Wink: Deniable Secure Messaging

Anrin Chakraborti, *Duke University;*
Darius Suciu and Radu Sion, *Stony Brook University*

https://www.usenix.org/conference/usenixsecurity23/presentation/chakraborti-wink

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the
32nd USENIX Security Symposium
is sponsored by USENIX.

Wink: Deniable Secure Messaging

Anrin Chakraborti [*]
Duke University

Darius Suciu [*]
Stony Brook University

Radu Sion
Stony Brook University

Abstract

End-to-end encrypted (E2EE) messaging is an essential first step in providing message confidentiality. Unfortunately, all security guarantees of end-to-end encryption are lost when keys or plaintext are disclosed, either due to device compromise or coercion by powerful adversaries. This work introduces Wink, the first *plausibly-deniable messaging* system protecting message confidentiality from partial device compromise and compelled key disclosure. Wink can surreptitiously inject hidden messages in standard random coins, e.g., in salts, IVs, used by existing E2EE protocols. It does so as part of legitimate secure cryptographic functionality deployed inside the widely-available trusted execution environment (TEE) TrustZone. This results in hidden communication using virtually unchanged *existing E2EE messaging apps*, as well as strong plausible deniability. Wink has been demonstrated with multiple existing E2EE applications (including Telegram and Signal) with minimal (external) instrumentation, negligible overheads, and crucially, without changing on-wire message formats.

1 Introduction

Secure messaging with end-to-end encryption has become the standard mode of communication between trusted peers. Almost all messaging apps today support *end-to-end encryption* for ensuring message confidentiality in the presence of compromised or malicious intermediate nodes; the messages are available in plaintext only on the user devices and encrypted when in transit with user-chosen keys.

However, if the encryption keys are eventually leaked to an adversary, message confidentiality is automatically lost. Unintended disclosure of cryptographic keys is not uncommon: (i) commercially-available devices are often vulnerable to malware including backdoors into widely-deployed secure apps [24, 26, 29], and ii) users can be coerced to hand over cryptographic information (e.g., keys) [18, 39, 41, 57].

Achieving truly private communication between trusted parties requires stronger security guarantees than what standard end-to-end encryption can offer. While existing solutions can solve some facets of the problem, they ultimately fall short of addressing the problem in its entirety. Some notable candidates include off-the-record messaging (OTR) [7], deniable encryption [10, 11, 48], ephemeral messages, and network stegnography [30]. Most of these techniques either fail when the end devices are also compromised by the adversary (in addition to the network monitoring), or/and are impractical for real-world applications (see Section 2 for more detailed comparisons). However, perhaps more importantly, due to their impracticality (and in some cases incompatibility with other applications), most of these techniques are not ubiquitously deployed; thus the mere presence of an application implementing these techniques can indicate to an adversary that the application is being specifically used to hide information. In light of these observations, this paper asks

> *Can we achieve end-to-end encrypted communication between trusted peers when the adversary can obtain access (e.g., through adversary-controlled malware) into the user device, can read on-wire transcripts, and can compel the users to reveal corresponding encryption keys (and other metadata)?*

At first glance, it may appear that efforts to answer this question are futile as the adversary controls the software on the compromised device and can therefore observe messages in the plain by monitoring I/O channels. In fact, even without any sophisticated mechanisms, if the adversary is aware that a message has been sent/received (e.g., by monitoring memory usage, network I/O), then breaching confidentiality by compelling the user to hand over the keys is trivial. Thus, against such a strong adversary the only recourse is to not only hide the message contents but also the fact that a message has been exchanged. Unfortunately, without a safe haven for running some critical parts of software outside the adversary's control (and knowledge), and storing keys for end-to-end encryption, the task is perhaps impossible.

---

[*] These authors contributed equally to this work

This work answers the question affirmatively under a more realistic and wide-spread threat model, where the adversary only partially compromises the device, and shows that a safe haven (ensuring message confidentiality against the adversary) is indeed realizable with the help of a TEE [1]. However, achieving this is not without challenges. First, a TEE is not designed to provide plausible deniability, and so while the TEE may provide confidentiality of information, it will not hide the artifacts of the execution (e.g., the fact that a message has been sent/received). Second, messaging applications are complex, and executing all its constituent tasks inside a TEE is impractical; it also makes the other software in the TEE vulnerable due to the increased trusted computing base (TCB). Finally, building an application for a TEE with the sole purpose of hiding message communication *without any other plausible use-case* is not enough since the possession of such software renders the user a suspect.

This paper presents the design and implementation of Wink, a *plausibly-deniable messaging* framework, which addresses all of these challenges. Wink enables trusted peers to exchange messages but later plausibly deny the exchange to an adversary by providing an alternate, plausible explanation for all the actions observed by the adversary (which includes network traffic and system artifacts). To achieve this, Wink realizes a safe haven by ensuring that the TEE is used minimally (with small TCB) and correctly (with secure communication channels to components outside the TEE). Wink is demonstrably compatible with two widely used E2EE messaging apps, namely Signal and Telegram, with only minimal instrumentation. Importantly, this integration does not alter the on-wire message format native to the messaging app(s).

Wink Cryptographic Library. Wink is a library that performs cryptographic operations at the request of E2EE messaging applications. This includes generating and storing keys, random coins, etc. and encrypting/decrypting messages. Wink runs inside the TEE and integrates with a standard E2EE messaging application that runs outside the TEE. In this way, only the cryptographic operations for the application is performed inside the TEE while all other operations (e.g., networking, UI) runs outside the TEE[2].

This design has two advantages: i) Wink can be seamlessly integrated with any E2EE messaging application without any change in messaging logic or specifications thus removing burden from the app developers, and ii) Wink enables security-conscious device vendors to provide privacy-enhanced communication capabilities by simply including the library in their software distributions. Consequently, having Wink on the device is not enough to raise suspicion unlike alternatives, e.g., deniable encryption, which have no other purpose for deployment other than hiding communication.

In addition to trusted cryptography for E2EE messaging applications, Wink also enables "hidden messaging"; hidden messages sent using Wink are later plausibly denied to the adversary. For this, a message transcript containing hidden messages is shown to be equivalent (to the adversary) to a message transcript that contains only messages sent through the E2EE messaging app which are revealed to the adversary. In this way, the adversary verifies that the observed message transcript indeed corresponds to user inputs into the messaging app (which may be adversary-controlled or monitored). This provides stronger deniability than ephemeral messages where no evidence can be provided after the fact.

Hidden Messaging over Public Channel. Wink establishes a "hidden communication channel" over the public communication channel that is already established by the E2EE messaging application running in the potentially compromised environment outside the TEE. To achieve this, Wink injects (encrypted) hidden messages in the random coins (e.g., salts, IVs, etc.) inherently used in the public messaging cryptography. Specifically, encrypted hidden messages (indistinguishable from random) are used as random coins and then sent over the public channel masqueraded as randomly generated cryptographic metadata for the public messages. When coerced, users can hand over the encrypted public messages (along with the keys, and metadata) while still being able to plausibly deny the existence of hidden messages.

Crucially, this approach does not impact the functionalities or alter the on-wire format of the messaging app. The E2EE app remains oblivious to the hidden message injection, and an adversary compromising the app by installing spyware or compromising the OS under which it runs has no visibility into the operations pertaining to hidden message injection. In fact, the way hidden messages are communicated over the public channel renders them "almost invisible" to the adversary. That is, the only indication of hidden messaging capabilities of the user's device is the installed Wink library. However, assuming that Wink is ubiquitously deployed, the presence of the Wink library is not sufficient to raise suspicion in itself, as all network traffic, execution transcripts, etc., look identical on all Wink-enabled devices *regardless of whether it is being used for hidden communication*.

Integration with E2EE Apps. A prototype implementation has been instantiated on TrustZone due to its widespread availability in mobile devices. We have integrated Wink with Telegram [53] and Signal [49] messengers. Benchmarks show that overheads including those introduced due to context switches between the messaging application, the OS, and the TEE are in the order of milliseconds. In practice, Wink-introduced delays are negligible compared to the time required for user interaction. Further, injecting hidden messages does not incur communication overheads since the message transcripts do not increase in size over public messaging.

---

[1] Of course, if the adversary is capable of completely compromising a device even if it has advanced hardware-based defenses such as TEE isolation, no safe haven exists. Breaking TEE isolation is however significantly more difficult than user coercion and partial compromise through spyware, etc.

[2] Using TEEs for implementing trusted cryptographic functionalities for untrusted apps is mainstream – e.g, KeyMaster [23] in ARM TrustZone.

2 Related Work

End-to-End Encrypted Messaging. End-to-end encrypted messaging enables parties to communicate with each other via messages that are available in plaintext only on the end devices. When in transit, the messages are encrypted with keys that are available only to the parties. Most messaging apps today support E2EE messaging in addition to guarantees like forward secrecy (see Section 3). While they are designed to provide orthogonal security guarantees, it is worth mentioning that end-to-end encrypted communication does not provide any deniability whatsoever once the encryption keys/metadata are revealed to the adversary.

Ephemeral Messages. Several messaging apps additionally support ephemeral messages where plaintext messages and keys/metadata are deleted periodically. While we can envision a solution for deniable messaging using ephemeral keys wherein keys are deleted after a message is sent/received, this does not provide strong plausible deniability guarantees: i) the messaging app may not be fully trusted/have backdoors, or may not duly delete the messages, ii) despite deletions it is possible for message artifacts and metadata to persist in the system e.g., in filesystem caches, and iii) if the device is already compromised, the adversary can observe messages in plaintext in I/O buffers, etc. This prompts the need for stronger mechanisms that are also resilient to device compromise.

2.1 Plausible Deniability

There is a long line of work building plausibly-deniable systems, mainly in the context of storage systems. This includes steganographic filesystems [1,25,44,46,61] and hidden volumes [6, 13–15]. These solutions only work for data stored at rest and cannot be directly applied for deniable messaging. In the context of messaging applications, there are several flavors of plausible deniability proposed in existing work, and Table 1 compares our model with existing solutions.

Off-the-Record Messaging. Off-the-record (OTR) messaging [7,37] ensures equivocation for messages – if Alice sends a message to Bob, she can later claim to a third-party (e.g., a judge) that Bob fabricated the message. To achieve this, OTR makes message ciphertexts malleable i.e., Bob could have generated a message from a potentially benign message that Alice sent. In this way, OTR protects against network adversaries with access to message transcripts, but if the endpoint devices are compromised, the message contents and the ciphertexts are available to the adversary.

Deniable Encryption. Deniable encryption [10,11,42,48] allows Alice and Bob to replay exactly a transcript of encrypted messages but to end up with a potentially different resulting plaintext than what was originally encrypted to present to an adversary. For example, even if the initial conversation between Alice and Bob was regarding a protest, the conversation transcript can be undetectably "tweaked" to show that the conversation was about attending a sports event. State-of-the-art techniques for deniable encryption rely on techniques such as indistinguishability obfuscation (IO) which are not practically realizable. In theory, it is unclear how IO can be realized securely on a compromised device, and even if this was possible, the adversary may obtain the messages in plaintext from other channels such as memory buffers.

Steganography. Steganographic techniques, particularly designed to hide information in network traffic, e.g., [30], synthetically generate cover traffic to obfuscate the hidden information. For instance, Meteor uses a generative model to create cover text to hide information. While this suffices against a network man-in-the-middle adversary, on a compromised device, the cover text generation process as well as the encoding/decoding process is observable to the adversary. In addition, the techniques proposed in [30] are not efficient enough to be applied in real-time to messaging apps. For instance, Meteor requires roughly 10 minutes to encode a 160-byte message, in addition to requiring up to 7KB of cover traffic. The throughput does not scale to messaging apps.

Wink uses random coins in symmetric key messaging applications to inject encryptions of hidden messages. In theory, this is similar to algorithm substitution attacks (ASA) where an adversary replaces an honest implementation of a cryptographic protocol with a subverted version [2–4]. Such substitutions of symmetric key cryptographic systems have been used before for the purposes of steganography [5]. Wink can similarly substitute the cryptographic system used by the E2EE messaging application to support hidden messaging over a public communication channel. To realize this on a compromised device, where an adversary can inspect the cryptographic algorithms in use, the burden of implementation will be on the device vendors. However, since the cryptographic machinery used by E2EE messaging applications is often highly complex, designing such a system while retaining all privacy guarantees is non-trivial and will impose a significant implementation burden on the device vendors. As we will discuss later, Wink uses a simpler idea for injecting hidden messages that only needs to modify how random coins are selected for a symmetric key encryption scheme.

2.2 TrustZone

ARM processors support running applications in a TEE enforced by TrustZone. Inside this TEE vendors enable protecting security-sensitive applications (TAs) that run isolated from a potentially compromised REE (the Normal World). Previous work presents how TrustZone can be leveraged for protecting payment operations (TrustPay [60]), providing one-time-passwords (TrustOPT [51]) and secure storage (DroidVault [34]). Most of the proposed functionality has materialized in the form of TAs inside commercial TrustZone devices.

Other works have focused on protecting users from Normal World adversaries. For example, TrustDump [52] enables col-

Scheme	Property	Dependency	Network Adversary	Key Disclosure	Device Compr.
E2EE messaging	Enc/dec on end-points, forward sec.	-	✓	✗	✗
E2EE + ephemeral keys	message/keys deleted periodically	-	✓	✓	✗
OTR Messaging	message content malleability	IND-CPA secure malleable enc.	✓	✓	✗
Deniable Encryption	ciphertext decrypts to any plaintext	Indistinguishability Obfuscation	✓	✓	✗
Network Steganography	obfuscation with cover traffic	-	✓	✗	✗
Wink	hidden messaging over public channel	TEE isolation	✓	✓	✓

Table 1: A comparison of different flavors of secure/deniable messaging found in existing work. Property: flavor of deniability provided by the system, Dependency: dependency/assumptions on which the system is built, Net Adversary: secure against a network adversary monitoring communication transcripts, Key Disclosure: secure against key disclosure attacks, Device Comp.: secure against an adversary compromising the end devices. Except for Wink, none of the existing solutions provide any level of deniability on a compromised device.

lecting reliable memory snapshots, TruZ-View [58] provides trusted I/O paths to users, VeriUI [35] provides attested login and SeCloak [33] provides reliable disabling of I/O devices from Secure World. Under TruZ-View, a user interface protects user input confidentiality from the untrusted OS. The VeriUI interface verifies user passwords inside Secure World, while the SeCloak interface enables users to enable and disable I/O devices from inside Secure World. Wink provides a similar user interface for entering hidden messages. Similar to SeCloak, the Wink interface takes over the display framebuffer to display hidden messages to the user. However, Wink goes further and also takes over the touch input in order to protect it against Normal World monitoring. The Wink provided interface enables users to inject hidden messages and read incoming ones, even under Normal World monitoring.

3 Background

3.1 ARM TrustZone

ARM Cortex processors enable building TEEs using the ARM TrustZone security extensions, or *TrustZone*. Under TrustZone, each physical processor core is split into two virtual CPUs. The processor either runs TEE software inside a *Secure World (SW)*, or rich execution environment (REE) software inside the *Normal World*. Switching between the TEE and REE is controlled by a special *Non-Secure (NS) bit*. When NS=0, he core runs TEE code; otherwise, REE software is executed. Physical memory regions and I/O peripherals are also tagged with an NS bit. Those tagged with NS=0 can only be accessed by Secure World, providing TEE exclusive control over the respective memory and I/O devices. The TEE can access all physical memory and dynamically allow or deny REE access to Secure World resources (e.g., peripherals).

The transition of control from Normal World to Secure World is known as a *world switch*. Both the REE and TEE can trigger world switches by issuing Secure Monitor Call (SMC) instructions. These instructions are handled by a *Secure Monitor*, which runs at ARM exception level EL3. Typically, regular applications running inside the REE and Trusted Applications (TAs) running under the TEE communicate with each other through OS-forwarded system calls as SMCs.

3.2 Signal

The Signal protocol is designed with asynchronous messaging in mind – messages can be sent even when the receiver is offline. This requires an intermediate server to facilitate key exchange with the help of information uploaded by each party during initialization. This key information for each user is stored as part of a *prekey bundle*, signed by the user, and is retrieved from the server when another user wants to establish a messaging session. The prekey bundle contains several types of keys which include: i) long-term **identity keys**, ii) medium-term **signed prekeys**, and iii) short-term **ephemeral keys**. Using these keys, a sender can establish a secure end-to-end encrypted messaging session with an offline receiver.

A particularly interesting feature of the Signal Protocol is forward-secrecy. This is provided by *key ratcheting*, first proposed in [7]. Ratcheting derives symmetric per-message keys starting from a chain key e.g., using KDFs. Previous message keys are deleted after a message has been sent/received and encrypted/decrypted. In this way, the confidentiality of past messages remains intact. Further, Signal also provides *future secrecy* by refreshing the chain keys periodically. This ensures that even if an adversary obtains the chain key at some point in time, the messages exchanged in the future remain secret. New chain keys are derived by exchanging *ratcheting keys* potentially with every message.

4 Model

Deployment. In a *plausibly-deniable messaging* (PDM) scenario, a trusted *Sender* wants to send a message(s) to a trusted *Receiver*. The sender and the receiver may later want to plausibly deny the exchange of certain messages to a powerful, coercive adversary (described next). For this purpose, both the sender and receiver use a plausibly-deniable messaging application on their respective devices. It is worth noting that the application does not enable the sender and the receiver to deny that (any) communication has taken place, but rather allows them to plausibly deny the contents of the conversation. Also, deniable encryption and off-the-record messaging, both of which provide some form of equivocation, provide a similar functionality. However, unlike these tools which mainly hide information in transit, PDM also hides all evidence (in network transcripts, systems artifacts, etc.) that a "hidden" message has been exchanged or a special system is being used with the sole purpose of hidden communication.

Plausible Deniability in Practice. For plausible deniability to be effective, we need to make a few general assumptions:

Ubiquitous Deployment: First, if plausible deniability systems remain a niche product, any user using such a system will appear suspect. Therefore, as with most works on plausible deniability, we will assume that Wink is ubiquitously deployed on mobile devices. In this regard, Wink arguably provides a strong use case even when hidden messaging is not the primary purpose, i.e., as a sanitized environment for implementing cryptography of E2EE apps. This is unlike solutions like deniable encryption where the use case beyond obfuscation of exchanged messages is unclear.

Rational Adversary: Second, as will describe later, Wink is deployed as a trusted service inside TrustZone. And since so far trusted services can only be installed by the device vendors, the inclusion of Wink in a system is completely in the hands of the vendor. This has an added benefit: if the vendor buys into the practicality of Wink, all devices by the vendor will have this trusted service. Also, by design, only the vendor is able to extract information from the trusted service. This renders any amount of rubber-hose cryptography – where the user is subject to confinement and torture – useless since the user is unable to extract secret information from Wink. Thus, a rational adversary will need to approach the vendor through proper channels, e.g., with warrants, to access this information. In the recent past, this model has proven to be successful in protecting user privacy from overly intrusive government [29]. Arguably, it is impossible to build a system resilient against adversaries that penalize the user irrationally regardless of evidence.

Deniability is Not Obfuscation: Finally, it is worth noting that plausible deniability is *not security by obfuscation*. The adversary may inspect the user device, analyze the binaries stored, and identify that the software has features that enable deniable communication. The goal is to ensure that the adversary cannot detect (or guess with high probability) that the user ever uses these features. To achieve this, a plausible deniability system provides a plausible, alternative explanation for every action that is observed by the adversary in runtime.

4.1 Threat Model

Capabilities. Since our adversary model is stronger than the adversaries considered in previous works (see Table 1), we need to carefully define the powers of this adversary. In the following, we enlist these capabilities. We consider a coercive adversary who may:

- Partially corrupt the sender's/receiver's devices and observe/record user inputs in plaintext.
- Analyze software binaries, firmware, etc., on the device.
- Observe, capture and store all (encrypted) communication for introspection.
- Obtain access to both the sender and receiver devices (possibly at the same time) and coerce the users to hand over cryptographic keys.

Before discussing what the adversary can compromise on the user's device, it is worth clarifying a few points. First, we allow the adversary to examine the user device, i.e., analyze the binaries, firmware images, etc. In effect, the adversary knows that the Wink library is being used (at least to implement cryptographic operations for E2EE messaging apps), but as we will show, it does not know that the library is being used in some cases for additional hidden messaging. Thus, the goal is not to hide the existence of the library but rather to make the modes of operation indistinguishable.

Second, the adversary may compel the user to hand over keys. As will describe later, Wink has a set of keys that are accessible through user-chosen passwords. Some of these keys are *public* and are handed over to the adversary and some are *hidden* which the user plausibly denies using. Due to the Wink design, the hidden keys are never provided to the user in the plain. The user is only able to use the hidden key through the Wink library based on a password input. When coerced, the user denies having a password that enables the hidden key(s) for hidden messaging operations.

System Setting & Device Compromise. Wink runs in a trusted execution environment (TEE) (namely TrustZone). Therefore, standard TEE-based security assumptions hold:

1. TEE isolation cannot be compromised via software or device hardware vulnerabilities.
2. The software running inside the TEE is trusted.
3. REE cannot overwrite the TEE set device configuration.
4. The device vendor is trusted and there are no backdoors into the TEE.

Attack Vectors. Based on the different privilege levels, there are four broad categories of adversarial attack vectors.

1. *Compelled disclosure:* The adversary coerces the user to hand over message transcripts, which includes the plaintext messages and the corresponding ciphertexts sent and received, and the related cryptographic metadata (e.g., keys, random coins, etc.).
2. *Compromising REE Apps (Normal World App Compromise)*: The adversary installs keyloggers [9], spyware [47] or even introduces backdoors into the messaging app. Using a compromised app, the adversary may try to violate message confidentiality, capture screenshots or key types and monitor the user's actions, with the goal of detecting hidden payloads in the exchanged messages.
3. *Compromising the REE Kernel (Normal World Root Access):* The adversary leverages the compromised kernel to monitor Normal World operations performed by Wink. Monitoring can include tracking TEE entries and exits, timing Wink calls, or detecting I/O resource usage. Only

a (hijacked) REE kernel would be capable of monitoring and reporting such fine-grained TEE operations.

4. *Compromising the TEE (Secure World Compromise):* The adversary leverages TEE software or hardware vulnerabilities to escalate their privileges inside the TEE. Once inside the TEE, the adversaries have direct access to the plausibly-deniable messaging application. Further, TEE compromise might provide adversaries with complete control over the device.

This paper mainly focuses on defending against attacks in categories 1 and 2, which hold even under the assumptions and are easily deployed, i.e. without requiring advanced exploits or alerting the user. Attacks in category 4 are out-of-scope as it immediately invalidates all assumptions and effectively give the adversary complete control over the device. Denial of service attacks e.g., intentionally blocking incoming/outgoing messages, etc. are specifically considered out of scope of the threat model. Note that all plausible deniability systems are vulnerable to such DoS attacks. Designing DoS-resilient plausible deniability schemes is an open problem.

Side Channels. As with any shared-hardware system Wink may be subject to side-channels that undermine certain security guarantees. We have identified two such side-channels that break plausible deniability provided by Wink: i) a timing-based channel that measures the time of process execution inside the TEE by tracking entry and exit inside the TEE, and ii) detecting Secure World screen-utilization by monitoring voltage, display frequency changes or other indicators of change in displayed images. We have addressed these side-channels in the design provided in Section 6. While we acknowledge that there may be other side-channels that are detrimental to the security of Wink, exploring and mitigating them is the subject of ongoing work, and orthogonal research [17, 38].

Cryptography. To make Wink practical, it is desirable to only employ standard (and efficient) cryptographic primitives. Specifically, Wink only requires the following: i) the existence of a secure public-key infrastructure (PKI) with trusted certificate authorities, ii) the existence of efficient mechanisms to jointly compute shared secrets between two trusted parties, and iii) the existence of cryptographically-secure one-way functions e.g., cryptographic hashes, KDFs, etc.

5 The Wink Design

This section details the Wink design. As discussed before, Wink requires a TEE-enabled device. The current version is instantiated under ARM TrustZone, the most commonly available TEE environment for commercial mobile devices. TrustZone provides all the TEE security guarantees listed in Section 4. Dozens of sensitive applications (TAs) are already protected by TrustZone on ARM commercial mobile devices (e.g., KeyMaster, SamsungPay, WideVineDRM, etc.) by running inside the Secure World.

An E2EE messaging app has several key components including networking, user I/O, and cryptographic operations. The most obvious (yet impractical) design of a "hardware-assisted secure world messaging app" for TrustZone-enabled devices runs all components of the app under Secure World protection. While hidden messaging is straightforward, since without compromising the Secure World the adversary has no visibility into the messaging process, there are at least three obvious problems with this design. First, adding an entire messaging app codebase to Secure World exponentially amplifies the TCB. For instance, the Signal and Telegram codebases have over 257 KLOC and 791 KLOC respectively, while open-source Secure World OSes have codebases that typically do not exceed 220 KLOC (e.g., OP-TEE kernel, Nividia TEE and LinaroTEE all contain less than 210 KLOC). Second, networking components can be exploited to gain control over the Secure World TAs or the OS. Finally, since TAs can only be installed by the device vendor, this design would require collaboration between the app developers and the vendors.

Therefore, in Wink, only the most critical components of the messaging process are executed inside the TEE. The challenge is to identify these components based on the adversarial capabilities. In this Section, we present a design where the adversary under consideration may compromise the messaging app running in the Normal World but is not capable of compromising the Normal World kernel through code injection, ROP, etc. For this threat model, only executing the cryptographic operations in the Secure World suffice to realize hidden messaging. The following Section deals with an adversary capable of compromising the Normal World kernel.

5.1 The Wink Secure World Application

Under TrustZone, Wink runs as a trusted application (TA) inside the Secure World and implements the cryptographic primitives used by the Normal World messaging app to provide a secure authenticated channel for public messaging. This is established using PKI keys whereby after an initial certificate exchange, a symmetric session key is jointly computed e.g., using the Diffie-Hellman Key Exchange protocol. The resulting *public session key* encrypts public messages in that session. Importantly, all cryptographic keys are protected by the TEE and the encryption/decryption of public messages is performed within the TEE, isolated from the untrusted software running in the REE. Compared to running E2EE messaging apps entirely in the REE, this provides stronger security guarantees for the cryptographic information, making Wink-integrated apps more resilient to unauthorized key disclosures, etc. Figure 1 illustrates the design.

In commercial TrustZone devices, TAs cannot be installed directly by the users. Instead, only vendor-signed software can execute as TAs inside Secure World. It is standard for the vendor to provide security-critical services such as Wink for users as TAs running inside Secure World. *Importantly, since*

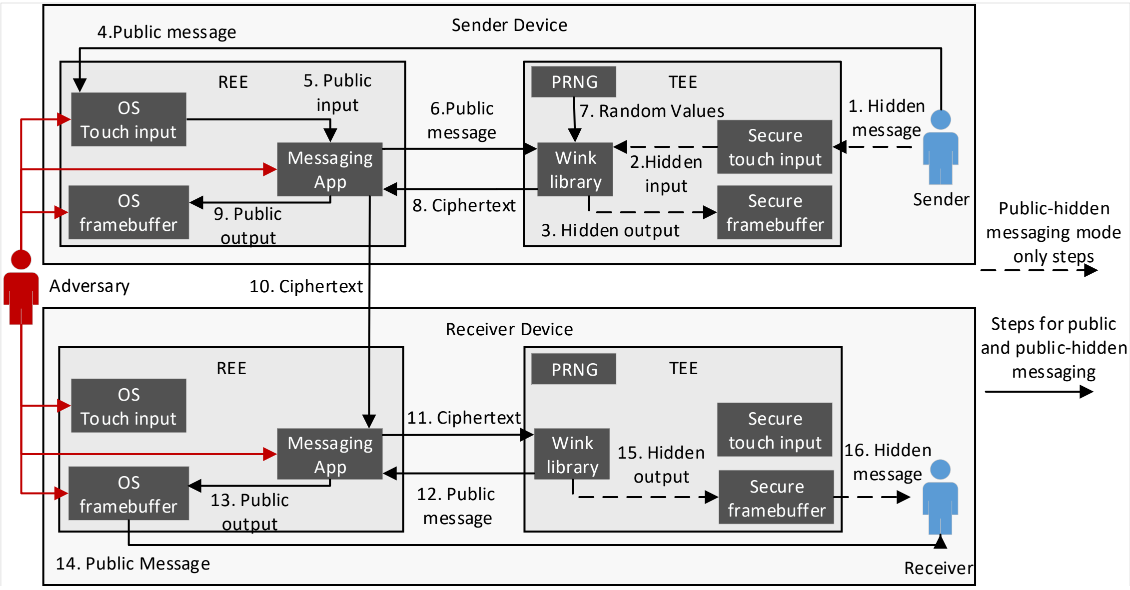


Figure 1: The Wink framework. Wink consists of a TEE hosted library that provides cryptographic implementations for E2EE messaging apps and secure I/O for providing users with a hidden messaging interface. The REE remains oblivious to any injected hidden messages.

users cannot install Wink, the presence of the corresponding library on a user device does not indicate in any way to the adversary that it is installed for the purpose of hidden messaging. Further, most TrustZone device vendors already provide crytographic libraries similar to the one Wink proposes as TAs (e.g., KeyMaster). The key difference between these libraries and Wink lies in the ability to provide hidden messaging.

Modes of operation. Wink has two modes of operation: a *public-only* messaging mode and a *public-hidden* messaging mode. The user should typically operate the device in the public-only mode and use the public-hidden mode when safe. A user password input decides the mode of operation. Wink requires three passwords: i) a *public* password corresponding to the public-only mode, ii) a *hidden* password corresponding to the public-hidden mode, and iii) a *disclosure* password for verification of public communication (described later). The ceremony for setting up the passwords is sketched in Section 5.3. When the Normal World messaging app is started, it requires the user to provide a password. The password input and verification are performed directly within the Secure World. The password determines the mode of operation.

- **Public-only**. In this mode, Wink only performs cryptographic operations for the messaging app in the Normal World. Wink operates in this mode on the public password input. Wink stores public message-specific metadata (which may also contain hidden data) until the messages are deleted by the messaging app. This metadata is stored in order to allow verification of public communication later under coercion (described later).
- **Public-hidden**. In this mode, Wink may also inject hidden messages into the public communication channel. This mode is enabled only upon the provision of a hidden password. The existence of the hidden password is denied under coercion. In this mode, users may also input and view hidden messages through a Secure World-provided user interface (described next). Encrypted hidden messages are injected into the public messaging channel while those already received are displayed on the interface. Wink protects hidden message confidentiality and only shows them to users upon hidden password input. Thus, it is crucial for the user to not reveal the existence of the hidden password to adversaries either accidentally or under coercion. A leaked hidden password can lead to revealing the public-hidden mode usage as well as the hidden messages exchanged.

5.2 Secure I/O for Hidden Messaging

Wink protects hidden messages both in transit and on the end devices themselves. For this, no evidence of hidden I/O is leaked Normal World software, including the messaging app itself and the Normal World OS. To accomplish this, the Wink cryptographic library sets up a hidden messaging interface by directly communicating with the touchscreen through Secure World-protected I/O channels.

In order to securely display information to the user, Wink takes control over the *touchscreen framebuffer driver* when the user holds the power button for three seconds. The power button interrupt is configured for Secure World usage such that Wink first receives all power button presses. This setup ensures that users can trigger the public-hidden mode interface without Normal World knowledge. Of course, regular button presses are forwarded to the Normal World for maintaining standard power button functionality. Once the library running in public-hidden mode receives this interrupt, first Wink temporarily prevents the Normal World from reading or writing framebuffer data. Then, the library saves the last framebuffer state and displays the hidden messaging interface. The hidden messaging interface is drawn on the framebuffer from inside the Secure World, displaying a keyboard and the exchanged hidden messages. Additionally, a user-specified watermark is also shown on the interface to prevent interface spoofing attacks. This watermark is maintained inside Secure World and its setup is detailed in 5.3.

For user input, Wink takes control over the touch input device and monitors for user touches from inside the Secure World. The user input monitoring starts once the hidden messaging interface is displayed. The monitoring stops once the user exists the interface by triggering the Wink hardware interrupt. The user-provided input is then communicated as one or more hidden message chunks. Both user input and output are hidden from the Normal World, as the input buffers are always cleared and the framebuffer restored prior to returning executing in Normal World.

5.3 Wink Setup Ceremony

Figure 2 depicts the ceremony of setting up the Wink library. Initially, the user asks the messaging app to set up her password. This request is forwarded to the Wink library, which will provide the user with an interface for entering the three passwords (public, hidden, and disclosure passwords). Once the user has entered her passwords, Wink will also ask the user to enter a text only she knows. This text is drawn on the hidden messaging interface as a watermark in order to prevent spoofing attacks from the Normal World. Once all required data is entered Wink returns execution to the messaging app.

Subsequently, the password hashes and the watermark never leave the Secure World. Instead, Wink indicates a successful verification of the messaging on the provision of either of the public or the hidden password. In both cases of operation, the information relayed to the messaging app is exactly the same, ensuring that the app is oblivious to the actual mode of operation. Only if the hidden password is provided hidden messaging functionalities are enabled in Wink.

5.4 Interfacing with Messaging Applications

Figure 3 provides a high-level overview of the interactions between the Normal World messaging app and the Wink library (e.g., setting up passwords, adding contacts, and exchanging messages). Specifically, the Normal World applications (messaging applications) can only access the library through a set of APIs provided for handling cryptographic operations and storing sensitive information (e.g., keys, and passwords). These APIs are exposed through a series of SMCs, which can be accessed by the messaging applications through OS-provided system calls. In order to use the provided APIs, the application invokes the corresponding system call. All system calls and parameters are visible to the OS, which forwards them to the library inside Secure World. Thus, it is crucial that no evidence of hidden messaging is leaked by the parameters passed or the calls to the cryptographic library themselves.

The Secure World hides the inner workings of Wink cryptographic operations from both Normal World applications and the OS. Once the Secure World receives an SMC call, it obtains complete control over the device and can perform the hidden operations required by Wink. These operations include (i) reading and writing hidden user input, and (ii) injecting hidden messages into the public communication channel. The public and hidden message exchanges under Wink are illustrated in Figure 3. The black steps are executed in both public-only and public-hidden mode, while the red steps are only executed in public-hidden mode. Importantly, the generated random coin depicted (Step 9) is overwritten by the hidden message ciphertext chunk in public-hidden mode.

Note that sending each hidden message chunk requires an accompanying public message. Further, the hidden message can only be reconstructed and read by the receiver when sufficient hidden message chunks are received for its reconstruction. Thus, to deliver a hidden message, the sender has to construct and send as many public messages as there are hidden message chunks. The number of chunks created by Wink for each hidden message depends on the bandwidth available, determined by the message format of the public messaging app, and is discussed in more detail in Sections 8 and 9. However, the sender can always construct and send additional public messages when they are required to finish the delivery of a hidden message. Wink helps the sender plan his public messages by noting how many are required to finish sending the current pool of hidden messages.

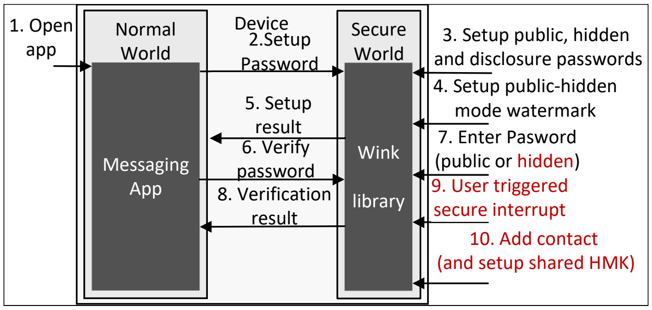


Figure 2: Wink setup ceremony. For a Wink-integrated messaging app, the Secure World hosted Wink library manages the password setup and verification. Entering the public-hidden mode password enables setting up hidden contacts and hidden message exchanges using symmetric encryption keys dubbed Hidden Master Keys (HMKs).

5.5 Key & Contact Management

The Wink Secure World TA securely stores all cryptographic information required for messaging (both public and hidden) encrypted under keys derived from a device-specific master key. Both the master and derived keys never leave Secure

World, ensuring that only Wink knows the derived keys. Similarly, Wink also stores encrypted contact information and Hidden Master Keys (HMKs) for hidden messaging, which are saved encrypted on physical persistence storage. Ideally, the dedicated TEE persistent storage would be used for saving these details. However, not all TrustZone-enabled commercial devices provide such storage. Thus, this information is stored in the form of an encrypted blob at a fixed Normal World location, a standard mechanism for securely storing TA data.

A pre-determined amount of persistent storage is reserved for saving HMK and contacts when Wink is installed. The entire blob is read into memory when the device boots up and there is a call to the Wink library, and thereafter re-encrypted and stored during a graceful shutdown. *This ceremony is performed regardless of the presence of any hidden keys, metadata, etc.* Naturally, due to the fixed capacity, the number of contacts that can be added for hidden messaging is limited, but since the only information Wink requires per contact is their corresponding HMKs, a small amount of storage may suffice. For instance, with 64-bit user identifiers and 1024-bit HMKs, 1MB of reserved storage will allow more than 3000 contacts. We also note that storage-level plausible-deniability is a well-studied problem, even in the context of mobile devices [16, 28, 46]. These solutions can be employed here (in future versions) for more efficient designs.

Hidden Keys & Contacts. To facilitate hidden messaging, for each contact, users are required to exchange a shared secret *once* over an out-of-band communication channel. This may be realized either through physical exchange when the users meet or through cryptographic protocols like deniable authenticated key exchange [22, 56]. In either case, the shared secret never leaves the Secure World. In case the users exchange the secret in person, it is input directly into the Secure World (using secure I/O) and is made available only on the provision of a correct hidden password. For this, when the messaging app adds a new contact, the Wink library generates the secret and a corresponding QR code. The QR code is then presented as a secure output from the Secure World. The counterpart Wink TA (being added as a contact), scans this QR code as input into the Secure World. For this, Secure World takes control of the camera temporarily and loads the QR image. Finally, the secret is extracted from the QR image and maintained inside Secure World, associated with the corresponding contact. This exchange process is similar to the physical verification process in several existing messaging apps. For example, Signal key bundles are verified also by scanning QR codes. Under Wink's much more powerful threat model, this optional feature becomes mandatory for hidden messaging due to the lack of a trusted communication path prior to the exchange of the shared secret.

The shared secret is used to derive a hidden master key (HMK). All hidden messages are encrypted under the HMK. Even a fully compromised Normal World cannot access the Wink HMKs maintained inside Secure World. Without having access to the HMK, the adversary cannot breach the confidentiality of the hidden messages *even if the public keys are revealed*. In the current design, each pair of communicating apps have a unique HMK, which is securely stored and encrypted with a password-derived key by the Secure World. All HMK(s) are made available to the Wink library only upon the provision of the correct hidden password. As a result, providing the hidden password makes all HMKs stored on the device available. In addition, Wink has a static HMK per user which is used for the hidden communication channel. Future designs will include more fine-grained access controls, and – at the cost of additional bandwidth – also possibly additional security features such as forward secrecy i.e., the HMK refreshed through key ratcheting and KDFs, etc.

Controlled Disclosure of Public Keys. Protecting public message keys in the TEE makes the public messaging opaque to the adversary. However, public message keys and public metadata have to be disclosed in accordance with the standard plausible deniability model where a user is asked to hand over a key by a coercive adversary[3] (see Section 4). Since the public message key(s) is stored in the Secure World, we need a mechanism to reveal this on-demand to the user, without making the keys accessible to compromised apps or OS. Without Secure World compromise, the only way to retrieve such information is through Wink provided APIs. Wink supports controlled disclosure using a disclosure password. On provision, Wink reveals the encryption keys and metadata for past public messages (up to the point stored by the messaging app) using secure output. Once the disclosure password has been used, Wink allows a password reset.

Only users in possession of a *disclosure password* (i.e. those also in possession of both hidden and public passwords) can use the provided API for disclosing public messaging details. The disclosure password enables Wink to provide a transparent public messaging mode, where users can disclose all public messaging details to adversaries upon coercion. However, Wink ensures that no users, even under coercion, can ever disclose the hidden messaging details. Importantly, the keys are revealed only using Secure World output and are not exposed to any Normal World software.

5.6 Security Analysis

The security arguments underlying Wink fall into three categories. First, it needs to be argued that for any generic E2EE messaging application that is compatible with Wink, integration does not come with lower security assurances. In other words, a Wink-enabled messaging application retains the same message privacy guarantees that the messaging application has without Wink. Second, it needs to be shown that injecting hidden messages into public communication does not impact the privacy guarantees of public communication,

---

[3] In several nation-states, handing over encryption keys is in fact law-mandated [39]

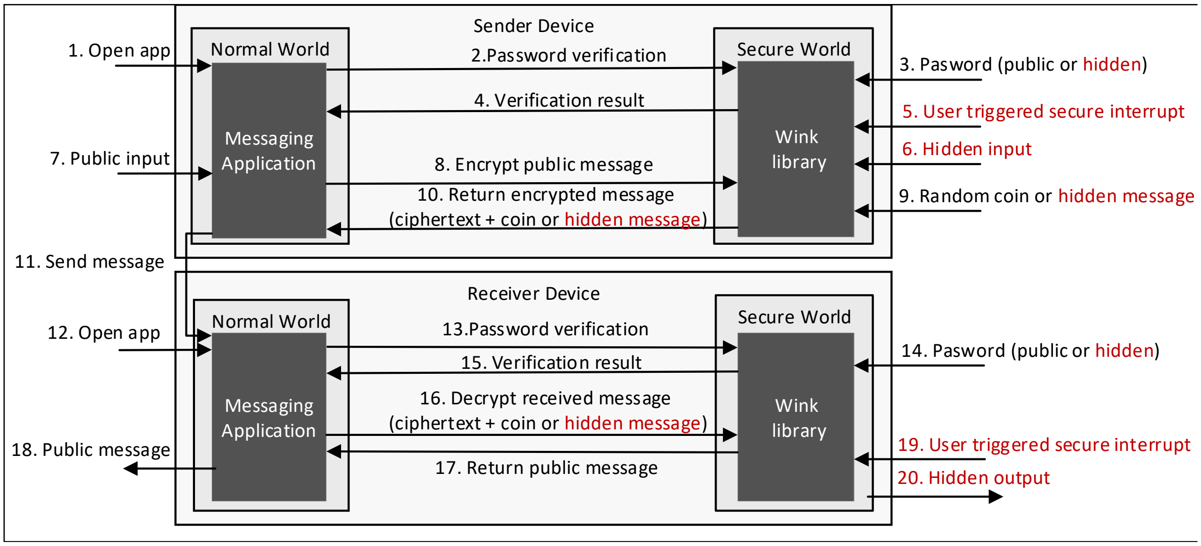


Figure 3: Message exchange under Wink. Based on the user password, Wink runs either in the public-only or the public-hidden mode. In public-hidden mode, the user can enter and read hidden messages, which are encrypted and exchanged between devices masqueraded as the random coins of public messages.

while also ensuring that the message injection remains undetectable in the message transcripts. Informally, message transcripts correspond to information exchanged on-wire which includes the message ciphertexts, random coins, etc. Finally, Wink should also ensure that any normal world monitoring (short of kernel compromise) cannot detect hidden message injection based on execution patterns. Execution transcripts correspond to on-device operations when Wink operates in a particular mode of operation which includes data inputs, context switches, the timing of operations, etc.

We will show that the message transcripts of a Wink-enabled E2EE application is indistinguishable from the message transcripts of the same application without Wink to a computationally-bound adversary observing the network transcripts. In other words, Wink does not distinguishably alter the message transcripts of the E2EE application, and thereby does not compromise the privacy guarantees of the vanilla application. Furthermore, most E2EE messaging apps aim to provide message confidentiality, integrity, and forward security. Typically, applications such as Signal and Telegram use authenticated encryption schemes that provide both authenticity (integrity) of messages and confidentiality.

Wink preserves message confidentiality, integrity, and forward security when integrated with an E2EE app. Showing that a Wink-enabled E2EE application retains integrity and forward security guarantees is straightforward: Wink only alters the way a random nonce is selected for the authenticated encryption scheme used in the E2EE app, which **impacts neither message integrity (tag generation is independent of the random nonce) nor forward security (which entirely depends on the key derivation mechanism)**. Thus, as long as the E2EE app implements *message integrity* using an authenticated encryption scheme, and provides *forward security* through its key derivation process a Wink-enabled version of the app retains these two properties. Showing that *message confidentiality* is also retained requires further analysis.

Theorem 1. *Given a generic E2EE messaging application* $\mathcal{M}$*, compatible with Wink, and a Wink-enabled version of* $\mathcal{M}$*, denoted* $\mathcal{W}$*, operating in either public-only or public-hidden mode, the* **message transcripts** *of* $\mathcal{M}$ *and* $\mathcal{W}$ *are indistinguishable to a PPT adversary if IND-CPA secure encryption producing ciphertexts indistinguishable from random exists.*

We will also show that for a device-resident adversary in the Normal World that is incapable of privilege escalation, the message transcripts of Wink operating in the public-only mode and public-hidden mode are indistinguishable.

Theorem 2. *Normal world adversaries (incapable of privilege escalation) and on-wire adversaries cannot distinguish between* **message transcripts** *pertaining to public-only communication and transcripts containing public + hidden communication for a Wink-enabled E2EE messaging app if there is an IND-CPA secure encryption producing ciphertexts indistinguishable from random exists.*

Finally, we will show that a Normal World adversary cannot distinguish between the public-only and public-hidden modes of operation based on their execution patterns.

Theorem 3. *Normal world adversaries (incapable of privilege escalation) cannot distinguish between* **execution transcripts** *pertaining to either public-only or public-hidden modes of operation.*

All proofs are in the full version[4].

---

[4] https://arxiv.org/abs/2207.08891

6 Wink Under REE Kernel Compromise

In this section, we will first describe the impact of REE kernel compromise on the Wink design presented in Section 5. Then, we will describe a solution unifying the public and hidden I/O channels through the Secure World preventing confidentiality breaches due to REE kernel compromise. Finally, we will analyze the impact on the TCB size, ease of integration with messaging apps, and overall plausible deniability.

Impact of Kernel Compromise. Events in the past [19, 31, 32] have shown that REE kernel vulnerabilities may enable sophisticated privilege escalation from compromised REE applications and provide adversaries with control over the REE kernel. Taking over the kernel will not only provide direct access to all public messages but also enable tracking and timing of TEE operations, as mentioned in Section 4. Specifically while Wink ensures that hidden message confidentiality remains intact despite REE kernel compromise, the interrupts used to initiate hidden message inputs can be monitored by the compromised REE kernel. Wink-introduced interrupts due to user I/O may be arbitrarily long (although typically in the order of seconds). Thus, it is straightforward for a compromised kernel to time Wink Secure World execution and determine when it handles interrupts specific to Wink hidden I/O. Padding interrupt handling to the order of user I/O would render such monitoring useless. However, it is infeasible due to the impact on system performance.

Unifying Public and Hidden Interfaces. Without moving the entire messaging app logic into the Secure World, one solution to tackle this problem is to obfuscate the hidden message I/O time with the public message I/O time. Specifically, in addition to the porting public messaging cryptography, Wink simply needs to provide a UI for public messaging directly using the Secure World I/O, alongside the hidden interface detailed in Section 5. In this way, the time required for public-only I/O can also be used for additional hidden I/O. Using the same interface for both public and hidden I/O is also more practical and user-friendly since it enables the user to view and write public and hidden messages concurrently. Hardware interrupts are no longer required to initiate hidden inputs rendering any Normal World monitoring useless.

Handling public I/O through the TEE also has an added advantage: *message confidentiality is now ensured against a compromised REE kernel*. More specifically, since the REE now only sees the messages encrypted and essentially acts like a "man-in-the-middle", confidentiality is guaranteed when using an appropriate IND-CPA secure encryption scheme. However, it is worth noting that this does not subsume the need for hidden messaging since ensuring confidentiality is not enough for plausible deniability. Coercive adversaries will still demand cryptographic metadata pertaining to the messages exchanged, and to maintain plausible deniability, the user now hands over the plaintext public messages, and the cryptographic metadata used to encrypt these messages using the controlled disclosure feature detailed in Section 5.

Cost of Integration. The added security of porting the public I/O to the Secure World comes at the cost of increased integration complexity and a larger TCB. In particular, each messaging app has a unique user interface for messaging, which can vary in complexity and code size depending on the features supported by the app client (e.g., pictures/movies, emojis, voice messages, etc). Integrating and maintaining UI for each app into Wink requires a major engineering effort and increases the Secure World TCB, perhaps unreasonably in some cases. In contrast, Wink only requires a minimalist implementation of an I/O interface for hidden messaging since the low bandwidth hidden communication channel offered by most messaging apps can only support text messages. Clearly, this interface is not enough for everyday public messaging needs with modern messaging clients.

Ideally, if multiple messaging apps were to subscribe and use a generic Secure World UI provided by the vendor, then the increase in TCB may be reasonably controlled in addition to making app integration and maintenance straightforward. We believe that security-conscious messengers have enough incentives to adopt this model in collaboration with device vendors given the added protection it offers for public messages. With messaging apps being frequently targeted to breach confidentiality via system compromise [43, 45, 54], a more secure E2EE app needs to be resilient to these attacks.

Security Analysis. Unifying the I/O interfaces for public and hidden messages does not alter the message transcripts generated in the public-only and public-hidden modes of operation. Without obtaining the cryptographic metadata for the hidden messages protected in the Secure World, the adversary cannot distinguish (with more than negligible advantage) message transcripts for public-only communication and public + hidden communication (Theorem 4). Compromising the REE does not enable the adversary to compromise the Secure World in our threat model. We will show that with the proposed changes to the Wink design, a compromised REE kernel also cannot distinguish the mode of operation based on the execution transcripts.

Theorem 4. *A compromised REE kernel cannot distinguish between Wink's public-only and public-hidden modes of operation based on their corresponding* **execution transcripts** *if IND-CPA secure encryption produces ciphertexts indistinguishable from random exists.*

The proof is in the full version[5]

Side-Channels. As with any shared-hardware system, the threat of side-channels is applicable to Wink. However, unifying the public and hidden message interface eliminates all hidden interface-specific resource usages. Thus, the REE adversary is prevented from using any potential side-channel based on these resource usages.

---

[5] https://arxiv.org/abs/2207.08891

As described before, timing entry and exits into the Secure World does not provide any additional information. Since the interface for inputting/outputting hidden messages and public messages is the same, the resource utilization due to screen usage is also indistinguishable between the cases when the screen is used only for public messaging vs. public + hidden messaging. Thus, monitoring resources such as power consumption, refresh rate changes, etc. due to the UI does not reveal any information to the adversary. That is, any variations observed in these parameters are as likely to be the result of only public message input/output as due to both public and hidden message input/output. The intermixing of public and hidden messaging operations inside the TEE.

Obviously, if additional hidden messages are processed inside the Secure World, additional computation is required. However, very few additional cycles are used for injecting the hidden payload in the random coins (as demonstrated in Section 8). Power consumption does not vary out of noise boundaries. Thus, while side-channels based on computation or other resources might be discovered in the future, it is not immediately clear if these side-channels can in fact be exploited to break plausible deniability (see Section 10 for further discussion). Discovering and mitigating such side-channels is indeed very important, and each side-channel mandates its own evaluation in terms of effectiveness and evaluation. However, this is beyond the scope of this paper.

7 Implementation

Only device vendors have access inside the TEEs of TrustZone commercial mobile devices and this access is strongly guarded. Instead, we have implemented the first Wink prototype on a i.MX6 Nitrogen6X Max [27] development board, featuring a mainstream ARM Cortex-A9 CPU and 4GB of DDR3 memory. More importantly, the board provides complete access to both Normal World and Secure World.

The Wink prototype uses U-boot [20] to load up a minimal operating system, OP-TEE [55] inside the Secure World and a vanilla Android 7.0 Nougat inside the Normal World. When the device powers on, U-boot first loads into memory the code of both OP-TEE and Android. Then, U-boot passes execution to OP-TEE, which starts executing with complete control over the device. OP-TEE sets up the Normal and Secure World regions, assigns interrupts for each, and prepares handlers for inter-world communication. Then, OP-TEE sets up drivers for Secure I/O and prepares a cryptographic library for handling Normal World requests. Once the Secure World is set up, OP-TEE passes execution to the boot code of the Android kernel (Linux version 4.1) inside Normal World.

The Wink trusted computing base (TCB) consists only of code executing inside Secure World. In the prototype, Secure World only contains a stripped-down version of OP-TEE (12769 LOC), drivers for the user I/O (4423 LOC), and the Wink cryptographic library (3009 LOC). In total, the Secure World only contains 20201 lines of code.

7.1 Secure World cryptographic library

As described in Section 5, only a Secure World TA is required for injecting hidden messages into an authenticated secure public messaging channel. For simplicity, the prototype TA runs as part of the OP-TEE which forwards all SMCs incoming from Normal World to the Wink cryptographic library API. The library can operate in either public-only or public-hidden mode (Section 5.1). The library initially starts in public mode, where it can (i) encrypt/decrypt messages, and (ii) save cryptographic keys and passwords. On password storage requests from the messaging app, the user can either enter only the public and disclosure passwords or also a hidden one through Secure I/O. On password verification, the library switches into public-hidden mode only when provided the hidden password. In public-hidden mode the library can also show the hidden messaging interface and inject hidden messages into the app's public messaging channel. *Crucially, Wink ensures that the duration of processing encryption/decryption requests in public-hidden mode is indistinguishable from when they are processed in public-only mode.*

When injecting a hidden message, the library notifies the user how many public messages are required to successfully send the entire hidden content. In the current implementation, for each public message, only a 15-byte hidden message chunk can be embedded for Telegram (Section 9) and a 16-byte chunk for Signal (Section 8.1) . The library keeps injecting the hidden message chunks until they are all exhausted. When no hidden messages are provided by the user, the library operates similarly to the public-only mode. On the receiver side, the library extracts the hidden messages and displays them when operating in the public-hidden mode.

7.2 Secure I/O

Not unlike cell phone chipsets, the i.MX6 provides a set of registers inside the Central Security Unit (CSU). These registers control the accessibility of peripherals, including I/O devices. Wink takes advantage of this in order to on-demand enable and disable Normal World access to the touchscreen input (I2C touch interface) and output (display framebuffer).

Output. Wink displays an interface for hidden messaging when Wink-specific hardware interrupt is triggered by the user. For our implementation, we have chosen to set up a Secure World GPIO key driver that intercepts the power button interrupts when Wink operates in public-hidden mode and shows the hidden messaging interface if this button is pressed for 3 or more seconds. Because the Secure World only intercepts the power button hardware interrupts, Normal World software cannot trick Wink into showing the respective interface. Further, users can only trigger the hidden messaging interface provided they know the Wink hidden password.

Temporarily altering the power button functionality also prevents the user from accidentally shutting down her phone, while the Wink library operating in public-hidden mode has not sent all hidden messages or shown all received ones.

The hidden messaging interface always displays the exchanged hidden messages and a static keyboard, which can be used by the user for input. A similar interface is displayed for entering and verifying passwords. A framebuffer driver inside Secure World is used to draw the two interfaces. Using its control over CSU configuration, the driver takes over the touchscreen display and its framebuffer while the library is running. If the power button is pressed while Wink operates in public-hidden mode, the framebuffer driver first saves the current Normal World framebuffer image. Then, the hidden messaging interface is drawn on the framebuffer and displayed to the user. The Normal World image is restored in the framebuffer prior to returning execution to the Normal World. This process hides the framebuffer driver operations from Normal World adversaries.

Normal World adversaries can also trick the users into entering hidden messages or passwords into a spoofed hidden messaging interface. To prevent such attacks, the framebuffer driver always displays the user-entered text watermark (at setup time) on the framebuffer. This text is provided using Secure I/O and is never revealed to the Normal World; the watermark assures the user that the authentic hidden messaging interface is on display.

Input. The Secure World provides hidden input to the cryptographic library by directly monitoring the I2C interface used by the touchscreen to send touch input data. While the hidden messaging interface is displayed, Secure World takes control of the I2C interface and prevents Normal World adversaries from monitoring the interface for hidden messages.

On touch events, the I2C interface provides the Secure World driver with a buffer that indicates touch screen presses and the coordinates of each touch. On touch presses, the driver converts the received data into a 2D point. This point is then mapped into a static keyboard layout displayed on the touchscreen Secure World framebuffer. On each touch event, the driver provides the entered keystrokes to the Wink library.

8 Evaluation

Integration with Messaging Apps. As proofs of concept, we have integrated the Wink library with Signal Private Messenger and Telegram Messenger. In addition, we have investigated several other messaging apps including Briar[6]. For Telegram we have identified a 15-byte salt exchanged under the MTProto 2.0 protocol as a potential random coin that can be exploited to inject hidden messages. On the other hand, the Signal message format provides two opportunities for hidden message injection: a 32-byte random ECDH public key exchanged with every message, and a 16-byte IV for the Sealed Sender encryption (an envelope) for exchanged messages. Finally, Briar-encrypted messages include a 32-byte random salt and 24-byte IV.

Application	Random Coins	No. of Public Messages
Telegram [53]	15 byte salt	$\left\lceil \frac{n}{15} \right\rceil + 2$
Signal [49]	16 byte IV	$\left\lceil \frac{n}{16} \right\rceil + 2$
Briar [8]	32 byte salt, 16 bytes IV	$\left\lceil \frac{n}{48} \right\rceil + 2$

Table 2: Bandwidth available for hidden messaging with different E2EE messaging apps. With Telegram, for a n byte hidden messages, $n/15+2$ public messages are required as cover traffic. Using the Signal 16 byte IV a similar bandwidth is available. Using both the 32-byte salt and 16-byte IV provides a significantly larger bandwidth for hidden message injection under Briar.

In our evaluation, we mainly present empirical results pertaining to the solution presented in Section 5 which protects against REE adversaries incapable of compromising the REE kernel. The only change required for making the solution resilient against REE kernel adversaries is performing public I/O through the Secure World interface which does not introduce additional timing overheads.

Bandwidth for Hidden Messaging. The length of the random coins determines the available bandwidth available for hidden message injection in each messaging app. Table 2 enlists the available bandwidths for the surveyed apps. For Telegram we have built a prototype (evaluated in Section 9) that enables sending a n-byte hidden message using $\left\lceil \frac{n}{15} \right\rceil + 2$ public messages, where the 2 extra messages are for the metadata. Similarly under Signal sending a n-byte hidden message requires $\left\lceil \frac{n}{16} \right\rceil + 2$ public messages (details in Section 8.1). Briar provides both a 32-byte salt and 16-byte IV which combined enables Wink to inject additional hidden message bytes per public message.

8.1 Integration with Signal

As a proof-of-concept, we have integrated the Android Signal Private Messenger with Wink (see Section 3). The instrumented Signal enables users to exchange hidden messages under the Wink protocol while retaining all existing functionalities for public messaging. For Signal, **60 LOC** have been changed for hooking in the Wink system calls, mostly consisting of Java Native Interface code.

Switching Operating Modes. In terms of user authentication, Signal does not maintain passwords or require users to explicitly log in. Instead, it mainly relies on 2FA (through SMS sent to provided phone numbers) and OS-level user authentication. Thus, we had to find an appropriate method of hooking into the password verification required under Wink. Instead of introducing mandatory authentication every time Signal is opened, we have opted to hook into the Signal PIN functionality. The PIN is a (numeric/alphanumeric) code used

---

[6] `https://briarproject.org`

	Signal	Wink-Integrated Signal	Overhead
Metadata Encryption time (ms)	0.17 ± 0.003	0.42 ± 0.002	+ 0.246 ms
Metadata Decryption time (ms)	0.19 ± 0.004	0.43 ± 0.003	+ 0.240 ms

Table 3: Encryption and decryption time for metadata (33 bytes) in Signal ± Wink. Overheads are under 0.25 msec per message. This is virtually unnoticeable and dominated by other per-message operations which take **thousands** of times longer (e.g., typing, network transfer, etc.).

to support features like non-phone number-based identifiers.

We instrument the Signal PIN verification and move it into Secure World under Wink. This enables users to provide passwords (Signal PIN) to the Wink library. In turn, the library provides verification results back to Signal. Similarly, the PIN input is managed through the Wink library allowing the setup of the three Wink passwords (public, hidden and disclosure).

Once the Wink hidden password is set up, the user can use Signal's PIN verification to enable the Wink public-hidden mode execution. As previously described in Section 9, once the hidden password is entered, the Secure World intercepts all power button presses and shows the Wink hidden messaging interface, and entered hidden messages are exchanged while appearing as random Signal IVs.

Hooking Hidden Messaging. For Signal, we inject the hidden messages into the IV used for encrypting message metadata under the sealed sender [50] functionality. In short, a sealed sender adds another layer of encryption (an envelope) to each exchanged message. The metadata IV for encrypted envelopes can be replaced with any random string, thus making it suitable for hidden message injection. We leverage this opportunity to integrate Wink by porting the envelope (metadata) encryption inside Secure World, similar to the Telegram integration described in the process described in Section 9 for Telegram. Thus, for each Signal exchanged message, the metadata is encrypted and exchanged using either a Wink generated random IV or a hidden message encrypted with CTR mode AES under a 16-byte randomly generated HMK.

To instrument the Signal PIN verification and to introduce hidden message exchange we only had to replace a few lines of code with system calls. However, only message metadata encryption is ported inside Secure World for this proof-of-concept. For a real-world deployment, further engineering effort would be required to also move the remaining Signal cryptographic functions used for message content encryption inside Secure World for protection.

Signal Metadata Encryption Overheads. Under the sealed sender functionality, Signal encrypts and decrypts the metadata of each exchanged message using AES in CTR mode with no padding. For our Wink proof-of-concept we have ported these two operations inside the Secure World library. To estimate the overheads for integrating Wink with Signal we compared the time required for encrypting/decrypting the fixed length metadata of exchanged messages between the vanilla and Wink-integrated Signal.

To evaluate the metadata encryption/decryption times we instrumented Signal to automatically encrypt/decrypt the metadata of each message sent/received 1000 times. The average AES encryption and decryption are measured, collecting the average computation time and standard deviation. Table 3 presents the average metadata encryption and decryption time. AES encryption of metadata introduces an overhead of under 0.246 milliseconds for either encryption or decryption (of metadata – always 33 bytes), with a 95% confidence interval. The overhead includes the additional data copy operations from the Normal World to the Secure World. Importantly, most other operations take thousands of times longer (typing, message transfers, etc). As a result Wink overheads are virtually unnoticeable.

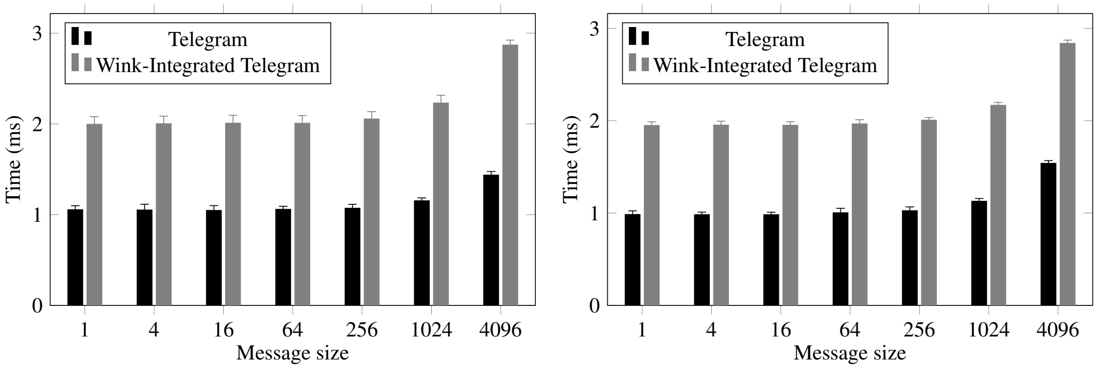
9 Integration with Telegram

We have integrated Telegram with Wink. Telegram is a popular open-source E2EE messaging app based on the MTProto 2.0 protocol. Compared to its more complex counterparts e.g., Signal, one advantage of the Telegram protocol is that the cryptographic operations are relatively simple to implement. We describe the salient features, more details are in [53].

Message Format. MTProto 2.0 derives a per-message 16 byte *message key* by hashing a shared *authentication key* and an augmented message payload, consisting of the message text itself, a 64-bit session identifier, a 64-bit salt (15 bytes in the actual implementation) and the required length of the padding. The message key is used to derive AES keys and IVs for the payload encryption using a key-derivation function (KDF). The encrypted payload, message key, and authentication key are sent over the wire. The salt provides the necessary randomness to the message keys and the payload.

The instrumented version of Telegram (with Wink) retains all existing functionalities for public messaging. Only **71 LOC** have been changed for hooking in the Wink system calls, mostly consisting of added Java Native Interface code and replacing the existing encryption/decryption logic. Importantly, since the Telegram message format is not altered, the instrumented version can still communicate with a vanilla Telegram version for public messaging, while relying on the Wink library for cryptographic operations.

Instrumenting Encrypted Messaging Logic. The Wink library stores authentication keys (shared key used in MTProto 2.0 protocol) inside Secure World and performs public message encryption and decryption. To this end, we instrumented the Telegram "secret chat" encrypted messaging logic using the MTProto 2.0 protocol. This logic mainly consists of deriving message keys and IVs through a series of SHA256 (or



(a) Encryption Time

(b) Decryption Time

Figure 4: Comparison of encryption, decryption times for Wink-integrated Telegram and vanilla Telegram. The error bars illustrate the margins of error for a 95% confidence interval. Performing Secure World cryptographic operations with the Wink library incurs a 2x overhead.

SHA1 for MTProto 1.0 protocol) computations, followed by an AES IGE [40] encryption or decryption. Under Wink, this logic is replicated inside the Secure World library.

Switching Operating Modes. For user authentication, Telegram allows users to set up a passcode (or PIN). Under Wink, Secure World library SMC calls replace Telegram passcode entry and verification logic. As a result, when passcode protection is enabled, the user is prompted by Wink to enter all three passwords (public, hidden, and disclosure), through a Secure I/O interface. These passwords are never revealed to the Normal World and the Wink library only relays to Telegram whether password verification succeeded or failed.

On providing the hidden password, the Secure World intercepts all power button presses and shows the Wink hidden messaging interface every time the power button is pressed for longer than 3 seconds. The interrupt-based provided interface enables users to manage hidden contacts and read/write hidden messages at any point during the Wink public-hidden mode execution. The interception of the power button presses ends when Wink resumes operating in public mode (e.g., by entering the public password). Hidden messages are then exchanged as random Telegram message salts.

Telegram Hidden Message Injection. Telegram's MTProto 2.0 protocol includes an 8-byte salt in the encrypted message sent over the wire. In the open-source code, we found that the actual salt size used is 15 bytes (perhaps to make Telegram more resilient to brute-force attacks). Wink uses this 15-byte salt for hidden message injection.

Hidden messages are encrypted with AES in CTR mode under a 16-byte randomly generated hidden master key. A hidden message, if longer than 15 bytes, is encrypted as a whole and the ciphertext is broken into 15-byte chunks. Each chunk is subsequently used as a random salt for a public message and sent over the wire as part of the encrypted payload.

In addition, each hidden message is preceded by a 16-byte random IV and a 8-byte message length indicator, which is broken in two 15-byte encrypted messages. The remaining 6 bytes out of the combined 30 bytes is used to add redundancy e.g., all 0s allowing the decryption logic to distinguish between actual messages and random information.

While this simple scheme judiciously utilizes the available bandwidth (by reducing the amount of metadata) it has one drawback: if the chunks for a long-hidden message are received out-of-order then the decryption logic on the receiver side may not be able to reorder the encrypted ciphertext chunks. We leverage the fact that each encrypted payload in Telegram has a *sequence number* which allows the receiver to correctly re-order out-of-order public messages.

Since the hidden message injection in Wink is synchronized with the public messaging i.e., each hidden chunk is injected with one public message, the ordering of the public messages also establishes the hidden message chunk ordering. Thus, Wink can leverage the public message sequence numbers to correctly reorder the hidden message chunks for decryption.

Telegram Message Processing Overheads. To estimate the performance overheads of integration, we compared the time required for encrypting/decrypting messages in Wink-integrated Telegram with the time required in the vanilla version. Note that the only difference between the two cases is that the Secure World-hosted Wink library implements the crypto for the Wink-integrated version while the vanilla version implements its own cryptography in the Normal World.

The experiment is set up by instrumenting Telegram to automatically generate and encrypt/decrypt messages of sizes between 1 and 4096 bytes (max size supported by Telegram). For decryption, a remote Telegram instance encrypts and provides messages between 1 and 4096 bytes. For each generated message the experiment is repeated 1000 times, measuring the encryption process – SHA256 computations for key and IV generation plus AES IGE encryption – duration. Then, the average execution time and standard deviation are computed. A similar process is carried out for timing the decryption of message ciphertexts.

Figure 4(a) and Figure 4(b) illustrate the average time required for encrypting and decrypting messages under both vanilla and Wink-integrated Telegram respectively. The observed results indicate that using the Wink library for secure cryptographic operations only makes the encryption, and decryption slower by a factor of 2x. This is expected given the context switches and the memcopies required for utilizing the Wink cryptographic operations. For a messaging application, the overhead is almost negligible since it only slightly increases overall message processing times (around 1 ms). The bulk of the time is spent on user inputs, which are usually in the order of a few seconds. The results also indicate that the time required for encryption and decryption is not significantly affected by the message size. *The almost minimal overheads justify using Wink only for more secure public messaging and strengthen plausible deniability arguments.*

10 Discussion

In this section, we will further discuss threat model assumptions and point out potential Wink design drawbacks.

TEE Compromise. Naturally, the TEEs themselves are far from invulnerable. Importantly, however, the small, comparatively carefully designed TEE TCB results immediately in proportionally fewer vulnerabilities, oftentimes by 2-3 orders of magnitudes! In fact, between 2013 and 2018 only a handful of CVEs have been reported for TEEs [12], in comparison to the 1647 CVEs reported for Linux, a rich OS. Additionally, in Trustonic, one of the most largely-deployed TEEs, only 5 vulnerabilities have been reported in this timeframe, *330 times* fewer than for Linux (1647). Moreover, most reported TEE vulnerabilities are located within TAs, which are isolated by the TEE OS from Wink and do not directly impact hidden messaging security. Overall, the fact is that the significantly smaller TCB/attack surface raises the bar significantly. However, hardening TEEs and specifically Trustzone against all possible attack vectors is orthogonal to the scope of this work.

Side Channels. As with any shared-hardware system Wink may be subject to side-channels that undermine its security guarantees; the solution in Section 6 mitigates several side-channels that are specifically detrimental to Wink security guarantees. However, we acknowledge that there may be side-channels that are not covered or mitigated under the current Wink design. There is a large body of work on mitigating these side-channels for general TrustZone-protected trusted services, e.g., [21, 36, 38, 59]. Integrating these solutions in future versions will eliminate the side-channels detrimental to Wink security. In addition, identifying other side-channels that may undermine Wink is the subject of future research.

Password-based attacks. As with any password-based system, Wink requires strong passwords, which can not be guessed in a reasonable amount of time. To prevent password brute forcing attacks, Wink imposes strict length and content requirements, and locks access permanently after a reasonable amount of attempts.

Spoofing Wink UI. Adversaries could try to show users a spoofed version of the Wink interface (e.g., from a compromised application), to trick them into leaking their passwords (hidden and public) or hidden messages. Such attacks can be mitigated in various ways, including by the user-selected watermark maintained only inside Secure World and shown on the Wink drawn interface.

Data leakage through Wink. Wink is preferably installed by a vendor as firmware in the Secure World and provides a secure communication channel that is invisible to the Normal World. It only becomes visible to users when they utilize it through the Wink hidden-messaging UI. Importantly, the vendor is trusted by the user; Secure World is assumed to be benign and not compromised.

However, it is important to understand what that means. To that end, let us consider a malicious vendor or a Secure World OS that compromises Wink operations. In addition to obtaining all confidential information exchanged as hidden messages, they can also introduce their own collected data (e.g., collected user confidential information) into hidden messages and exfiltrate it without the user or Normal World software ever knowing. Thus, a compromised Wink can become a powerful tool for adversaries if they require a mechanism for leaking data from Secure World without leaving any evidence. Note, however, that such adversaries can introduce their encrypted data within any other exchanged random coins within Normal and Secure World software and achieve a similar result. Wink does not introduce such data leakage as a new attack vector.

11 Conclusion

This work presented Wink, a plausibly-deniable messaging framework enabling users to reclaim the ability to communicate securely even in the presence of powerful surveillance or coercive adversaries. It works by surreptitiously injecting hidden messages in cryptographic randomness inherent in end-to-end encrypted messaging. Users can plausibly deny the exchange of hidden messages, and also any evidence of using the messaging software itself. Wink can be efficiently integrated with a number of existing E2EE applications including Telegram and Signal with minimal external instrumentation, and crucially without needing to change existing standard on-wire message formatting.

Acknowledgments

This work was supported in part through NSF award 2052951. We thank our shepherd and the anonymous Usenix Security Symposium reviewers for their excellent suggestions and feedback. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation.

References

[1] Anderson, R., Needham, R., Shamir, A.: The steganographic file system. In: Information Hiding (1998)

[2] Ateniese, G., Magri, B., Venturi, D.: Subversion-resilient signature schemes. In: ACM Conference on Computer and Communications Security. pp. 364–375 (2015)

[3] Bellare, M., Jaeger, J., Kane, D.: Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In: ACM Conference on Computer and Communications Security. pp. 1431–1440 (2015)

[4] Bellare, M., Paterson, K.G., Rogaway, P.: Security of symmetric encryption against mass surveillance. In: Advances in Cryptology – CRYPTO. pp. 1–19 (2014)

[5] Berndt, S., Liśkiewicz, M.: Algorithm substitution attacks from a steganographic perspective. In: ACM Conference on Computer and Communications Security. pp. 1649–1660 (2017)

[6] Blass, E.O., Mayberry, T., Noubir, G., Onarlioglu, K.: Toward robust hidden volumes using write-only oblivious ram. In: ACM Conference on Computer and Communications Security. pp. 203–214 (2014)

[7] Borisov, N., Goldberg, I., Brewer, E.: Off-the-record communication, or, why not to use pgp. In: Workshop on Privacy in the Electronic Society (2004)

[8] Briar. https://briarproject.org

[9] Byrant, F.: 7 best keyloggers for android without root. Spy Drill (2021)

[10] Canetti, R., Park, S., Poburinnaya, O.: Fully deniable interactive encryption. In: Advances in Cryptology – CRYPTO. pp. 807–835 (2020)

[11] Canetti, R., Dwork, C., Naor, M., Ostrovsky, R.: Deniable encryption. In: Advances in Cryptology – CRYPTO. pp. 90–104 (1997)

[12] Cerdeira, D., Santos, N., Fonseca, P., Pinto, S.: Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In: IEEE Symposium on Security and Privacy. pp. 1416–1432 (2020)

[13] Chakraborti, A., Chen, C., Sion, R.: DataLair: Efficient block storage with plausible deniability against multi-snapshot adversaries. In: Privacy Enhancing Technologies Symposium. pp. 175–193 (2017)

[14] Chang, B., Zhang, F., Chen, B., Li, Y., Zhu, W., Tian, Y., Wang, Z., Ching, A.: Mobiceal: Towards secure and practical plausibly deniable encryption on mobile devices. In: IEEE/IFIP International Conference on Dependable Systems and Networks (2018)

[15] Chen, C., Chakraborti, A., Sion, R.: Pd-dm: An efficient locality-preserving block device mapper with plausible deniability. In: Privacy Enhancing Technologies Symposium (2019)

[16] Chen, C., Chakraborti, A., Sion, R.: INFUSE: Invisible plausibly-deniable file system for NAND flash. In: Privacy Enhancing Technologies Symposium. pp. 239–254 (2020)

[17] Cho, H., Zhang, P., Kim, D., Park, J., Lee, C.H., Zhao, Z., Doupé, A., Ahn, G.J.: Prime+count: Novel cross-world covert channels on arm trustzone. In: Annual Computer Security Applications Conference. p. 441–452 (2018)

[18] Coynash, H.: Russian fsb terrorizes another ukrainian activist in occupied crimea. http://khpg.org/en/index.php?id=1549897598

[19] Cui, Y.: Elevation of privilege vulnerability in the kernel (cve-2015-6640)). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6640

[20] Denk, W., et al.: Das u-boot–the universal boot loader. http://www.denx.de/wiki/U-Boot (2013)

[21] Dessouky, G., Frassetto, T., Sadeghi, A.R.: {HybCache}: Hybrid {Side-Channel-Resilient} caches for trusted execution environments. In: USENIX Security Symposium. pp. 451–468 (2020)

[22] Di Raimondo, M., Gennaro, R., Krawczyk, H.: Deniable authentication and key exchange. In: ACM Conference on Computer and Communications Security. p. 400–409 (2006)

[23] Google: Hardware-backed keystore. https://source.android.com/security/keystore (2019)

[24] Guardian staff: Whatsapp design feature means some encrypted messages could be read by third party. https://www.theguardian.com/technology/2017/jan/13/whatsapp-design-feature-encrypted-messages

[25] Han, J., Pan, M., Gao, D., Pang, H.: A multi-user steganographic file system on untrusted shared storage. In: Annual Computer Security Applications Conference (2010)

[26] Helms, K.: Us senators introduce 'lawful access to encrypted data act' — with backdoor mandate. https://tinyurl.com/lawful-access-backdoor

[27] Nitrogen6_max development board. https://boundarydevices.com/product/nitrogen6max/

[28] Jia, S., Xia, L., Chen, B., Liu, P.: Deftl: Implementing plausibly deniable encryption in flash translation layer. In: ACM Conference on Computer and Communications Security (2017)

[29] Kahney, L.: The fbi wanted a back door to the iphone. https://www.wired.com/story/the-time-tim-cook-stood-his-ground-against-fbi/

[30] Kaptchuk, G., Jois, T.M., Green, M., Rubin, A.D.: Meteor: Cryptographically secure steganography for realistic distributions. In: ACM Conference on Computer and Communications Security. pp. 1529–1548 (2021)

[31] laginimaineb: Android linux kernel privilege escalation (cve-2014-4322). http://bits-please.blogspot.com/2015/08/android-linux-kernel-privilege.html

[32] laginimaineb: Android linux kernel privilege escalation (cve-2014-4323). http://bits-please.blogspot.com/2015/08/android-linux-kernel-privilege_26.html

[33] Lentz, M., Sen, R., Druschel, P., Bhattacharjee, B.: Secloak: Arm trustzone-based mobile peripheral control. In: ACM Annual International Conference on Mobile Systems, Applications, and Services. pp. 1–13 (2018)

[34] Li, X., Hu, H., Bai, G., Jia, Y., Liang, Z., Saxena, P.: DroidVault: A trusted data vault for android devices. In: International Conference on Engineering of Complex Computer Systems (Aug 2014)

[35] Liu, D., Cox, L.P.: VeriUI. In: Workshop on Mobile Computing Systems and Applications (2014)

[36] Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., Lee, R.B.: Catalyst: Defeating last-level cache side channel attacks in cloud computing. In: IEEE International Symposium on High Performance Computer Architecture. pp. 406–418 (2016)

[37] Liu, H., Vasserman, E., Hopper, N.: Improved group off-the-record messaging. In: ACM Conference on Computer and Communications Security (2013)

[38] Liu, N., Yu, M., Zang, W., Sandhu, R.S.: Cost and effectiveness of trustzone defense and side-channel attack on arm platform. In: Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications. vol. 11, pp. 1–15 (2020)

[39] Lomas, N.: Could the UK be about to break end-to-end encryption? https://techcrunch.com/2017/05/27/could-the-uk-be-about\-to-break-end-to-end-encryption/

[40] Mull, J.: Aes ige encryption. https://mgp25.com/AESIGE/

[41] Oates, J.: Youth jailed for not handing over encryption password. https://www.theregister.co.uk/2010/10/06/jail_password_ripa/

[42] O'Neill, A., Peikert, C., Waters, B.: Bi-deniable public-key encryption. In: Advances in Cryptology – CRYPTO. pp. 525–542

[43] Owen, M.: Invisible kismet imessage exploit used to hack journalists' iphones. https://appleinsider.com/articles/20/12/21/invisible-kismet-imessage-exploit-\used-to-hack-journalists-iphones

[44] Pang, H., Tan, K.L., Zhou, X.: Stegfs: a steganographic file system. In: IEEE International Conference on Data Engineering. pp. 657–667 (2003)

[45] Pegg, D., Cutler, S.: What is pegasus spyware and how does it hack phones? https://www.theguardian.com/news/2021/jul/18/what-is-pegasus\-spyware-and-how-does-it-hack-phones

[46] Peters, T., Gondree, M., Peterson, Z.N.J.: DEFY: A deniable, encrypted file system for log-structured storage. In: ISOC Network and Distributed System Security Symposium (2015)

[47] Raja, A.: 10 best free hidden spy apps for android. Tech Times (2019)

[48] Sahai, A., Waters, B.: How to use indistinguishability obfuscation: Deniable encryption, and more. In: ACM Symposium on Theory of Computing. p. 475–484 (2014)

[49] Signal. https://signal.org/en/

[50] Technology preview: Sealed sender for signal. https://signal.org/blog/sealed-sender/

[51] Sun, H., Sun, K., Wang, Y., Jing, J.: TrustOTP. In: ACM Conference on Computer and Communications Security (2015)

[52] Sun, H., Sun, K., Wang, Y., Jing, J., Jajodia, S.: TrustDump: Reliable memory acquisition on smartphones. In: European Symposium on Research in Computer Security. pp. 202–218 (2014)

[53] Telegram messenger. https://telegram.org

[54] ThreatFabric: Vultur, with a v for vnc. https://www.threatfabric.com/blogs/vultur-v-for-vnc.html#brunhilda

[55] TrustedFirmware.org: Op-tee documentation. https://optee.readthedocs.io/en/latest/ (2019)

[56] Unger, N., Goldberg, I.: Improved strongly deniable authenticated key exchanges for secure messaging. In: Privacy Enhancing Technologies Symposium. pp. 21 – 66 (2018)

[57] Victor, D.: Forced searches of phones and laptops at u.s. border are illegal, lawsuit claims. https://www.nytimes.com/2017/09/13/technology/aclu-border-patrol-lawsuit.html

[58] Ying, K., Thavai, P., Du, W.: Truz-view: Developing trustzone user interface for mobile os using delegation integration model. In: ACM Conference on Data and Application Security and Privacy. pp. 1–12 (2019)

[59] Zhao, S., Zhang, Q., Qin, Y., Feng, W., Feng, D.: Sectee: A software-based approach to secure enclave architecture using tee. In: ACM Conference on Computer and Communications Security. p. 1723–1740 (2019)

[60] Zheng, X., Yang, L., Ma, J., Shi, G., Meng, D.: TrustPAY: Trusted mobile payment on security enhanced ARM TrustZone platforms. In: IEEE Symposium on Computers and Communication (2016)

[61] Zhou, X., Pang, H., Tan, K.L.: Hiding data accesses in steganographic file system. In: IEEE International Conference on Data Engineering (2004)