TruzCall: Secure VoIP Calling on Android using ARM TrustZone

Amit Ahlawat
Dept. of EECS
Syracuse University
Syracuse, NY, USA
aahlawat@syr.edu

Wenliang Du
Dept. of EECS
Syracuse University
Syracuse, NY, USA
wedu@syr.edu

***Abstract*—Use of mobile phones today has become pervasive throughout society. A common use of a phone involves calling another person using VoIP apps. However the OSes on mobile devices are prone to compromise creating a risk for users who want to have private conversations when calling someone. Mobile devices today provide a hardware-protected mode called trusted execution environment (TEE) to protect users from a compromised OS. In this paper we propose a design to allow a user to make a secure end-to-end protected VoIP call from a compromised mobile phone. We implemented our design, *TruzCall* using Android OS and `TrustZone TEE` running `OP-TEE` OS. We built a prototype using the `TrustZone`-enabled Hikey development board and tested our design using the open source VoIP app `Linphone`. Our testing utilizes a simulation based environment that allows a Hikey board to use a real phone for audio hardware.**

***Index Terms*—Security, Mobile Computing, Voice I/O**

I. Introduction

Mobile phones are one of the most common devices used by people today, with the basic function of calling another person. In recent years, VoIP apps such as Signal [1] and Whatsapp [2] have become popular ways for making a call. Unfortunately the mobile OS platforms on which these apps run have made the use of VoIP apps more risky in terms of user privacy. One of the popular mobile OS Android now has a majority market share [3], [4]. At the same time, CVE numbers show that the number of disclosed vulnerabilities in Android has remained high (more than 500) from 2015 to 2018 [5], [6]. A recent attack on Android could achieve arbitrary code execution in a privileged process by using a crafted image file [7]. The problem is also compounded by the fact that various actors are trying to compromise Android OS including hacking groups [8] and nation states [9]. The ever present risk of mobile OS compromise can limit one of the important rights in human society i.e. freedom of speech. In context of mobile phones, this translates to being able to call anyone and talk on any subject without fear of someone else listening on the call. Today different types of users need to have a secure means of calling, including activists, journalists, government employees etc.

Most `ARM` based smartphones today have a security feature called `TrustZone` which provides a trusted execution environment (`TEE`) called *secure world* which is isolated from the environment where normal apps are installed, called the *normal world*. In this paper we ask the following question: *How can we enable a VoIP app to leverage ARM TrustZone to protect user's conversation from an untrusted OS during a VoIP call while using existing OS APIs and VoIP protocol ?* In this paper, we present a design called TruzCall that allows an Android user to make a secure end-to-end encrypted VoIP call from a compromised mobile phone. Using TruzCall a compromised Android OS cannot listen to user's conversation during a VoIP call. In comparison to existing work, TruzCall's goal is to be transparent to developers and to the existing VoIP infrastructure. We want to allow developers to be able to use existing OS APIs and VoIP protocols to write VoIP apps, while being able to secure the user's conversation using `TEE`. At the same time, the use of TruzCall should not require any change to the VoIP infrastructure in place.

One of the challenges that is encountered in designing a system like TruzCall is latency. Since VoIP is a real time system, if the normal world stack invokes `TEE` at one or more points, it will add computation time to the VoIP call. Any additional time will add latency and will thus affect voice quality. The design should reduce latency. Another challenge is the hardware setup to do prototype evaluation. In `TEE` research, interfacing hardware peripherals like mic and speaker with the `TEE` OS on a development board can be challenging for non-hardware experts with limited resources. In order to evaluate the design, we want to use a hardware setup that allows easier prototyping.

II. Problem and Idea

In this section we discuss the threat model for the secure VoIP problem. We further elaborate the problem and how our main idea differs from the existing work.

A. Threat Model and Assumptions

The normal world OS (including Android and the VoIP app) is not trusted. The secure world, including the TEE OS and trusted applications (TA), is trusted. The user using the device is trusted. The device hardware, including the audio peripherals (mic and speaker), is trusted. The VoIP network is not trusted, although this work does not try to protect against network based attacks. This work is targeted for users who

want to securely call friends, family or someone they know personally or have met before. This work does not cover key exchange done by VoIP apps (at the beginning of the call), which is why it cannot be used to call an unknown person. To use the design discussed in this paper, two users need to exchange a secret phrase using a secure side channel (this will be used to derive the key). This work can be extended to add key exchange using the TEE by splitting protocols like DTLS [10]. We also assume that the user wants to use TruzCall for a one-to-one call, and not conference calling.

B. Problem

When user initiates a call using a VoIP app, the app uses OS APIs to fetch audio, processes the audio, and sends out the packet over the network (reverse flow for incoming packets). The app uses a VoIP protocol like SRTP to encrypt and calculate HMAC for the audio payload (in RTP packets), and send the encrypted payload to the callee device. In order for the VoIP app to use TrustZone to protect the user's conversation, the app should still be able to use the existing OS APIs and existing VoIP protocols. The user should be able to have the conversation using the existing audio hardware and the conversation audio should not be leaked to the normal-world OS. The problem of having a secure VoIP call on an untrusted OS has been covered in a previous work [11]. However there is no existing work that addresses this problem and satisfies our design constraints. VoIP apps consist of some essential stages in their control flow and we want to preserve the relative structure of the VoIP software stack. We focus on the essential layers of audio I/O, RTP packet construction / parsing, SRTP, and network I/O. We also want to minimize the TCB as TruzCall leverages the TEE.

C. Existing Work on Secure VoIP

The idea of having a secure VoIP call on an untrusted OS has been discussed before in the work *"A Hardware-Assisted Proof-of-Concept for Secure VoIP Clients on Untrusted Operating Systems"* [11]. In this section we discuss the differences between TruzCall and this related work. The existing work is done on a Xilinx board, which includes a PS section and PL section (FPGA). The PS and PL sections are analogous to normal and secure world respectively. The work is intended for devices like VoIP phones (handset). They used the Linphone app [12] for testing and modified it such that for incoming SRTP packet, the header information and payload is forwarded to secure hardware, and for outgoing packet the SRTP header and encrypted payload are sent from secure hardware to the normal world. There are several differences between the existing work and TruzCall: (1) Commercial mobile phones don't rely on FPGA; instead they ship with ARM boards that have TrustZone. The existing work does not address any challenges related to leveraging TrustZone for secure VoIP. (2) Xillinux OS does not reflect mainstream mobile OS like Android. The existing work does not address leveraging TrustZone in mobile OS audio stacks to allow existing Audio APIs to be used. (3) A VoIP app has a flow for handling audio packets. In the existing work the RTP layer has been eliminated from the normal-world app flow as the design forwards header/payload with secure hardware at the SRTP layer. This breaks the relative structure of the software stack used to implement a VoIP app. The design does not utilize Audio APIs in the normal world to record/play audio data which changes the way developers write VoIP apps. Moving header generation/parsing functionality into the secure world increases the TCB as only part of the SRTP layer remains in the normal world. TruzCall's goal is to maintain the relative structure of the essential parts of the software stack for a VoIP app and avoid moving unnecessary components into the TEE.

D. Main Idea

Figure 1 shows the main idea of TruzCall. The various stages in a VoIP stack work in parallel as a pipeline, each stage feeding data to the next. The audio pipeline in the VoIP app consists of some essential stages. To allow the VoIP pipeline to maintain its existing flow while keeping user's conversation audio in the TEE, we invoke TEE at the stages for audio API usage and SRTP. This allows the use of the existing relative structure of the software stack.

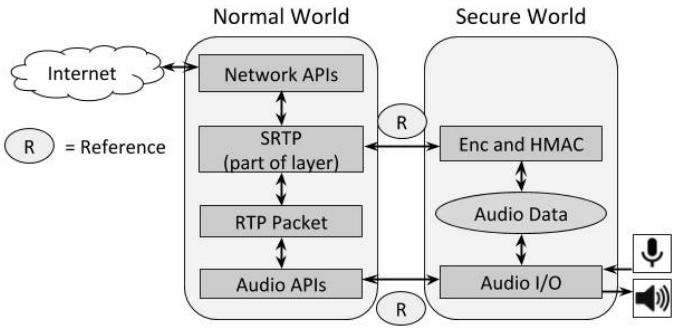

Fig. 1. TruzCall Main Idea

At the beginning of the call, TEE takes control of the audio peripherals. This can be done using TrustZone hardware features and has been done in other works like SeCloak [13]. In order for the TEE invocation at several stages of the VoIP stack to work together, we use a reference design pattern. When the VoIP app asks for audio using existing APIs, the TEE invocation provides it a reference to the real audio data (saved in TEE) via the existing normal-world OS audio APIs. The app then proceeds with preparing the RTP packet. When the flow reaches the SRTP layer and it needs to encrypt the data in the RTP payload (which is a reference), we invoke the TEE and pass the audio data reference. The TEE encrypts the data corresponding to the reference and returns the encrypted payload and HMAC to the SRTP layer to allow the VoIP app flow to continue. This way only essential cryptography operations for SRTP are moved into the TEE. The reverse flow happens for packets received by the device for playback.

E. TEE Integration Using References

Reference is an abstract concept and is used in TEE related designs when the normal world has multiple stages. References

have been used in the DRM media pipeline [14] and in the recent work TruzDroid [15]. A reference also implies some type of associated data in the TEE and its management. The application of references and its data management in any use case requires insight into the problem. In DRM, TEE is used with normal world pipeline stages to provide secure decryption of media like audio and video. On a mobile device, DRM commonly handles received data [16], while TruzCall has to secure audio to be sent and received. The approach of handling data and the integration of TEE with normal world stages makes TruzCall's handling of secure audio different from DRM. In DRM as data arrives into a buffer, it can be decrypted and queued to be played out [17]. As we will see in Section III-C, due to packetization and jitter handling in VoIP, the data is collected or played out in smaller chunks compared to the buffer to be sent out or received. This requires a different design for how data is managed in the TEE in order to reduce latency (Section III-E). Also the way TEE is integrated in normal world stages in DRM and TruzCall have different implications. For example, in DRM implementation [18], TEE can be used with every normal world stage of the pipeline. If the same approach was applied to VoIP, it would increase the overall latency of the solution. The structure of a reference is also problem specific. In the case of DRM [14] [19], a reference can represent a native handle containing a file descriptor to represent a secure buffer. In TruzCall, the reference is designed to reflect the size of audio data in usage, as well as a cache index in the secure TEE memory for audio data organized in ring buffers. Coincidentally, in TruzDroid [15] references are used to carry length and TEE memory location information.

F. VoIP Call Flow

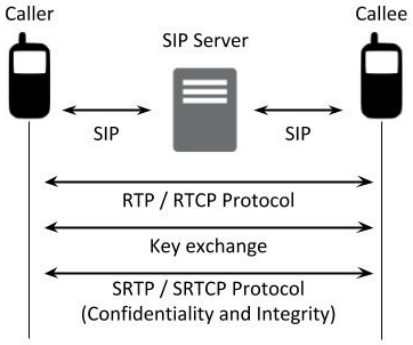

Fig. 2. VoIP Call Flow

A comparison of protocols used by VoIP software using end-to-end encryption can be found at [20]. From the data available for protocols used by apps, a common protocol for VoIP with open source implementation is SRTP [21] using SIP [22] for call initiation. Figure 2 gives a high level view of the flow involved in connecting a VoIP call. If a caller wants to call a callee, they will first use the SIP application-layer protocol [22], [23] to exchange information. The information is exchanged using SDP messages [24] enclosed within SIP messages. The SIP protocol does not carry any audio data; it is used to initiate a session between the two end points. Once the connection is established, protocols like RTP [25] are used to deliver audio between the two end points. RTP is used alongside the RTP Control Protocol (RTCP). RTP is used to carry media streams, while RTCP is used to monitor transmission statistics and quality of service. In context of the TruzCall design, we need to point out the sequence number field of the RTP header, which increments by one for each RTP packet sent and can be used by the receiver to detect packet loss and to restore packet sequence. SRTP is a profile of RTP that provides confidentiality, message authentication, and replay protection to RTP traffic. A sister protocol SRTCP provides the same features for RTCP. SRTP resides between the RTP application and the transport layer. It intercepts RTP packets and then forwards an SRTP packet containing encrypted payload and HMAC on the sending side, and intercepts SRTP packets and verifies HMAC and decrypts payload to provide an RTP packet up the stack on the receiving side. SRTP and SRTCP need keys for encryption and HMAC. These keys are derived from master keys which are set up using a key exchange mechanism. Protocols used by VoIP to setup master keys include DTLS [10] and ZRTP [26].

III. TruzCall Design

In this section, we discuss how the TruzCall design achieves a secure VoIP call. We want to emphasize that the changes we made to the Linphone app are within the various libraries used by Linphone. The app is composed of several modules, including libraries for SRTP, RTP [27], SIP [28] and audio I/O [29]. A different VoIP app using the same libraries should be able to use TruzCall's design. The changes made in the audio framework would be applicable to any VoIP app.

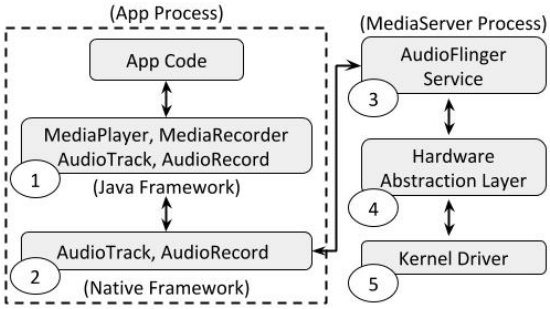

Fig. 3. Android Audio Architecture

A. TEE Invocation and Data Encoding

In this section we discuss how TEE is leveraged by various stages of the normal-world VoIP audio pipeline. We also discuss what encoding is used by the TEE to convey the audio data to the normal-world pipeline. Figure 3 shows the architecture of Android's audio stack [30]. An Android app can use various Java APIs for Audio I/O, all of which use the same underlying native framework. This communicates with the underlying AudioFlinger service (Android's sound

server [31]). In order to protect the user's conversation during a VoIP call, we need to leverage TEE to provide the VoIP app the user's audio without ever releasing the plain text audio from the secure world. This means user's audio can only enter the normal world in an encoded form. The question becomes at which layer in the normal-world stack should TEE be invoked for audio. In Figure 3, Audioflinger (3) is responsible for resampling [32] and mixing audio streams [31], as well as applying effects. If we use TEE at this layer, we would have to make sure that there is a path that doesn't alter the data obtained by or to be given to the TEE, in order not to break the audio encoding. Using TEE at (4) or (5) will incur the same issue as data will pass through the AudioFlinger. Layer (1) provides the app with several APIs to read/write audio. To allow the VoIP app developer to use any API for Audio I/O, our decision was to use TEE at layer (2).

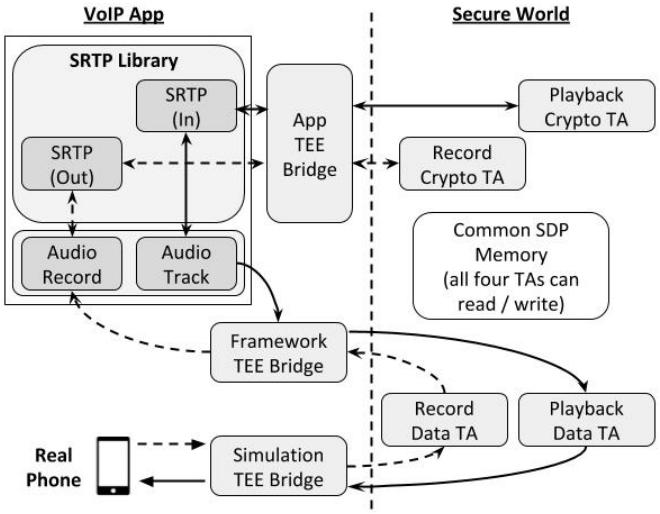

Fig. 4. TEE Invocation by Audio Framework and SRTP

1) Audio Data Encoding: Once the TEE invocation point for the audio framework has been identified, we have to decide an encoding to provide audio data to the normal world. The data provided to the native audio framework can be encrypted by the TEE. In this case, the cryptographic operations done in the app's SRTP layer will become redundant; the audio data will be encrypted twice. It will also add latency to the VoIP flow because of the additional time spent encrypting the audio data again. One way to handle this design option would be to disable the operations done in the normal world SRTP layer, but this would disable an essential stage of the app flow. The goal of TruzCall is to preserve the relative structure of the essential layers in the VoIP app, including the SRTP layer. In order to allow the app to still use the SRTP library for encryption and HMAC, we don't provide encrypted data to the native audio framework. When the app requests audio data, the native audio framework gets a reference for the audio. The reference is a string with the same length as the requested audio data. The RTP layer prepares a packet containing audio reference(s) as the payload. When the SRTP layer needs to encrypt the packet, it invokes the TEE which encrypts the audio data corresponding to the audio reference(s) in the RTP payload and calculates the HMAC for the RTP packet. Once the TEE returns the result, the SRTP flow can continue to send the packet out. On the receiving device the reverse will happen. The SRTP library will invoke the TEE to get an audio reference corresponding to the RTP encrypted payload, with the decrypted audio staying in the TEE. When the native audio framework needs to play the audio, the reference is given to the TEE which plays the corresponding audio. Figure 4 shows the TEE invocation points (the RTP layer is omitted). To improve audio quality and reduce bandwidth requirements, a VoIP app performs additional audio computation (e.g. resampling, compression) between stages Audio API and RTP packet generation/parsing (Figure 1). We disabled these stages in order for reference data to not to altered.

2) Independent Audio Pipeline Stages: Given two types of TEE invocations (by the native audio framework and by the SRTP library), TruzCall needs to make sure that the TA logic and corresponding data for these invocations is handled in a way such that there is no bottleneck created in the normal-world audio pipeline. To handle the two types of TEE invocations, the design needs to allow sharing of data via a common memory space between the corresponding TA logic. The plain text audio in TEE must be accessible to the cryptographic logic when SRTP library provides it a reference and conversely the audio data decrypted must be accessible to the TEE audio playback logic when it is provided with a reference by the native audio framework. When a TA is invoked, it can access three types of memory including stack, heap and shared memory. Only data in heap and shared memory can retain its value across multiple TEE invocations. TEE provides two types of shared memory, namely unsecure shared memory (used by normal world to pass arguments) and secure shared memory (not visible to normal world, but visible to TEE components). The two candidates to keep plain text audio in common memory are heap and secure shared memory. Heap cannot be used for this design because our design constraint demands reduced latency. In order to use heap as a common memory, the TEE logic corresponding to different normal-world stages will need to belong to the same TA because the TEE OS provides isolated heaps for different TAs. This would require multiple normal-world pipeline stages to invoke the same TA, which would require the TA to be configured with TA_FLAG_MULTI_SESSION [33]. This would make the TA invocations serialized i.e. different normal-world stages won't be able to call the TA simultaneously (the call from one stage will have to wait for the call from the other stage to finish). This would create a performance bottleneck and add latency. Therefore we use the secure shared memory to provide common memory for plain text audio in the TEE. OP-TEE provides this feature via secure data path (SDP) [34]. It allows a secure pool of memory to be allocated in the TEE with normal world having a reference to this memory. The SDP reference is made available to the TEE bridges in the normal world. The normal-world bridges pass the reference when

invoking corresponding TAs so that the common memory containing the plain text audio is accessible in the TA logic.

3) TEE Bridges and TAs: Figure 4 shows three TEE bridges and four TAs inside the TEE. The TEE bridges are native daemons (running with root privilege) that allow normal world components to invoke the TAs. The App TEE Bridge allows the SRTP layer (Java code) to invoke the Record Crypto & Playback Crypto TAs responsible for cryptographic operations (encryption and HMAC) in the TEE. The Framework TEE Bridge allows the native audio framework to invoke the Record Data TA responsible for collecting audio data and providing reference for audio data, and Playback Data TA responsible for playing out audio data corresponding to the provided references. The Simulation TEE Bridge allows the design to record & play audio using a simulation environment by using a real phone to provide the audio hardware (discussed in Section V).

TruzCall sends and receives audio references to/from the TEE, which means each time the normal world needs audio or wants to play audio a TEE invocation will be needed. Each invocation from the normal world involves opening a session with the TEE OS. Each TEE invocation session consumes some memory in the TEE OS due to saved state. At the same time the TEE environment is only assigned a limited amount of memory [35]. If the normal world keeps opening sessions based on the requirements of an on-going VoIP call, the TEE OS will exhaust its memory and deny any more TA invocations which will stop the secure call. Closing a session and opening it again for each TEE invocation will contribute to latency. To solve this issue we make our TEE bridges *persistent* by reusing TEE sessions. A bridge only initiates one TA session (with each TA that needs to be used) at the beginning of the call. All other TEE invocations via the bridge reuse the persistent session. This way the VoIP call can use TEE without exhausting its memory and can go on for any duration.

B. VoIP Call Initiation

In this section we will discuss how TruzCall handles the VoIP call setup. As mentioned in Section II-A, we assume that the user wants to call a known person as we do not handle the key exchange using the TEE. Before a secure call is setup, the caller and callee need to exchange a secret phrase using a text entry that will be input using a secure UI. This has been addressed in other works [15], [36]–[38]. When the user types in this secret phrase, the user also enters the SIP address of the callee. The secret phrase and the associated SIP address are saved in the TEE trusted storage [39].

The user will initiate the VoIP call using the app's UI in the normal world. The call will need to first establish a connection using SIP using a SIP INVITE packet to the Linphone server. Before sending this packet, we invoke a TA and pass the callee's SIP address. The user will be shown a confirmation UI asking whether a secure call should be initiated. Once the user approves, the TA will lookup the secret phrase associated with the SIP address. Both the SRTP and SRTCP protocols need two sets of master key and salt (for send and receive directions). The TA concatenates the secret phrase with a random string generated using the TEE random device. The TA calculates the master keys and salts by concatenating this new string with four fixed values and generating SHA-256 hashes. Each master key needs to be 16 byte and master salt needs to be 14 byte, so each key + salt pair is 30 bytes (we use first 240 bits of the hash). The TA keeps the master keys and salts in memory. Next the TEE would take control of the audio peripherals on the device so that normal world cannot access the user's conversation audio during the VoIP call (in our testing we use a simulation based environment, but in an actual product TEE will need to control the audio hardware). A secure LED light (only accessible to the TEE) will be turned on which allows the user to know whether the audio hardware is under TEE's control. The TA returns control to the normal world and returns the random string that was concatenated to the secret phrase. The SIP flow continues and uses this random string as its CALL-ID [22]. The CALL-ID will be conveyed to the receiving device when it receives the SIP INVITE so that it can generate the corresponding master keys and salts. Once SIP has established a connection, the app will use the RTP protocol to communicate with the other device on the call. RTP RFC [25] dictates that the initial value of the sequence number should be random. After SIP has established a connection, a TA is invoked which generates a random number using TEE random device. This number is returned to the normal world and is used as the initial sequence number. In Section III-D we discuss how the TEE checks whether the normal world has obeyed to use the sequence number given by the TEE.

As shown in Figure 2, after an RTP channel is setup, a key exchange needs to take place to obtain master keys and salts to secure RTP and RTCP. Instead of using protocols like DTLS [10] and ZRTP [26], the app invokes the TA which has the master keys and salts in memory. Instead of returning the master keys and salts, the TA returns references (random strings with same length as key/salt and mapped to these data in the TA memory) to the normal world. For secure RTP / RTCP channel to be setup the app uses a key derivation function (KDF). This derives a session encryption key, session HMAC key and a session salt based on a master key and salt. We use the TA to generate the session keys and salts, by passing it the references for master keys and salts. We use the same approach to generate the keys in the TA as the normal world does in the non-secure case. The keys are generated using AES-CTR. The counter and plain text are fixed in the app for individual cases of key calculation; only variable involved is the master key and salt. The KDF passes the counters and plain texts to the TA. The TA returns references for session keys and salts to secure RTP. The TA returns the sessions keys & salts to secure RTCP in plain text, because we don't handle RTCP in the TEE for the TruzCall design as RTCP does not carry audio payload. It should be noted that the TEE invocation by KDF is only utilized once (at the beginning of call). It does not add any latency to user's conversation once the secure call is setup. Once the session keys and salts are setup, RTP and

RTCP can be secured using SRTP and SRTCP.

So far, we have described the call setup flow on the caller's device. The flow on the callee device will be similar. When the SIP INVITE is received, before handling it, a TA is invoked and is passed the caller's SIP address and the CALL-ID. The control of the audio hardware will be taken over by the TEE. The TA looks up the secret phrase corresponding to the SIP address. The TA will calculate the master keys and salts. The KDF in normal world will invoke the TA in a similar manner to generate session keys and salts to secure RTP and RTCP.

C. TEE Invocation by Audio Framework

Android native framework consists of AudioRecord and AudioTrack, which contain the functions obtainBuffer() and releaseBuffer(). All Audio I/O utilizes these functions. TruzCall invokes TEE in these native framework functions. In this section we discuss how these invocations work. In Android's implementation (AOSP), these native functions interact with the Audio Flinger, which provides the app process a buffer to either read data from or write data to. In TruzCall, the native functions interact with the TAs to either get audio reference from or send audio reference to the TEE. The native framework allows reading and writing audio in different modes [40], [41], including a callback mode using which the audio data is fetched from or provided to a callback function. Linphone's native mediastreamer library [29] uses the callback mechanism for audio I/O. The native framework runs native threads (AudioRecordThread and AudioTrack Thread) which use obtainBuffer(), the callback and releaseBuffer() in a while loop (Figure 5). The threadloop() function containing this while loop is executed periodically based native Thread class [42], [43].

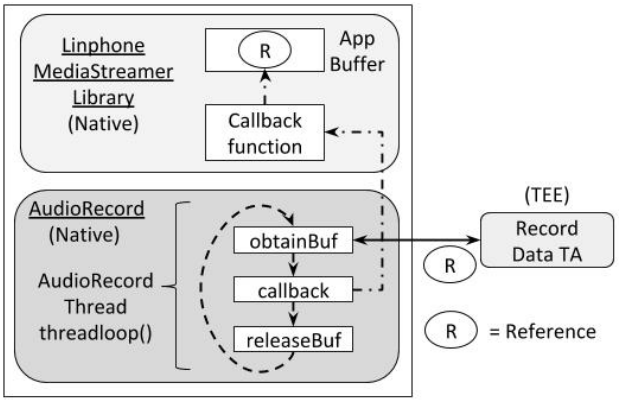

Fig. 5. Use of TEE in Native AudioRecord

1) TEE Invocation by AudioRecord: VoIP apps using RTP buffer audio data before sending it out (packetization [44]). In case of Linphone, 640 bytes is buffered. In AOSP's implementation, to construct 640 bytes of audio data, at the call initiation the app instructs the audio framework that it should be notified each time 640 bytes of audio data is available. As the call progresses, the AudioRecordThread attempts to get the requested amount of audio from the AudioFlinger via obtainBuffer(). If enough audio data is not available, the framework notifies the app with the available amount via the callback and makes up for the remainder by continuing the loop. TruzCall emulates this behavior as the AudioRecordThread uses obtainBuffer() to allocate a buffer and ask the Record Data TA for a reference based on the size requested by the app. If the requested amount of audio data is not available, the Record Data TA returns a reference of the same length as the available amount. The AudioRecordThread sends the reference to the mediastreamer library via a callback. The releaseBuffer() call frees the buffer. The AudioRecordThread makes up for the remainder by continuing the loop.

2) TEE Invocation by AudioTrack: VoIP apps using RTP use a jitter buffer. The RTP library [27] uses this buffer to hold packets as they arrive because of the possible variable delay involved. This allows the packets to be played in sequence. When the call is in progress, the amount of audio played by the app varies based on how much data the app wants to make available. When using Android's AOSP implementation, at call initiation the app instructs the native audio framework to request a certain number of bytes from the app during the call. The AudioTrackThread is constrained by the amount of audio data the AudioFlinger can take based on the obtainBuffer() call. The AudioTrackThread requests the app based on the buffer size available from AudioFlinger. The app responds with a size equal to the minimum of size asked and size available. The AudioTrackThread sends the audio data to AudioFlinger using releaseBuffer(). The AudioTrackThread handles the remainder by continuing the loop. TruzCall's design emulates this behavior. Initially AudioTrackThread requests the app based on the configured size via the callback. The callback gets the audio reference from mediastreamer. The reference received from the app is sent to the Playback Data TA in releaseBuffer(). The TA responds with the available size in TEE. If there is a remainder from the configured size (set at call initiation), then the loop is continued, and the AudioTrackThread requests a size from the app based on the buffer size available in the TEE.

D. TEE Invocation by SRTP

In this section we discuss how SRTP leverages the TEE for encryption and HMAC. The SRTP library does replay detection [21], which we do not move into the TEE. The SRTP library in Linphone uses AES-CTR for encryption using 128 bit keys and uses SHA-128 when calculating HMAC. For AES-CTR, the SRTP library calculates the counter from four values: packet index, SSRC, salt and a block counter [45]. Packet index is a combination of the sequence number and a rollover counter (counts sequence number rollover of 65535). Packet index is distinct for each packet. The salt is calculated at the beginning of the call and is kept in the TEE. SSRC is an

identifier for a source of RTP packets involved in a VoIP call and is given to TEE at the beginning of the call. The block counter increments from zero for each packet. As mentioned in Section III-C, the native audio framework provides audio references to the app based on the size of available audio. This results in the RTP packet eventually constructed in the app consisting of a set of references in the payload. For each RTP packet, the SRTP layer sends the entire packet and session encryption & HMAC key references to the Record Crypto TA. The TA calculates the counter for AES-CTR using the sequence number in the RTP header. For the first packet the TA compares the sequence number against the initial sequence number to ensure that the normal world is using the sequence number specified by the TEE. For subsequent packets the sequence number is expected to increment by one each time and the TA verifies this (in case of rollover TA verifies that the packet index is increasing). The TA encrypts the audio data corresponding to the set of references in the RTP payload (further discussed in Section III-E). Once the encrypted payload is in place in the packet, the TA computes the HMAC and returns the result to the normal world. The SRTP library can then continue with sending the packet out. On the receiver device, the reverse steps happen. The Playback Crypto TA is given the received packet. The TA verifies the HMAC. If the verification fails, the TA informs the normal world. Otherwise, the TA calculates the counter from the sequence number and SSRC in the packet, the salt (from call setup) and the block counter. The TA decrypts the payload, replaces it with a reference and returns the result to the normal world. The SRTP layer forwards the packet containing the reference to the RTP handling layer to continue playback.

E. Reference Data Management

In this section we explain how TruzCall manages the plain text audio data in the TEE memory, and how it translates references to audio data or generates references for audio data. To manage audio data in the TEE, we utilize ring buffers similar to the normal world. Android follows the standard practice of using FIFO buffers to manage audio data. This is done in the AudioFlinger [46] and in Linux's ALSA driver [47]. We use two ring buffers inside TEE's SDP memory, one for record data and other for playback data.

1) Data Management for Record: The VoIP app buffers a certain number of bytes before constructing an RTP packet. The native audio framework may send multiple requests to TEE to provide the required number of bytes to the app. As shown in Figure 6, each time the native audio framework requests a certain number of bytes, the Record Data TA moves the requested (or available) number of bytes from the ring buffer to a separate cache in the SDP memory. The data in the ring buffer is provided by the Simulation TEE Bridge which gets it from our simulation hardware setup. The cache is necessary because by the time the SRTP layer invokes TEE, the data corresponding to the reference(s) may have been overwritten in the ring buffer (the overwriting behavior is similar to how audio drivers in Linux buffer

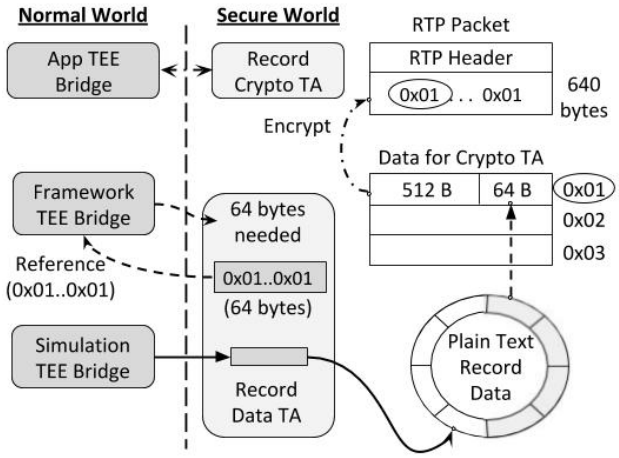

Fig. 6. Reference Data Management for Record

data [48]). The TEE needs to give the audio framework a reference corresponding to the audio data moved into the cache. As discussed in Section III-D, when the SRTP library invokes the Record Crypto TA, it needs to encrypt the RTP payload, for which it needs a buffer containing all the audio data corresponding to the set of references.

One of the design constraints of TruzCall is to reduce latency. A simple implementation would be to lookup the audio data corresponding to each reference, assemble the buffer and then proceed to encryption and HMAC. This would add latency because of the time spent in the TEE to assemble the buffer before actually starting the encryption (data corresponding to each reference would require two memcpy() operations). In order to reduce latency we need an approach that uses less time in the TEE to prepare the buffer to be encrypted. When the SRTP library invokes TEE, the buffer corresponding to the RTP payload should already be setup ready to be used. To achieve this, we organize the cache in the SDP memory holding plain text audio in multiples of packetization buffer size (configurable at call initiation). Whenever the native audio framework asks TEE for audio data, before returning a reference we copy the corresponding (or available) bytes of audio into the cache. The cache is always preparing the next buffer for RTP. Since the reference to be returned by TEE is supposed to be the same length as requested (or available) number of bytes, the TA returns a string which is generated by using memset() and repeating the index in the cache (e.g. in Figure 6, string returned is 0x01..0x01). This string is the reference for the normal world. When the SRTP library invokes TEE, the first byte in the RTP payload is the index in the cache for the next buffer to be encrypted. This approach results in one memcpy() needed for data per reference. The difference between two vs one memcpy() may appear insignificant, but it should be noted that TEE invocation happens several times per second during a call, and all that latency adds up to affect voice quality.

2) Data Management for Playback: Similar to how a cache is maintained to prepare RTP payload for encryption,

a separate cache is used in the SDP memory to keep the playback RTP payload decrypted in the TEE. As shown in Figure 7, when the SRTP library receives a packet from the network, it forwards it to the Playback Crypto TA for HMAC verification and decryption. Once decrypted the buffer is added to the next index in the cache. The reference returned to the SRTP library is of the same length as the RTP payload, and is assigned the cache index value (using memset()). When the native audio framework requests playback data from the app, the size can vary (discussed in Section III-C). As the Playback Data TA gets requests to play audio, it copies data from the cache index into the playback ring buffer and keeps track of how much data has been played from the index. Once a certain cache index is exhausted the next one is used (as specified by the passed reference). Figure 7 shows a case when the playback request spans audio data from two indexes 0x01 and 0x02 (the passed reference string had 0x01 40 times and 0x02 460 times). The data in the playback ring buffer is played out by the Simulation TEE Bridge.

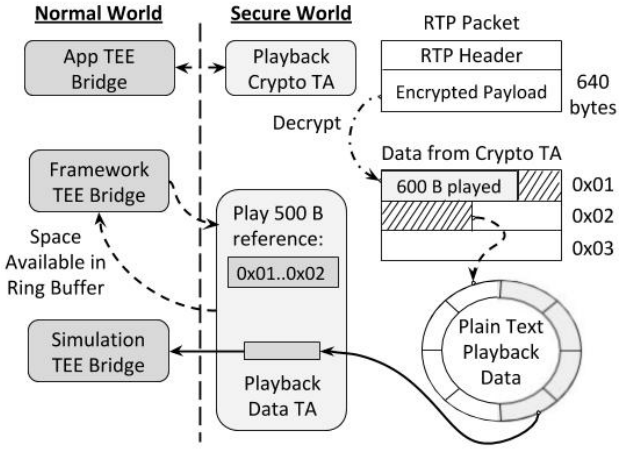

Fig. 7. Reference Data Management for Playback

A question that can be asked is why can't one just make the ring buffers large enough so that enough data is always available for record or enough space is available for playback ? TEE environments operate with limited amount of memory. In a production environment, severals TAs can be present in the TEE for various use cases, which can reduce the amount of memory available. In addition, the amount of audio data available in TEE at any time depends on the type of audio hardware and the type of interface used. Also, we use 640 bytes to organize the cache based on the packet size used by Linphone. A different VoIP app may ask more or less bytes per packet. The goal of TruzCall's design is to be generic such that it can help reduce latency in different scenarios for VoIP.

IV. Security Analysis

In this section, we present the security analysis of the TruzCall design. We assume side channel attacks, hardware related attacks and attacks related to VoIP network are out of scope. The goal of the malicious normal-world OS is to obtain the plain text audio for a VoIP call. The OS can attempt to do this at various phases of the VoIP call. In each phase, the described scenarios won't work because of the various properties of the design. The OS may try to obtain the secret phrase typed by the user. During the secret phrase entry, TEE controls the UI and input, and user is informed of this using a secure LED. This has been discussed in existing work [15], [36]–[38]. The OS may try to fool the user that the secure call is initiated, but not give control to the TEE and mimic the secure UI for call initiation as shown by the TEE. The OS will not be able to access the secure LED, which is used to inform the user whether the audio peripherals are indeed in control of the TEE. Due to this, the OS cannot fool the user regarding secure call initiation. The OS may try to obtain the master key. The OS won't know the master key calculated during call initiation as the secret phrase used for its calculation is protected and the TEE gives the normal world only a reference to the master key. The encryption and decryption for SRTP in the TEE uses AES-CTR, which is a stream cipher and can be subjected to various attacks [49], including keystream reuse, bit-flipping and chosen-IV attacks. The normal-world OS can influence the counter because the sequence number is sent by the normal world. If the same key and counter are used, the XOR of cipher text can give XOR of plain text. In TruzCall, the counter is not allowed to be repeated. As mentioned in Section III-D, the counter calculated in TEE is derived from packet index, which is derived from sequence number and rollover counter. The TA verifies that the packet index is increasing each time. Bit-flipping requires knowledge of part of the plain text. The normal-world OS does not have access to the plain text audio. Chosen-IV attack relies on choosing certain IVs and analyzing the generated keystreams. The normal-world OS cannot observe the keystream as it resides in TEE memory.

As mentioned in Section III-D, the SRTP library does replay detection. It does this based on packet index and uses a replay list & window to detect replay attacks. We do not move this functionality into the TEE. The normal-world OS may attempt to replay received packets. This is countered as the TA checks to ensure that the packet index handled is always increasing. The normal-world OS can attempt to replay voice payload for outgoing packets by holding onto references seen before. The size of the audio cache in the SDP memory provides a brief time gap before same index is used again due to index roll over. The TA zeros out the memory once the data at a certain index has been used. Reuse of an older index won't result in re-sending of data.

V. Simulation Test Environment

In this section we discuss the simulation based approach used for building the hardware environment for testing TruzCall. As far as we know, this is the first time a simulation based approach has been applied to the area of TEE research. Similar approach is used in other areas like embedded system testing where it is referred to as hardware-in-the-loop simulation [50]. In TEE research one often needs to interface hardware periph-

erals with the TEE OS. This task can be challenging for non-hardware experts, depending on the available support from the hardware vendor and available driver support from the TEE OS vendor. In our prototype, we use the Hikey 620 development board [51]. The OP-TEE OS provides different driver support [52] for different boards, and for the Hikey it provides the UART driver. Common audio hardware [53] used in prototyping rely on I2S for which no driver is provided by OP-TEE. Given the lack of support from the hardware vendor and the community, with limited resources it would not be efficient to develop a board specific driver stack to make I2S work on Hikey. The board has USB interface available, but using it with TEE would require introducing the USB stack in the TEE OS. UART could be used to get audio into TEE, but it would require audio compression techniques like DPCM [54] and ADPCM [55] with sample rate limited by the UART bandwidth. To build the hardware test environment to demonstrate TruzCall, we want to use an approach that does not depend on the support from the hardware vendor, the driver support available from the TEE OS vendor, and can best retain the quality of data needed for the experiment. To meet this requirement, we introduce a simulation based testing environment, in which we use a real phone to provide the audio hardware and stream audio data from the phone to the TA in the TEE OS via our Simulation TEE bridge. The bridge is considered part of the secure world.

To setup the environment, we use a Nexus 5X phone with each of two Hikey 620 development boards (two ends of VoIP call during evaluation). Both Hikeys run Android OS version 7.1.2 in the normal world and OP-TEE OS version 2.5 in the secure world. The Hikeys use USB ethernet adapters for internet access. Both Hikeys are connected to the same switch and can reach the internet via a connected router. The internet access is needed because the VoIP app needs to connect to its server for call initiation. We use the open source Linphone app [12] (version v3.3.2). In Figures 6 and 7, we showed how the Simulation TEE bridge provides data for record and gets data for playback. The bridge communicates with an Android app on the Nexus phone over TCP to send / receive audio data. The combination of the bridge and the external phone replaces the need for drivers inside the TEE OS for audio hardware access by the TAs. The simulation bridge does send/receive plain text audio between the external phone and the TEE Data TAs, but this component is used for easier prototyping. If a vendor adopted TruzCall, the simulation bridge would no longer be needed as TAs would directly use audio drivers provided by the vendor in TEE. In that case user's conversation plain text audio would never be returned to the normal world. The app on the Nexus phone records and plays audio in 16-bit PCM format (mono) at a sample rate of 16 KHz. The app continuously sends recorded audio to the bridge which makes it available to the ring buffer for record data in the TEE. The bridge periodically gets available audio in the TEE playback ring buffer and sends it to the app for playback on the phone. Although the simulation environment provides the benefit of making hardware setup easier for prototyping, it does add latency because of the time taken to send/receive audio data to/from the external phone.

VI. Evaluation

In this section we discuss the evaluation done for TruzCall using the Linphone app and our simulation test environment. From the point a call is established TruzCall uses existing VoIP protocols. Any additional delay added is on the end device. The design doesn't change the delay on the network. The evaluation focuses on measuring modifications for secure VoIP on the end device. Network delay can vary as it does in everyday usage of VoIP. Since both Hikey boards act as sender and receiver during a VoIP call, we report metrics collected on one of the devices. The reported metrics are based on three VoIP app configurations: (1) C-Off, (2) C-On and (3) Secure. In the first two cases, the VoIP app does not use TruzCall, but the additional audio computation stages are turned off vs on respectively. In the third case, the VoIP app uses TruzCall and the additional stages are turned off. Comparing the non-secure cases with USB audio (hardware attached to normal world) against secure case with simulation setup would be unfair because the simulation would add some latency. In all cases, we use the simulation environment for audio data. In the non-secure cases, audio data obtained by the Simulation Bridge is passed directly to the native audio framework.

A. Performance

In this section, we compare the impact of TruzCall on the time taken during a VoIP call. TruzCall impacts the amount of time the app uses between getting audio data and sending out a packet (and vice versa for received audio). We report the time taken in the SRTP layer as that involves the use of TEE in the secure case. Once a call is established, the time taken for a spoken word to be heard at the other end of the call will change when TruzCall is used (end-to-end time). We also report the time it takes the app to get audio data for record or send audio data for playback using our simulation setup. We focus here on the time taken between native audio framework and the Simulation Bridge (we exclude the time taken by the daemon to send/receive audio data to/from the external phone over the network). The reported results are the average from 20 measurements. The overhead added in SRTP is 0.48 ms for outgoing packets and 0.54 ms for incoming packets. This has little impact on overall performance as TruzCall adds a quarter second average overhead compared to C-off for end-to-end time during a call. The end-to-end time for C-on is higher because it uses additional computation stages in the VoIP pipeline, which are not used by the secure case.

	Non-Secure	Secure
SRTP Time per Outgoing packet (ms)	0.16	0.64
SRTP Time per Incoming packet (ms)	0.12	0.66
End-to-End Time (seconds)	C-off: 4.27 C-on: 5.6	4.51
Audio Input Time (ms / KB)	16.95	18.45
Audio Output Time (ms / KB)	14.31	32.96

B. VoIP Quality

VoIP call quality can be affected by several factors [44], [56], [57], including packet loss, voice quality, delay and delay variation (jitter). For VoIP, 1-2.5% of packet loss is considered acceptable [58]. We include measurements for 2% packet loss in the test for voice quality. To test packet loss, we use the Linux `iptables` tool. Mean opinion score (`MOS`) is a well-known measure of voice quality [59]. It is a subjective test wherein participants judge the quality of a voice transmission system by rating the voice quality on a scale of 1 to 5. We used Amazon Mechanical Turk [60] to gather the data from 60 participants (US-based). We provided audio recordings from calls using non-secure (C-on) and secure cases. The recordings were audio data received on one of the Nexus phones in our simulation setup. We also asked the participants to answer a question based on each recording to check if they understand the content and to ensure survey quality. The survey and the recordings can be found at [61]–[65]. We report the MOS scores and percentage of participants that answered the questions correctly. The MOS scores were expected to be low because of the additional latency from the simulation setup. MOS scores provide user perceived quality difference between the non-secure and secure cases. The participants were able to comprehend the contents of the secure call at least 81% of the time. This result would be better if an audio driver was available in the `TEE`, as simulation makes prototyping easier but adds latency during testing.

		C-on	Secure
MOS (no packet loss)		2.1	1.3
MOS (2% packet loss)		2.0	1.2
Correct Answer (no loss)		95%	95%
Correct Answer (2% loss)		98%	81%
	C-off	C-on	Secure
JBM (ms)	55	211	207
IAJ (average)	26.41	27.38	26.12
IAJ (median)	26.5	27.3	26.6
JB (ms)	67.5	89.06	79.26

There are several types of delay [44], [66] involved in VoIP. In our evaluation of TruzCall, the relevant delays include processing delay and packetization delay. Processing delay relates to the audio codec algorithm which is used for compression. Since we disabled additional audio computation stages in the secure case, we do not measure the delay incurred for this stage. The packetization delay relates to buffering of audio by the `RTP` library before sending out a packet. TruzCall does not change the amount of audio buffered for each packet. We measure the time taken to prepare each `RTP` packet before it is handed off to the `SRTP` layer. The average time taken for each case was as follows: (1) C-On: 19.98 ms, (2) C-off: 18.08 ms, (3) Secure: 21.23 ms. During a VoIP call, RTP packets may arrive out of sequence and/or at varying intervals [56], [67], [68]. VoIP apps like Linphone use a jitter buffer [69] to hold incoming packets before the corresponding audio is played out, which adds some delay. Since TruzCall uses TEE at different layers of the VoIP stack, TEE invocations can add timing irregularity and contribute to jitter. We report three metrics related to jitter: (1) JBM: maximum jitter buffer delay obtained from RTCP XR [70], (2) IAJ: inter-arrival jitter obtained from RTCP SR [25], (3) JB: jitter buffer size. Metric (1) is the maximum delay applied to received packets by the jitter buffer. Metric (2) is mean deviation of the difference in packet spacing at the receiver compared to the sender for a pair of packets (we report the average and median). For metric (3), we report the average value. The values correspond to a 15 minute call. The secure case adds average 1.25 ms overhead in RTP packet construction, but adds less jitter compared to C-on, due to less number of stages in the VoIP pipeline. When compared to equal number of pipeline stages in C-off, secure case does add jitter overhead, but still results in a quarter second average end-to-end time overhead.

VII. Related Work

The closest work to TruzCall is [11] and we have presented a comparison against this work in Section II-C. In this section we discuss related work in the area of secure calling and `TEE`.

1) Secure Calling: Balasubramaniyan et al. [71] apply machine learning on audio features to identify a call source. Shirvanian et al. [72] investigate the security and usability of crypto phones and present an improved crypto phone in [73] by removing human user from the loop for checksum comparison. Reaves et al. [74] propose an authentication protocol over the audio channel. Marx et al. [75] apply VoIP signatures for non-repudiation and integrity protection. Rohloff et al. [76] apply homomorphic encryption for secure VoIP teleconferencing. Ashok et al. [77] apply ECDH algorithm to strengthen the key for VoIP encryption and decryption. Heuser et al. [78] protect metadata by providing call unlinkability. Kohls et al. [79] apply steganography to hide information in VoIP communication. In all these works, it is assumed that the device being used to make the call is trusted, while in TruzCall it is assumed the OS of the user device is compromised.

2) Trusted Execution Environment: SeCloak [13] allows the user to turn device peripherals on or off using `TEE`. TrustOTP [80] integrates hardware-based one-time password solution with TrustZone. AdAttester [81] uses TrustZone to provide attested click and display for android advertisements. vTZ [82] virtualizes TrustZone in the VM. CacheKit [83] enhances TrustZone's memory privacy against physical attack. Cho et al. [84] show cross-world covert channels on `TrustZone` using Prime+Count technique. Zhang et al. [85] show how to steal secrets from `TEE` using timing-based cache side-channel.

VIII. Summary

In this paper, we proposed a design to allow a user to make a secure end-to-end protected VoIP call from a compromised mobile phone. We implemented our design by modifying Android OS and `OP-TEE` OS. We tested the design using the open source app `Linphone` on the `TrustZone`-enabled Hikey development board utilizing a simulation based test environment. We evaluated the design's performance and VoIP quality during a real call. This project was supported in part by the NSF grant 1718086.

REFERENCES

[1] "Signal Messenger," https://www.signal.org/, 2013, [Accessed: Feb 2019].

[2] "Whatsapp - Simple, Secure, Reliable messaging." https://www.whatsapp.com/, 2009, [Accessed: Feb 2019].

[3] "IDC - Smartphone Market Share," https://www.idc.com/promo/smartphone-market-share/os, 2018, [Accessed: Feb 2019].

[4] "GlobalStats Stat Counter - Mobile Operating System Market Share Worldwide," http://gs.statcounter.com/os-market-share/mobile/worldwide, 2019.

[5] "Google Android Vulnerability Statistics," https://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224, 2018, [Accessed: Feb 2019].

[6] "Android Security Bulletins," https://source.android.com/security/bulletin/, 2019, [Accessed: Feb 2019].

[7] "Google Android Vulnerability Statistics," https://bgr.com/2019/02/07/android-security-update-one-png-image-file-can-compromise-your-phone/, 2019, [Accessed: Feb 28, 2019].

[8] "Hacking Group Spies on Android Users in India Using PoriewSpy," https://blog.trendmicro.com/trendlabs-security-intelligence/hacking-group-spies-android-users-india-using-poriewspy/, 2018, [Accessed: Feb 28, 2019].

[9] "An inside look at nation-state cyber surveillance programs," https://blog.lookout.com/shmoocon-2019, 2019, [Accessed: Feb 28, 2019].

[10] "Datagram Transport Layer Security," https://tools.ietf.org/html/rfc6347, 2012.

[11] M. Ender, G. Duppmann, A. Wild, T. Poppelmann, and T. Guneysu, "A Hardware-Assisted Proof-of-Concept for Secure VoIP Clients on Untrusted Operating Systems," in *Proceedings of 2014 International Conference on ReConFigurable Computing and FPGAs*, ser. ReConFig' 14, Cancun, Mexico, Dec 8-10 2014. [Online]. Available: https://doi.org/10.1109/ReConFig.2014.7032489

[12] "Linphone - open source VoIP project," https://www.linphone.org/, 2001, [Accessed: Feb 2019].

[13] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee, "Secloak: Arm trustzone-based mobile peripheral control," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '18. New York, NY, USA: ACM, 2018, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/3210240.3210334

[14] "Android Media Framework Hardening," https://source.android.com/devices/media/framework-hardening, 2019.

[15] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, "Truz-droid: Integrating trustzone with mobile operating system," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '18. New York, NY, USA: ACM, 2018, pp. 14–27. [Online]. Available: http://doi.acm.org/10.1145/3210240.3210338

[16] "Android DRM," https://source.android.com/devices/drm, 2019.

[17] "Protecting your premium HD content with Widevine Digital rights management (DRM) on Inforce platforms," https://web.archive.org/web/20190805191938/https://www.inforcecomputing.com/protecting_with_premium_hd_content/, 2019.

[18] "Secure Playback using OP-TEE," https://web.archive.org/web/20190723021453/http://hkg15.pathable.com/static/attachments/112702/1424379017.pdf?1424379017, 2015.

[19] "Secure Data Path with OP-TEE," https://web.archive.org/web/20190723021519/http://s3.amazonaws.com/connect.linaro.org/bud17/Presentations/BUD17-400%20-%20Secure%20Data%20Path%20with%20OPTEE.pdf, 2017.

[20] Wikipedia, "Comparison of voip software — Wikipedia, the free encyclopedia," "https://en.wikipedia.org/wiki/Comparison_of_VoIP_software", 2019.

[21] "The Secure Real-time Transport Protocol (SRTP)," https://tools.ietf.org/html/rfc3711, 2004.

[22] "SIP: Session Initiation Protocol," https://tools.ietf.org/html/rfc3261, 2002.

[23] "From SIP to RTP - Overview," https://www.informaticapressapochista.com/asterisk/from-sip-to-rtp-part-1/, 2012, [Accessed: Feb 28, 2019].

[24] "SDP: Session Description Protocol," https://tools.ietf.org/html/rfc2327, 1998.

[25] "RTP: A Transport Protocol for Real-Time Applications," https://tools.ietf.org/html/rfc3550, 2003.

[26] "ZRTP: Media Path Key Agreement for Unicast Secure RTP," https://tools.ietf.org/html/rfc6189, 2011.

[27] "oRTP in Linphone architecture," https://www.linphone.org/technical-corner/ortp, 2019, [Accessed: April 2, 2019].

[28] "Belle-sip in Linphone architecture," http://linphone.org/technical-corner/belle-sip, 2019, [Accessed: April 2, 2019].

[29] "Mediastreamer2 in Linphone architecture," http://linphone.org/technical-corner/mediastreamer2, 2019, [Accessed: April 2, 2019].

[30] "Android Audio Overview," https://source.android.com/devices/audio/, 2019.

[31] "Android Audio Terminology," https://source.android.com/devices/audio/terminology, 2019.

[32] "Sample Rate Conversion," https://source.android.com/devices/audio/src, 2019.

[33] "Meanings of TA FLAGS," https://web.archive.org/web/20190312204816/https://github.com/OP-TEE/optee_os/issues/1590, 2017.

[34] "Secure Data Path with OPTEE," https://www.slideshare.net/linaroorg/bud17400-secure-data-path-with-optee, 2017.

[35] "OPTEE OS - Hikey Platform Memory Config," https://github.com/OP-TEE/optee_os/blob/master/core/arch/arm/plat-hikey/platform_config.h, 2018.

[36] 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*, ser. APSys '14. New York, NY, USA: ACM, 2014, pp. 8:1–8:7. [Online]. Available: http://doi.acm.org/10.1145/2637166.2637225

[37] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena, "DroidVault: A Trusted Data Vault for Android Devices," in *Proceedings of the 2014 19th International Conference on Engineering of Complex Computer Systems*, ser. ICECCS' 14, Tianjin, China, Aug 4-7 2014. [Online]. Available: https://doi.org/10.1109/ICECCS.2014.13

[38] K. Ying, P. Thavai, and W. Du, "Truz-view: Developing trustzone user interface for mobile os using delegation integration model," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '19. New York, NY, USA: ACM, 2019, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3292006.3300035

[39] "Secure storage in OP-TEE," https://www.slideshare.net/linaroorg/sfo15503-secure-storage-in-optee, 2015.

[40] "AndroidXref Android 7.1.2 - AudioRecord.h," http://androidxref.com/7.1.2_r36/xref/frameworks/av/include/media/AudioRecord.h, 2017.

[41] "AndroidXref Android 7.1.2 - AudioTrack.h," http://androidxref.com/7.1.2_r36/xref/frameworks/av/include/media/AudioTrack.h, 2017.

[42] "AndroidXref Android 7.1.2 - Thread.h," http://androidxref.com/7.1.2_r36/xref/system/core/include/utils/Thread.h, 2017.

[43] "How to write a native thread and how to use it," http://shooting.logdown.com/posts/247468-android-native-thread, 2014.

[44] "How To Measure VoIP Quality And Jitter," https://route-test.com/voip-quality-delay-jitter-measurement/, 2019, [Accessed: March 23, 2019].

[45] "SRTP Crypto - aes icm," https://github.com/Linphone-sync/srtp/blob/master/crypto/cipher/aes_icm.c, 2014.

[46] "In-depth understanding of the Android audio framework," https://blog.csdn.net/ch97ckd/article/details/78641457, 2017, [Accessed: Feb 28, 2019].

[47] "ALSA driver–HW Buffer," http://www.echojb.com/hardware/2016/12/21/283392.html, 2016.

[48] "Introduction to Sound Programming with ALSA," https://web.archive.org/web/20190421034814/https://www.linuxjournal.com/article/6735, 2004.

[49] Wikipedia, "Stream cipher attacks — Wikipedia, the free encyclopedia," "https://en.wikipedia.org/wiki/Stream_cipher_attacks", 2019.

[50] "What Is Hardware-in-the-Loop?" http://www.ni.com/en-us/innovations/white-papers/17/what-is-hardware-in-the-loop-.html, 2006.

[51] "Hikey (LeMaker)," https://www.96boards.org/product/hikey/, 2019, [Accessed: April 15, 2019].

[52] "OPTEE OS Drivers (Github)," https://github.com/OP-TEE/optee_os/tree/master/core/drivers, 2019.

[53] "I2S Amplifier Breakout," https://www.digikey.com/short/pzp5n4, 2019, [Accessed: April 15, 2019].

[54] Wikipedia, "Differential pulse-code modulation — Wikipedia, the free encyclopedia," "https://en.wikipedia.org/wiki/Differential_pulse-code_modulation", 2018.

[55] ——, "Adaptive differential pulse-code modulation — Wikipedia, the free encyclopedia," "https://en.wikipedia.org/wiki/Adaptive_differential_pulse-code_modulation", 2018.

[56] "RTP, Jitter and audio quality in VoIP," https://kb.smartvox.co.uk/voip-sip/rtp-jitter-audio-quality-voip/, 2012.

[57] "Mean Opinion Score for VoIP Testing," https://www.voipmechanic.com/mos-mean-opinion-score.htm, 2019, [Accessed: March 23, 2019].

[58] Wikipedia, "Packet loss — Wikipedia, the free encyclopedia," "https://en.wikipedia.org/wiki/Packet_loss", 2019.

[59] "Measuring Voice Quality," https://route-test.com/mean-opinion-score-mos-measure-voice-quality-voip/, 2019, [Accessed: April 7, 2019].

[60] "Amazon Mechanical Turk," https://www.mturk.com/, 2019, [Accessed: April 7, 2019].

[61] "TruzCall voice quality survey," https://drive.google.com/file/d/1HdEPQqIUBORvzESjy2zKUcW-6XnUV3Wr/view?usp=sharing, 2019.

[62] "TruzCall voice recording - non secure 0 percent loss," https://drive.google.com/file/d/1Fz8pR-FjLl4ug5jrt0BZ2KU2cJ5JKUeT/view?usp=sharing, 2019.

[63] "TruzCall voice recording - non secure 2 percent loss," https://drive.google.com/file/d/1pKitr_GN19tP46pAEORQfQ9N8E8vyl0x/view?usp=sharing, 2019.

[64] "TruzCall voice recording - secure 0 percent loss," https://drive.google.com/file/d/1kf77uWONtPZzkMzoMf9eIWCCvQf3P0T_/view?usp=sharing, 2019.

[65] "TruzCall voice recording - secure 2 percent loss," https://drive.google.com/file/d/1yy-ZzMecRMfbW6ho6xDV9xtye5Regy9L/view?usp=sharing, 2019.

[66] "Understanding Delay in Packet Voice Networks," https://www.cisco.com/c/en/us/support/docs/voice/voice-quality/5125-delay-details.html, 2006.

[67] "VoIP Basics: About Jitter," https://web.archive.org/save/http://toncar.cz/Tutorials/VoIP/VoIP_Basics_Jitter.html, 2019, [Accessed: April 7, 2019].

[68] "RTP, RTCP and Jitter Buffer," https://blog.wildix.com/rtp-rtcp-jitter-buffer/, 2018.

[69] "Linphone - Introduction to our new RTP adaptive jitter buffer algorithm," https://www.linphone.org/news/introduction-our-new-rtp-adaptive-jitter-buffer-algorithm, 2019, [Accessed: April 7, 2019].

[70] "RTP Control Protocol Extended Reports (RTCP XR)," https://tools.ietf.org/html/rfc3611, 2003.

[71] V. A. Balasubramaniyan, A. Poonawalla, M. Ahamad, M. T. Hunter, and P. Traynor, "Pindr0p: Using single-ended audio features to determine call provenance," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866320

[72] M. Shirvanian and N. Saxena, "On the security and usability of crypto phones," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA: ACM, 2015, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/2818000.2818007

[73] ——, "Cccp: Closed caption crypto phones to resist mitm attacks, human errors and click-through," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 1329–1342. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134013

[74] B. Reaves, L. Blue, and P. Traynor, "Authloop: End-to-end cryptographic authentication for telephony over voice channels," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 963–978. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/reaves

[75] R. Marx, N. Kuntze, and H. Lauer, "Bringing voip signatures to mobile devices," in *Proceedings of Principles, Systems and Applications on IP Telecommunications*, ser. IPTComm '13. New York, NY, USA: ACM, 2013, pp. 3:1–3:7. [Online]. Available: http://doi.acm.org/10.1145/2589649.2554669

[76] K. Rohloff, D. B. Cousins, and D. Sumorok, "Scalable, practical voip teleconferencing with end-to-end homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 5, pp. 1031–1041, May 2017. [Online]. Available: https://doi.org/10.1109/TIFS.2016.2639340

[77] Ashok. S, Arjun. A, and T. Subashri, "Dynamic ecdh mechanism for enhancing privacy of voice calls on mobile phones over voip server," in *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*, May 2014, pp. 1179–1184. [Online]. Available: https://doi.org/10.1109/ICACCCT.2014.7019284

[78] S. Heuser, B. Reaves, P. K. Pendyala, H. Carter, A. Dmitrienko, W. Enck, N. Kiyavash, A.-R. Sadeghi, and P. Traynor, "Phonion: Practical Protection of Metadata in Telephony Networks," in *Proceedings on Privacy Enhancing Technologies*, Minneapolis, MN, USA, Jul 18 - 21 2017. [Online]. Available: https://petsymposium.org/2017/papers/issue1/paper27-2017-1-source.pdf

[79] K. Kohls, T. Holz, D. Kolossa, and C. Pöpper, "Skypeline: Robust hidden data transmission for voip," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: ACM, 2016, pp. 877–888. [Online]. Available: http://doi.acm.org/10.1145/2897845.2897913

[80] H. Sun, K. Sun, Y. Wang, and J. Jing, "Trustotp: Transforming smartphones into secure one-time password tokens," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 976–988. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813692

[81] W. Li, H. Li, H. Chen, and Y. Xia, "Adattester: Secure online mobile advertisement attestation using trustzone," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '15. New York, NY, USA: ACM, 2015, pp. 75–88. [Online]. Available: http://doi.acm.org/10.1145/2742647.2742676

[82] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing ARM trustzone," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 541–556. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua

[83] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou, "Cachekit: Evading memory introspection using cache incoherence," in *2016 IEEE European Symposium on Security and Privacy*, March 2016, pp. 337–352. [Online]. Available: https://doi.org/10.1109/EuroSP.2016.34

[84] H. Cho, P. Zhang, D. Kim, J. Park, C.-H. Lee, Z. Zhao, A. Doupé, and G.-J. Ahn, "Prime+count: Novel cross-world covert channels on arm trustzone," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: ACM, 2018, pp. 441–452. [Online]. Available: http://doi.acm.org/10.1145/3274694.3274704

[85] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Trusense: Information leakage from trustzone," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, April 2018, pp. 1097–1105. [Online]. Available: https://doi.org/10.1109/INFOCOM.2018.8486293