system components. To bypass device-profile scrutiny, Fraus implements two virtual hardware components including cellar modems and WiFi interfaces. They enable each emulator instance to generate a seemingly authentic device profile. Meanwhile, Fraus offers a distributed emulator cluster management system, which is scalable, fault-tolerant and easy-to-use. Its API abstraction features a logically centralized view, which allows an user to easily program and launch an attack task without concerning the details of the underlying distributed protocols.

We consolidate the above techniques and implement the prototype of Fraus on top of Android 6.0. We conduct comprehensive experiments to evaluate Fraus against the top 300 applications from the Google Play store. Experimental results show that Fraus has high application compatibility and system stability because 96% of the selected applications run successfully, while the number is only 59% for existing emulators. Fraus also reduces CPU usage and memory footprint up to 90% and 60% respectively compared with the existing emulators. The small resource demand enables fitting 125 instances in a low-end server equipped with 32GB RAM, reflecting the high cost efficiency.

To the best of our knowledge, Fraus is the first system that is designed to launch a cost-efficient and scalable MCFA via devices emulators. By designing Fraus, we aim to raise public concerns about the simplicity of committing click fraud and to suggest countermeasures to mitigate such risks.

The rest of this paper is organized as follows. Section. II investigates the problems of launching MCFA using existing emulators. Section. III presents the approach to disable the compute-intensive graphics emulation. Section. IV describes optimization techniques that reduce the memory footprint. Section. V illustrates the methodology to bypass device profile scrutiny. Section. VI elaborates the architecture for the distributed emulator instance management system. Section. VII reports the evaluation results. Section. VIII discusses the countermeasures to combat Fraus. Section. IX surveys the related work and Section. X concludes this paper.

II. Background and Motivation

Little research has been conducted to investigate the usability of emulators on MCFA. In this section, we attempt to evaluate the usability and identify the potential problems pending to be addressed via preliminary experiments. We first select 4 target mobile applications from two genres that are frequent victims of MCFA, including social media and online advertising. The selected social media applications consist of *Facebook* and *Wechat*. The selected online advertising applications, which reward users for every view of a commercial video clip, include *Daily Cash* and *Lucky Cash*. We run these applications on three widely-used options including *Android Official Emulator* [9], *BlueStacks* [10] and *GenyMotion* [11]. These emulators are allocated with an one-core 2.2 GHz virtual CPU, 512 MB RAM, and 12 MB video RAM with GPU acceleration disabled. This resource configuration is chosen to be close to the specification of a low-end virtual machine provided by most cloud platforms [7]. In our mocking attack, we identify the following difficulties.

TABLE I: System Wide CPU Usage When Running Selected Applications.

CPU Usage	Offical	BlueStack	GenyMotion
Wechat	43%	47%	45%
Facebook	57%	54%	59%
Daily Cash	79%	76%	78%
Lucky Cash	87%	83%	84%

TABLE II: CPU utilization of each running process in an emulator.

CPU Usage	Surfaceflinger	Mediaserver	Systemservice	App. itself	Others
Wechat	25%	0%	3%	11%	4%
Facebook	37%	2%	4%	13%	1%
Daily Cash	11%	49%	7%	7%	5%
Lucky Cash	23%	43%	5%	10%	6%

Performance Issues. The first difficulty arises from the performance issues of emulators. We observe that the emulators tend to be CPU-intensive. To demonstrate our observation, we measure the average system-wide CPU usage when the emulators are automated to perform predefined tasks such as clicking a like button or viewing a video clip in the 4 applications. The results are shown in Table I. It can be seen that all the applications are incurring high CPU workload regardless of the types of emulators. Specifically, the social media applications consume almost half of the CPU processing power. The advertising applications appear to be more compute-intensive and nearly saturate the CPU.

To pinpoint the root cause of high system-wide CPU utilization, we further demonstrate the usage of each running process in an emulator in Table II. Note that we only list the CPU statistics for the Android official emulator because different emulators yield similar CPU performance. One essential finding from the table is that although the overall CPU usage is high, only a small proportion is attributed to the application itself. Two system processes *Surfaceflinger* and *Mediaserver* are the main culprits of the excessive CPU usage. Surfaceflinger is an Android system service, in charge of compositing graphical user interfaces (GUI) of applications into a single buffer, which is finally displayed by a screen. Mediaserver is another important system service that entitles an application to play various types of video files. The two system services involve a great amount of graphical computation, which can be offloaded to GPUs in physical devices. However, GPUs are not available in an emulator and the graphical computation has to be conducted by CPUs inefficiently, which justifies the high CPU workload of the Surfaceflinger and Mediaserver.

Another performance concern is the excessive memory usage. The minimum memory requirement of BlueStack and GenyMotion is 2 GB. It means that the only way to run them in the virtual machine with 512 MB RAM is to enable swapping. However, swapping may negatively impact the system performance due to the potential thrashing behavior. The Android official emulator somehow possesses a smaller footprint, which is approximately 500 MBs right after the system starts up. Nevertheless, the free RAM is extremely limited and swapping is still required in order to run any other applications.

Defective Device Profile. We meet the second obstacle when running applications that enforce device profile scrutiny. A device profile serves as a unique device identification. It usually consists of several hardware-specific properties. Commonly used properties include:

1) WiFi Media Access Control (MAC) Address, a unique identifier assigned to WiFi interfaces.
2) The cell phone number of the device.
3) International Mobile Subscriber Identity (IMSI), a unique number to identify the user of a cellular network.
4) International Mobile Equipment Identity (IMEI), a unique 15-digit serial number given to every mobile phone's cellular modem.

All the properties are retrieved from hardware components including cellular modems and WiFi interfaces. Since they are not emulated, the above property values remain empty in an emulator. The artifact then allows application providers to easily identify and block the emulators. To make matters worse, the authenticity of the properties, especially the phone number, can be easily verified by the applications. For instance, many applications send the number a SMS containing verification codes to ensure the number is genuine. It makes randomly generating the property values an infeasible approach to bypass the scrutiny.

Deployment and Management. We encounter another challenging issue when launching a large-scale MCFA. In such an attack, a massive number of emulator instances have to be deployed and managed. To avoid this labor-tensive procedure, we adopt a centralized cluster management system [8] in our preliminary study. However, we discover that the centralized architecture quickly fails to cope with the scale-out and fault tolerance issues when the number of emulators dramatically increases. To address this issue, we need to design a distributed management system. Meanwhile, to preserve the simplicity of the centralized management system, we aim to design a high-level API abstraction presenting a logically centralized view, which can conceal the complicated details of the underlying distributed protocols.

In light of the above issues, we design Fraus, a system that is lightweight and able to bypass the device profile scrutiny. Meanwhile, Fraus's distributed emulator management system enables us to launch a large-scale MCFA in an effortless manner. In the following sections, we will elaborate Fraus's solutions towards the above issues one by one.

III. Preserving CPU Resources

As stated in the previous section, due to the lack of GPUs in an emulator, graphical computation such as GUI composition and video playback has to be conducted by CPUs inefficiently, which results in a huge waste of CPU resources. In this section, we elaborate our approaches to preserve the CPU resources for an emulator. The main intuition is that we can circumvent the graphical computation in Android because attackers most of the time do not care about the graphical output of the emulators, which are running automation scripts to generate click events.

A. Overview of the Android Graphics System

To make our approaches better understood, we first introduce the architecture of the Android graphics system. As

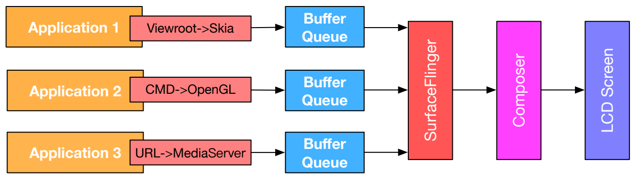

Fig. 2: Android Graphics Overview

shown in Fig 2, an Android application can draw its contents on the device screen via three different routines including *Skia*, *OpenGL* and *Media Server*.

Skia. The most commonly-used routine relies on the *Skia* library to performs 2D graphics drawing. Specifically, an application provides the Skia with its GUI layout hierarchy, which organizes all the GUI components (such as texts, buttons and images) in a tree structure. The tree's root is referred to as *ViewRoot*. Beginning with the ViewRoot, drawing is performed by walking the tree and rendering all child components by invoking their *draw()* method. The method draws the GUI component's appearance on top of a graphics buffer. The resulting buffer then can be inserted into a buffer queue and wait for the further processing from the underlying system service.

OpenGL. An application can perform 3D graphics drawing with the help of the *OpenGL ES* library. OpenGL ES is a cross-platform graphics API set that specifies a standard interface for GPUs. By invoking the APIs, applications could deliver 3D drawing commands to GPUs. Once GPUs finish the tasks, the resulting buffers will be also inserted to a buffer queue. Note that emulators possess no GPUs, thus they have to be emulated via software.

Media Server. An application can utilize a system service named Mediaserver to display video contents on the screen. Specifically, the media server accepts a media file and processes it by using built-in media decoders. The decoded buffers then will be sent to the buffer queue. Note that the video decoding is considered to be a fairly CPU-intensive procedure, a physical device therefore usually offloads it to the a dedicated hardware decoder that does not exist in an emulator.

All the routines places the resulting buffers to the buffer queues. The queues are managed by a system service named Surfaceflinger. Specifically, Surfaceflinger will dequeue the buffers from all the queues and forward them to the underlying composer. The composer then composites all the buffers and generates a final picture on the screen. The process is carried out periodically and the frequency is decided by the LCD screen hardware. In brief, the hardware will indicate the Surfaceflinger to start the process by generating a VSYNC signal. In an emulator, the VSYNC signal is generated from a software timer every 1/60 second.

B. Naive Approach: Suppressing VSYNC Signal

A simple solution to skip the graphical computation is to suppress the VSYNC signal by adjusting the software timer. Without the VSYNC signal, the Surfaceflinger would stop dequeuing buffers from the buffer queues. Then the number of buffers in the queues would start accumulating. Once the number reaches a limit, which is three by default, the Android applications would not be able to enqueue more graphics buffers and have to stall the drawing process. In order words, no more draw() methods would be invoked and the graphical computation would be circumvented.

Despite the simplicity of the approach, it may affect the stability of certain applications that contain custom GUI components. Custom components entitle developers to create novel user interfaces for special needs. To implement such a component, developers have to write their own draw() method and manipulate the graphics buffers to sketch the component's appearance. Besides, auxiliary functionalities such as performance tuning and logging may also be implemented in the draw() method. Since the draw() method is suppressed by our approach, the auxiliary functionalities would never get the chance to be executed, which may potentially crash the applications.

C. Sophisticated Approach: Submitting Blank Buffers

Clearly, the main drawback of the naive approach is that it may interfere with the application execution and the final outcome varies from application to application. Motivated by this issue, we propose a more sophisticated approach that is more reliable and fully application-independent. The main idea is that we alter the behavior of the three graphics routines; they now simply submit blank buffers to the buffer queues, when applications request them to draw any contents. This approach not only circumvents the intensive graphical computation, but also poses no interference to all applications. From the applications' viewpoint, they can invoke the draw() method periodically as they are running in a normal Android system.

We implement this approach by adopting a technique named Dynamic Linker Hooking [12]. Specifically, hooking is the process of intercepting a program's execution at a specific point, typically entries of functions, in order to alter or augment the program's behavior. The dynamic linker hooking technique enables hooking in the runtime by forcing a program to load shared libraries specified by the user instead of the original ones provided by operating systems. This technique allows us to avoid recompiling the source codes of Android and to quickly apply the following patches to the three graphics routines.

Skia. We patch all the low-level Skia APIs to avoid the drawing procedures. Specifically, Android internally draws each GUI components by invoking a series of low-level Skia APIs such as *SkBitmapDevice::drawPaint*, *SkBitmapDevice::drawPoints*, and *SkBitmapDevice::drawRect*. The APIs originally conduct compute-intensive linear algebra computation to place correct shapes on a graphics buffer. After the patching, these APIs become stub functions which simply return a result indicating the rendering operation has been successful.

OpenGL ES. The stock OpenGL ES library only supports a very basic set of OpenGL commands and it is barely compatible with modern Android applications [13]. To address this issue, we first replace the stock OpenGL ES library with a fully-fledge open-source one [14]. Two kinds of patches are then applied on the library. First, we rewrite all the graphics APIs whose names start with *glDraw*. The modified functions skip the complicated 3D geometry rendering and simply report a result indicating operations are successful. Second, we also patch the *memcpy* function used in the graphics APIs including *glBufferData* and *glCopyTexImage2D*. The two APIs are originally designed to copy bulky texture data from the user process to the GPU emulator. Realizing the excessive data transfer may negatively degrade system performance, we replace the *memcpy* function with a no-operation (NOP) to skip the data copying.

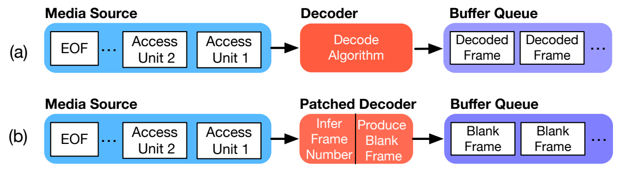

Fig. 3: (a) The decoding process of the media server. (b) The patched decoder avoiding the decoding computation.

Media Server. We patch the commonly-used media decoders including *mpeg2dec*, *avcdec* and *aacdec* in the media server. As shown in Fig. 3 (a), the origin decoders repeatedly request a media access unit from the source. The unit then can be decoded into a displayable frame or a few milliseconds audio clip, which will be sent to a buffer queue. The playback is finished if the decoders successfully detect the end of stream (EOF) in the source.

To avoid decoding computation, we let the decoders dump the access units received and directly send frames filled with zero to the buffer queue. However, this method has a limitation; without running the decoding algorithm, the decoder now cannot determine how many audio frames exist in an audio access unit. If the decoder cannot generate a correct number of audio frames, the duration of the playback may be either shortened or prolonged. It may affect the stability of certain applications, for instance, advertisement applications which rely on the duration of playback to count a view.

To address this issue, we propose an alternative solution as shown in Fig. 3 (b). The patched decoder will infer the number of frames needed to be generated by examining the timestamp property existed in access units. Specifically, every access unit carries a timestamp indicating the time at which its first decoded frame is played. The timestamp difference between the current and consecutive access units then can be used to calculate the available frames given the audio sampling rate.

IV. Reduce Memory Footprint

In this section, we demonstrate how to optimize the memory footprint of an emulator. The memory-exhausted problem comes from that Android loads nearly all system components to the memory during the booting process, even most of them are not used by applications. In Fraus, we defer the loading of those components until when they are needed.

We first measure the memory usage in the official emulator running Android 6.0. Note that the measurement is conducted right after the booting process and the emulator has been restored its factory settings. The memory statistics shows that the total memory usage is approximately 500 MB, specifically, 290 MB of which is allocated to built-in applications (e.g., calendar, email, and music) while the remaining is consumed by a set of system services (e.g., Systemserver, Surfaceflinger and Mediaserver).

It can be seen from the statistics that a huge memory overhead lies in the built-in applications, which automatically start along with the system. This behavior is usually referred to as preloading, which helps improve user experience by shortening the applications' loading time. Since these built-in applications are seldom used in a MCFA, preloading them is simply a waste of memory. To reclaim the memory, we patch the applications to disable preloading. Specifically, an application can be preloaded by either registering a listener for the *boot_completed* event or setting the *android:persistent* property in its manifest file. Thus our patches remove all the listeners for the boot event and the android:persistent property if existed.

Apart from the built-in applications, we discover that another major memory consumer are the system services. Android possesses approximately 70 Android system services that provide applications with the information and capabilities necessary to work. Typically, all the services are instantiated by Android during the booting process. However, a great number of them may never be used by an application, which results in the waste of the memory space.

To reduce the memory footprint, one naive approach is to directly disable certain system services which are unlikely to be used. Clearly, this method may impede the system functionality and stability, because applications are likely to crash when they attempt to communicate with the disabled services. Instead, we propose an alternative approach making use of the lazy-loading and deduplication techniques. Specifically, we first divide all the system services into three categories including *futile occasionally-used*, and *critical* based on the necessity of their functionalities. Then we applied different approaches to reclaim the memory.

Futile Services The futile services such as *Battery Service* and *Vibrator Services* are highly unlikely to be used in an emulator due to the lack of the corresponding hardware components like batteries and vibration motors. However, these services still occupy a certain amount of RAM space due to the memory allocation of internal data structures. To reduce the services' memory overhead, we replace them with stub services. Stub services expose identical API interfaces as the original ones do, but the implementation of the APIs just serves as a placeholder and no memory is allocated. Applications that invoke the APIs will simply receive a result indicating the corresponding operations are successful.

Occasionally-used Services The occasionally-used services such as *MediaRouter* and *TextService* still provide infrequently-used functionalities to a small number of applications. To reduce their memory overhead, we propose a lazy loading mechanism. It allows these services to postpone the instantiation to when they receive the first request from applications. To implement this mechanism, we first design a set of proxy objects of those services. A proxy object implements same API interfaces as the original services do and forwards the API requests to the corresponding services if they already instantiate. Otherwise, the proxy then is in charge of instantiating the services. Meanwhile, we patch a system process named *SystemServer* such that Android loads the proxy objects instead of the real services to the memory during the booting process.

Critical Services The critical services such as *Activity Manager* and *Window Manager* provide the basic system functionalities to every application and thus should remain intact. Nevertheless, realizing these critical services are identical and run in every emulator instance, we can easily infer that in the physical server hosting the emulators, there exists enormous duplication in memory. To reduce the memory usage, we adopt a deduplication technique named Kernel Same-page Merging (KSM). It merges identical memory pages from multiple emulator instances into one memory region, which significantly reduces the memory consumption and increases the number of instances that can be run concurrently.

V. Bypassing Device Profile Scrutiny

As stated in Section II, current emulators are not likely to pass device profile scrutiny, because their device profiles are missing properties including 1) WiFi MAC addresses, 2) IMEI, 3) IMSI, and 4) phone numbers. The main cause of this problem is that the emulators possess no WiFi interface and cellular modem, from which the above properties are retrieved. In this section, we demonstrate the approach that bypasses the device profile scrutiny by emulating the missing hardware components. The components help provide a complete device profile and disguise the emulator as a legitimate device.

A. *Integrating Simulated WiFi Interfaces*

We first focus on the WiFi MAC address property. A MAC address is an identifier assigned to WiFi interfaces for communications at the data link layer. In a physical device, the MAC address can be accessed by an application using the APIs from the *WiFi* service, for instance, *WifiInfo.getMacAddress()* or *NetworkInterface.getNetworkInterfaces()*. However, in an emulator these methods simply return an empty value or raise an exception complaining of the missing WiFi interface.

To address this issue, we enhance the emulator with a simulated WiFi interface. In brief, we implement a loadable Linux kernel module that simulates major wireless functionalities and exposes itself as a wireless driver in the same way a common physical WiFi interface does. It maneuvers the Android system and WiFi service into believing that a WiFi interface is properly installed. The simulated WiFi interface then can provide a randomly-generated MAC address. MAC address randomization is widely acceptable [15] and it is unlikely that an application will verify the authenticity of the address.

B. *Emulating Cellular Modems*

Compared with MAC addresses, generating legitimate values for the remaining properties (i.e., phone numbers, IMSI and IMEI) is more challenging. These properties are obtained from a cellular network via a modem and they can be easily verified by an application. For instance, an application provider can examine the authenticity of a given phone number by sending it a SMS containing secret codes. The codes are typically required before the user can start using the application.

It can be seen that we cannot bypass the scrutiny mechanism by simply generating random property values. Instead, the emulator is required to possess a working phone number and receive SMS from a real cellular network. In Fraus, we achieve this by implementing a software modem that bridges emulators to a cellular network just like a physical modem does. It enables the modem to provide verifiable device profile properties.

To make our approach better understood, we first introduce how the Android telephony system works. As shown in Fig 4, Android applications can request the telephony service to perform a set of cellular-related tasks (e.g., sending SMS or

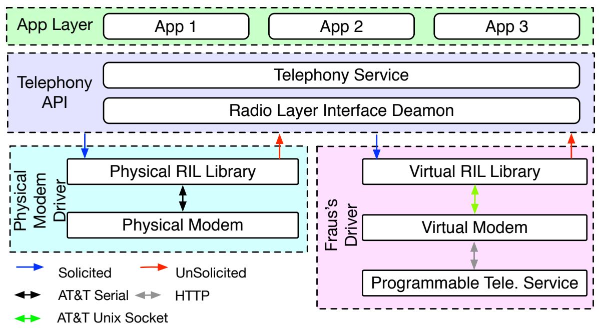

Fig. 4: Overview of the Android Telephony Architecture

dialing numbers) via APIs. These tasks will be forwarded to a native process named radio interface layer daemon (RILD). The daemon further forwards the requests to the device's modem driver library. The library then translates the requests to modem-recognizable commands (e.g., AT&T commands) and sends them to the modem for execution. Note that these commands are referred to as solicited commands as they are initiated by the telephony service. Meanwhile, the driver library will continuously monitor the responses of the delivered commands and other radio events (e.g., network ready or new message arrived) that originated from the modem. The responses and events are called unsolicited commands, which will then be propagated to the RILD and upper layers. Note that the driver library serves as a hardware abstraction layer (HAL), which exposes unified interfaces to the RILD regardless of the model of the modem. It allows RILD to be agnostic about lower-level driver implementations. In other words, to integrate a new modem to Android, developers do not have to modify the RILD and they solely need to implement the corresponding driver library.

In Fraus, a 'virtual' modem and its driver library are implemented and loaded by the Android telephony system. Figure 4 depicts the work flow of the implementation. Upon receiving a request from the RILD, the library now delivers the commands via a inter-process communication socket to a user space program. This program serves as the software modem and emulates the essential functionality of a physical modem, which is to connect to a cellular network and perform cellular communication (e.g., send/receive SMS).

Specifically, the modem achieves this by taking advantages of cloud programmable telecommunication services (PTS), which allow developers to acquire a genuine mobile phone number and to programmatically make phone calls or send/receive SMS via HTTP requests. On one hand, the modem is in charge of parsing the received solicited commands and repacking them to corresponding HTTP requests understood by the PTS. For example, when a command to send a SMS is received, the modem then makes a HTTP post to the PTS containing the body of the message and the phone number of the recipient. On the other hand, the modem continuously sends HTTP polling requests to PTS to determine whether certain cellular network events occur. Upon the arrival of a phone call or SMS, the modem then generates an unsolicited command to notify the Android telephony system.

VI. Preparing for Large-Scale Attacks

To alleviate attackers' burden of manually setting up and managing emulator instances in a large-scale MCFA, Fraus provides an emulator cluster management system. The system features a distributed architecture such that it can cope with scalability and fault-tolerance issues. Furthermore, it provides high-level APIs that present a logically centralized view and conceal the details of the underlying distributed protocols.

A. Enabling Scale-out and Fault Tolerance

One important feature of our architecture is its support for scale-out. As shown in Fig 5, Fraus incorporates multiple management nodes named *Device Manager*, each of which acts as the exclusive controller for a proportion of emulator instances. More specifically, an individual device manager is solely in charge of assigning MCFA tasks to and collecting status information from the emulator instances it controls. Such an architecture ensures the scalability by adding additional device managers to the cluster when the number of instances increases.

The distributed architecture is also designed to provide fault tolerance. The intuition is that the architecture allows multiple redundant manager instances waiting to take over in case a manager fails. By this mean, the failed manager's work could be immediately redistributed to another manager. In details, an emulator, upon creation, will establish multiple connections with available managers, but exactly one of which will be elected as the *primary manager* by performing a consensus-based leader election [16]. From that moment on, the primary manager starts controlling the emulator and the other managers remain idle. When the primary manager fails, an election will be held again and a newly-elected manager will take over the emulator.

Fraus implements the above mechanism with the help of Multicast DNS (mDNS) [17] and ZooKeeper [18]. Specifically, mDNS enables an emulator to discover and connect to the IP addresses of available device managers. ZooKeeper is utilized to perform leader election and failure detection. A manager must contact the ZooKeeper ensemble in order to become the primary manager for an emulator. If a manager loses its connection with ZooKeeper, a failure will be announced and another manager will attempt to take over the control by undergoing an election.

B. Presenting a Logically Centralized View

Although the cluster is now running across multiple device managers, Fraus preserves the simplicity of a centralized architecture with a useful API abstraction exposing a logically centralized view. It allows individuals to program and launch attack tasks without concerning the low-level details such as how the emulator instances are set up and how the tasks are dispatched and executed under the distributed environment.

Specifically, Fraus provides four types of APIs which cover every aspect of a large-scale MCFA:

1) Development APIs. Deployment APIs allow users to specific the number of emulators needed in each MCFA.
2) Automation APIs. Automation APIs assist individuals to compose the GUI automation scripts for an application.
3) Configuration APIs. Configuration APIs entitle users to adjust the setting of an emulator (e.g., device profile and network configuration).

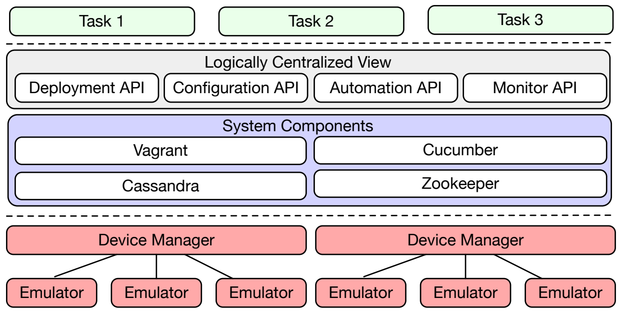

Fig. 5: System Architecture.

4) Monitor APIs. Monitor APIs provide real-time information of each emulator (e.g., memory or CPU usage).

Figure 5 depicts the implementation details of these APIs. They are implemented on top of a set of open-source building blocks. Specifically, The deployment APIs rely on *Vagrant* [19] for creating and maintaining virtual machine instances in physical servers. We also extend the original Vagrant with a variety of plug-ins to bring the support for various cloud platforms.

The automation APIs encapsulate an Android testing framework named *Cucumber* [20]. It enables users to express the automation instructions using natural language that can be easily understood by non-technical individuals. For instance, an user can easily enter his username to an application with a simple instruction like *Enter text 'username' into field with id 'username_textfield'*. Note that if the user is not in favor of the automation APIs, he can also choose other automation frameworks written in a variety of languages such as Python and Ruby.

The configuration APIs and monitor APIs make use of the distributed database *Cassandra* [21] to maintain the configuration and status information of every emulator. In details, Cassandra serves as a hub facilitating information exchange between users and emulators. For instance, the status information from an emulator is stored in the database and can be queried by users in the near future. Similarly, configurations specified by users are stored in the database and they can be retrieved by emulators via polling.

Note that the above building blocks run in each device manager. Each manager can synchronize its information with others using ZooKeeper and Cassandra such that everyone has the access to the global information to provide the logically centralized view.

VII. Experiments

In this section, we provide detailed evaluation on Fraus in terms of application compatibility, resource savings, and cluster deployment.

A. *Ethical Considerations*

Note that the main motivation is to raise public awareness of the effectiveness and simplicity of launching a MCFA using Fraus. Therefore, all the following experiments are carefully designed such that they incur neither monetary damages nor service interruptions.

B. *Application Compatibility*

To evaluate the the application compatibility, we implement an automation script that downloads and runs the top 300 applications from the Google Play Store on Fraus. Note that a large partition of them are games, which are the hardest applications to support because of their CPU-intensive and graphics-intensive nature. For comparison purposes, the experiment is carried out in the Android official emulator.

The experimental results show that 96% of the applications run successfully on Fraus, while in the Android official emulator only 59% can start properly. Fraus is highly stable and compatible with existing applications. Out of curiosity, we also pinpoint the causes of the failed cases. We discover that the official emulator failed to run most programs mainly because of the device profile scrutiny and the buggy GPU emulation, which frequently leads to the crash of the graphics system. These problems do not affect Fraus, which only fails to run a small number of applications due to CPU emulation defects. In brief, certain uncommon CPU instructions used by the applications are not properly emulated, leading to the crashing situations. The defects allow us to tell Fraus apart from physical devices, which can be used as a countermeasure to combat Fraus. We will discuss it in details in Section VIII.

C. *Resource Savings*

To measure the CPU saving of Fraus, we select 6 popular social-media and advertising applications including *Facebook*, *Wechat*, *Instagram*, *Tumblr*, *DailyCash* and *LuckyCash*, which are the frequent victims of the MCFA. Specifically, every application is automated to perform predefined tasks such as clicking a like button or viewing a video clip. Each task is performed for 30 times and the average CPU usage is reported. Note that the CPU usage retrieved from the *top* command in Android only updates every one second, which is relatively coarse-grained. We overcome the issue by directly polling the contents from the system file */proc/stat*, which provides the almost real-time CPU usage. For comparison purposes, we repeat the experiments on the official emulator.

Figure 6 demonstrates the experimental results. It can be seen that Fraus significantly suppresses the CPU utilization. For instance, Wechat consumes approximately half of the CPU resource on the unoptimized official emulator, while Fraus manage to decrease the workload to 14.7%. A more obvious example is that LuckyCash on the official emulator nearly exhausts the CPU, while the same application on Fraus only incurs tiny CPU usage of 17.9%, which is an approximate 90% reduction. A closer look at the usage of each system process reveals that the saving is mainly attributed to the Surfaceflinger and Mediaserver processes. It is expected as their core computation is evaded in Fraus.

We also obtain the memory statistics of Fraus with the help of the *ps* command. The footprint of Fraus is approximately 190 MB, which is significantly smaller than the official emulator that nearly consumes 500 MB. The 60% memory reduction indicates the effectiveness of the lazy loading mechanism.

D. *Cluster Deployment*

We launch a large-scale mocking attack with the help of the Fraus' cluster management system. Specifically, we set up a cluster running across four physical servers (Intel i7-6700 3.40 GHz CPU and 32GB RAM). The servers utilize

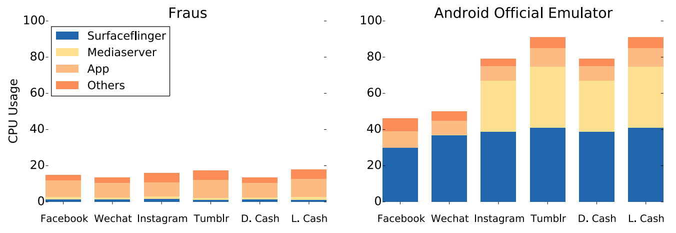

Fig. 6: Histogram of Elapsed Time for Three Devices

the *KVM* hypervisor to provide virtual machine containers. We allocate each container with 512 MB RAM and one virtual CPU. Theoretically, a physical server then can hold less then 64 instances simultaneously. In reality, owning to the small resource demand of Fraus, we manage to fit 125 instances, indicating the high cost efficiency. Inevitably the huge number of instances may push the server's CPUs to its limit. Nevertheless, the impact is limited because most Android applications are not time-critical and can bear the small delay caused by the insufficient CPU power.

VIII. CounterMeasures

In this section, we propose two feasible countermeasures to combat Fraus. The intuition underlying our countermeasures is that applications can detect the emulator environment and refuse to perform any activities by terminating or stalling the execution. The most common approach to detect the presence of an emulator environment is based on the fingerprinting of the runtime environment. This includes checking for specific artifacts, such as some specific environment variables, background processes, or defective results obtained from certain API interfaces [22]. However, we argue that these methods are not fully reliable because an attacker can always modify the system source codes to build a deceiving runtime environment.

Instead, our approaches detect the Fraus environment by checking the platform-specific characteristics that are different with respect to a physical device. The characteristics include a tiny defect in the CPU instruction set and significantly different timing properties of the graphics system.

Examining CPU Instruction Sets. An emulator is typically X86-based in pursuit of performance, because the readily-available X86 instructions can be directly executed owning to the Hardware Virtualization technology. Nevertheless, the X86-based emulator has to handle a practical issue; because nearly all physical devices are equipped with ARM CPUs, various Android applications written in C/C++ are solely compiled with the ARM instruction set, which cannot be directly executed by X86-based system. To address this issue, a binary translator named *native bridge* is utilized to translate the instructions from ARM to X86.

Our first approach makes use of the fact that the binary translator does not provide support for the ARM *NEON* CPU instruction set, which is intended for accelerating audio and video encoding. This defect offers us a chance to distinguish whether the device has an ARM CPU. In details, an application can embed a C program containing the ARM NEON instructions. If the program can be successfully loaded, it means the device is equipping with an ARM CPU and it is not likely an emulator.

Examining Timing Properties of the Graphics System. Fraus evades the intensive graphical computation in order to

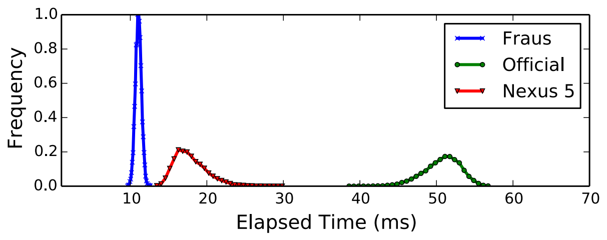

Fig. 7: Normalized Histogram of Elapsed Time for Three Devices

save CPU processing power. However, it comes with a side effect that the timing properties of the graphics system are affected. For instance, the time consumed by an application to draw a frame is significantly shortened. To demonstrate the effect, we first implement a test program that renders a rotating 3D color-cube and measures the elapsed time per invocation of the draw() method (i.e., time consumed to draw a frame). The program is then tested on two emulators including the official emulator and Fraus as well as a physical device LG Nexus 5. For each device, we sample the elapsed time for 10000 frames and report the histogram in Fig. 7.

One important finding is that the devices are statistically different from each other (Wilcoxon, $p \leq 0.001$). For example, Fraus and Nexus 5 possess a median of 11 ms and 18 ms respectively, while the statistics value for the official emulator soars into 59 ms. Thus, we can easily rule out the official emulator. Meanwhile, we notice that Fraus exhibits a bell curve with observable measurements from 9 to 11 and the majority of measurements are clustered closely around the median. In contrast, Nexus 5 presents a right skewed distribution ranging from 12 to 27 with the data points spreading far from the median. The difference are mainly due to that in a physical device, the elapsed time likely varies depending on the complexity of each frame. For instance, it consumes more time to draw a frame filling with 10 triangles than those only with 1 triangle. Conversely, the complexity does not affect Fraus because the involved computation is skipped and the elapsed time for each frame should remain highly consistent.

Based on the above observations, heuristics can be devised to detect the Fraus runtime environment. Specifically, We first measure the standard deviation s of the sample. After that, we obtain the L-skewness statistics r [23] as follows:

$$r = \frac{\frac{1}{3}\binom{n}{3}^{-1}\sum_{i=1}^{n}\left\{\binom{i-1}{2} - 2\binom{i-1}{1}\binom{n-i}{1} + \binom{n-i}{2}\right\}x_i}{\frac{1}{2}\binom{n}{2}^{-1}\sum_{i=1}^{n}\left\{\binom{i-1}{1} - \binom{n-i}{1}\right\}x_i}, \quad (1)$$

where x_i represents the ith sample. We will claim Fraus is detected if $s \leq 1$ and $abs(r) \leq 0.05$. The intuition behind the heuristic is that a tiny r and s indicates a highly-symmetry and tightly-coupled bell curve, which is the Fraus's unique feature.

IX. Related Work

MCFA is a serious threat to the Internet economy. From an economics viewpoint, the research work [24] investigates the mechanisms and processes associated with the click-fraud industry. Miller et al. [25], on the other hand, provides a detailed survey on the click fraud techniques. Specifically, there exists two common approaches to launch a MCFA. One is through mobile malware [26], which performs click actions stealthily as a background task. MadFraud [27] further discovers that the click fraud malware attempts to remain stealthy by only periodically sending clicks. Nevertheless, the power of this attack is of uncertainty because it is mainly decided by the number of infected phones. Moreover, the advance of security countermeasures such as [28] and [29] also renders a majority of click fraud malware ineffective. Thus, attackers shift their focus on the second method, referred to as 'click farms' [30]. A typical click farm deploys thousands of mobile devices and automates them to generate click events by computers. Clearly, the power of this approach is increased along with the number of devices, however, which also incurs a substantial purchasing expense. One alternative to the expensive physical devices is emulators. Though emulators have been utilized in a great amount of literature for runtime analysis [31] and malware detection [32], little research has been conducted to investigate the usability of emulators on MCFA. We bridge the gap by presenting Fraus, that enables us to launch a MCFA with emulators.

To prevent the MCFA, a number of countermeasures have been introduced. The simplest solution is to use threshold-based detection; if a huge number of click events with the same device profile are received, they can be considered as fraud. This approach is not effective as device profiles can be easily modified. More sophisticated solutions are typically based on data-mining techniques [33]. For example, the research work [34] utilizes group Bloom filters over click event time series to detect the fraud. Another interesting approach is to use bait contents [35] to check whether clients are automated. We can combine these approaches with our proposed countermeasures to further increase the resistance against Fraus.

X. Conclusion

In this paper, we raise public concerns of the simplicity of launching a large-scale MCFA by presenting Fraus. Fraus has a tiny CPU and memory resource demand. Besides, Fraus can generate a seemingly authentic device profile and disguise itself as a legitimate device. To facilitate a large-scale attack, Fraus also provides a distributed emulator cluster management system, which is scalable and fault-tolerant. We evaluate the performance of Fraus and demonstrate that Fraus has high application compatibility and cost efficiency. Finally, we also propose countermeasures leveraging Fraus's platform-specific characteristics to combat such an attack.

References

[1] "Mobile click fraud: What 24 billion clicks on 700 ad networks reveal." [Online]. Available: https://www.tune.com/blog/mobile-ad-fraud-24-billion-clicks-700-ad-networks-reveals/

[2] "Businesses could lose 16.4 billion to online advertising click fraud in 2017: Report." [Online]. Available: http://www.cnbc.com/2017/03/15/businesses-could-lose-164-billion-to-online-advert-fraud-in-2017.html

[3] "The bizarre 'click farm' of 10,000 phones that give fake 'likes' to our most-loved apps." [Online]. Available: http://www.mirror.co.uk/news/world-news/bizarre-click-farm-10000-phones-10419403

[4] G. Cho, J. Cho, Y. Song, and H. Kim, "An empirical study of click fraud in mobile advertising networks," in *Availability, Reliability and Security (ARES), 2015 10th International Conference on*. IEEE, 2015, pp. 382–388.

[5] "Inside chinese click farms." [Online]. Available: https://www.kotaku.com.au/2017/05/inside-chinese-click-farms/

[6] "Averge selling price for smartphones." [Online]. Available: https://www.statista.com/statistics/283334/global-average-selling-price-smartphones/

[7] "Vps comparsion." [Online]. Available: https://github.com/joedicastro/vps-comparison

[8] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna, "Baredroid: Large-scale analysis of android apps on real devices," in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015, pp. 71–80.

[9] "Official android emulator." [Online]. Available: https://developer.android.com/studio/run/emulator.html

[10] "Bluestack." [Online]. Available: http://www.bluestacks.com/

[11] "Genymotion." [Online]. Available: https://www.genymotion.com/

[12] "Dynamic linker hooking techniques." [Online]. Available: http://man7.org/linux/man-pages/man8/ld.so.8.html

[13] M. Zechner, J. DiMarzio, and R. Green, "An android in every home," in *Beginning Android Games*. Springer, 2016, pp. 1–14.

[14] T. Graphics, "Mesa3d source code. available from git," *anongit. freedesktop. org/git/mesa/mesa*, 2008.

[15] M. Vanhoef, C. Matte, M. Cunche, L. S. Cardoso, and F. Piessens, "Why mac address randomization is not enough: An analysis of wi-fi network discovery mechanisms," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 413–424.

[16] N. A. Lynch, *Distributed algorithms*. Morgan Kaufmann, 1996.

[17] C. N. Ververidis and G. C. Polyzos, "Service discovery for mobile ad hoc networks: a survey of issues and techniques," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 3, 2008.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8. Boston, MA, USA, 2010, p. 9.

[19] M. Hashimoto, *Vagrant: Up and Running: Create and Manage Virtualized Development Environments*. " O'Reilly Media, Inc.", 2013.

[20] M. K. Kulkarni and A. Soumya, "Deployment of calabash automation framework to analyze the performance of an android application," *Journal for Research— Volume*, vol. 2, no. 03, 2016.

[21] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[22] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014, pp. 447–458.

[23] D. Joanes and C. Gill, "Comparing measures of sample skewness and kurtosis," *Journal of the Royal Statistical Society: Series D (The Statistician)*, vol. 47, no. 1, pp. 183–189, 1998.

[24] N. Kshetri, "The economics of click fraud," *IEEE Security & Privacy*, vol. 8, no. 3, pp. 45–53, 2010.

[25] B. Miller, P. Pearce, C. Grier, C. Kreibich, and V. Paxson, "What's clicking what? techniques and innovations of today's clickbots." in *DIMVA*. Springer, 2011, pp. 164–183.

[26] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 3–14.

[27] 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*. ACM, 2014, pp. 123–134.

[28] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 89–103.

[29] S. Nath, "Madscope: Characterizing mobile in-app targeted ads," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 59–73.

[30] A. Juels, S. Stamm, and M. Jakobsson, "Combating click fraud via premium clicks." in *USENIX Security Symposium*, 2007, pp. 17–26.

[31] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[32] L.-K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis." in *USENIX security symposium*, 2012, pp. 569–584.

[33] X. Zhu, H. Tao, Z. Wu, J. Cao, K. Kalish, and J. Kayne, "Ad fraud categorization and detection methods," in *Fraud Prevention in Online Digital Advertising*. Springer, 2017, pp. 25–38.

[34] L. Zhang and Y. Guan, "Detecting click fraud in pay-per-click streams of online advertising networks," in *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*. IEEE, 2008, pp. 77–84.

[35] H. Haddadi, "Fighting online click-fraud using bluff ads," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 2, pp. 21–25, 2010.