Tabellion: Secure Legal Contracts on Mobile Devices

Saeed Mirzamohammadi
CS, UC Irvine
saeed@uci.edu

Yuxin (Myles) Liu
CS, UC Irvine
yuxil11@uci.edu

Tianmei Ann Huang
Law, UC Irvine
tianmeah@lawnet.uci.edu

Ardalan Amiri Sani
CS, UC Irvine
ardalan@uci.edu

Sharad Agarwal
Microsoft Research
Sharad.Agarwal@microsoft.com

Sung Eun (Summer) Kim
Law, UC Irvine
skim@law.uci.edu

ABSTRACT

A legal contract is an agreement between two or more parties as to something that is to be done in the future. Forming contracts electronically is desirable since it is convenient. However, existing electronic contract platforms have a critical shortcoming. They do not provide strong evidence that a contract has been legally and validly created. More specifically, they do not provide strong evidence that an electronic signature is authentic, that there was mutual assent, and that the parties had an opportunity to read the contract. We present Tabellion, a system for forming legal contracts on mobile devices, such as smartphones and tablets, that addresses the above shortcoming. We define four secure primitives and use them in Tabellion to introduce self-evident contracts, the validity of which can be verified by independent inspectors. We show how these primitives can be implemented securely in the Trusted Execution Environment (TEE) of mobile devices as well as a secure enclave in a centralized server, all with a small Trusted Computing Base (TCB). Moreover, we demonstrate that it is feasible to build a fully functional contract platform on top of these primitives. We develop ~15,000 lines of code (LoC) for our prototype, only ~1,000 of which need to be trusted. Through analysis, prototype measurements, and a 30-person user study, we show that Tabellion is secure, achieves acceptable performance, and provides slightly better usability than the state-of-the-art electronic contract platform, DocuSign, for viewing and signing contracts.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; **Trusted computing**; **Virtualization and security**; **Operating systems security**; **Authentication**; **Biometrics**; • **Applied computing → Law**; • **Networks → Time synchronization protocols**.

KEYWORDS

legal contract, electronic contract, electronic signature, mobile device, trusted computing, Trusted Execution Environment (TEE)

ACM Reference Format:
Saeed Mirzamohammadi, Yuxin (Myles) Liu, Tianmei Ann Huang, Ardalan Amiri Sani, Sharad Agarwal, and Sung Eun (Summer) Kim. 2020. Tabellion: Secure Legal Contracts on Mobile Devices. In *The 18th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '20), June 15–19, 2020, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3386901.3389027

1 INTRODUCTION

Forming contracts electronically is desirable in many transactions, including real estate sales and leases, venture capital investments, and work for hire, due to its significant convenience compared to traditional methods. Not surprisingly, the global market for electronic signatures and contracts is predicted to grow significantly in the next couple of years. One report estimates the market to grow to $4.01 billion by 2023 from $844.7 million in 2017 [9]. Another report estimates the market to grow to $3.44 billion by 2022 from $517 million in 2015 [22]. The unfortunate COVID-19 outbreak in 2020 and the resulting social distancing approach deployed to combat it has further accelerated the use of electronic signatures and contracts [77].

As a result of this growth, many electronic contract platforms have emerged [11, 18, 20, 23, 28, 32, 33]. While these platforms are convenient to use, unfortunately, they have an important shortcoming: they do not provide strong evidence that a contract has been legally and validly created (as defined by the law of contracts). More specifically, they do not provide strong evidence that the signatures are authentic, that there was mutual assent, and that the parties had an opportunity to read the contract. As an example, in a recent US court case [6], the court was unconvinced that an electronic signature performed with DocuSign [18] was adequate as it could be manipulated or forged with ease.

In this paper, we present a system solution to provide *strong evidence (i.e., hard-to-fabricate and hard-to-refute evidence)* for the legal and valid formation of a contract on mobile devices. Our system, called Tabellion[1], leverages the Trusted Execution Environment (TEE) on mobile devices and an SGX enclave in a centralized server. Doing so, however, raises four research questions that we answer in this paper.

Q1. How can the contract platform provide strong evidence for all the requirements of a legal contract? We answer this question in three steps. First, we introduce four secure primitives, three for

[1]Merriam-Webster dictionary defines tabellion as (1) "a scrivener under the Roman Empire with some notarial powers," and (2) "an official scribe or notary public especially in England and New England in the 17th and 18th centuries."

client devices (secure photo, secure timestamp, and secure screenshot) and one for the centralized server (secure notarization). These primitives can be used to generate strong evidence for a legal and valid contract. Second, we introduce a secure contract protocol to use these primitives to form a contract. Finally, we introduce self-evident contracts, which contain all the required evidence for the legal and valid formation of the contract. That is, each user, and if needed, the court or an adjudicator, can independently verify the contract compliance with applicable law requirements.

Q2. Can the aforementioned primitives be realized securely, i.e., with a small Trusted Computing Base (TCB)? Having a small TCB for the secure primitives is important as it makes them less prone to software bugs (which can get exploited by attackers), and hence makes the evidence they help generate stronger. Moreover, a smaller code base can be easily inspected for safety and even certified. We show that these seemingly complex primitives can be implemented with ~1,000 LoC (out of ~15,000 LoC that we developed in our prototype). To achieve this, we introduce several novel solutions including a solution to secure the camera photo buffer, a delay-resistant Network Time Protocol (NTP), and a solution to secure the framebuffer.

Q3. Given that a contract platform provides complex functionalities, e.g., contract viewing, contract submission, and negotiations, can a fully functional platform be realized using the aforementioned primitives and protocol? We answer this question positively and build a fully functional contract platform on top of these primitives and protocol. We discuss how we address several challenges in realizing the required functionality without adding any more trusted code. Indeed, we show that Tabellion's design enables us to add a capability, called *negotiation integrity tracking*, that no existing platform supports.

Q4. Does Tabellion provide strong protection against attacks on a legal contract? We answer this question positively in two parts. First, we define the set of possible attacks on a contract platform including repudiation attacks, impersonation attacks, and confusion attacks. Second, we evaluate the security of Tabellion using a detailed security analysis and show that Tabellion can provide a strong defense against these attacks.

We design and build Tabellion for a mobile-first world where contracts are executed on smartphones and tablets. It is becoming more common to use mobile devices to sign contracts such as mortgages, vehicle leases, and bank loans [46, 67, 75, 76]. Indeed, signing contracts on mobile devices is believed to be the "the future of loans and mortgages" according to a recent study [67]. We leverage the latest mobile research technologies, including the use of TEE and secure biometric sensors now available on modern devices.

We implement Tabellion on a HiKey development board as it allows us to program the TEE (based on TrustZone and virtualization hardware). While necessary for performance measurements and security analysis, we cannot effectively use this board for a user study and energy measurements. Therefore, we build a second prototype of Tabellion on real mobile devices for that part of the evaluation. Since the TEE on these devices is not yet programmable by non-vendors, we emulate the secure primitives in Tabellion's application.

We extensively evaluate Tabellion. We measure the time it takes to carry out various steps of the contract. We show that the execution time of these operations in Tabellion is in the order of several seconds (20 to 35 seconds for very large contracts), which is small enough for a good user experience. We also show that using Tabellion does not consume a noticeable amount of energy. Finally, we evaluate the usability of Tabellion with a 30-person user study. We show that, compared to DocuSign (the state-of-the-art electronic contract platform today), Tabellion provides slightly better convenience (for contract viewing and signing), readability, and understanding of the contract. It also enables the users to read and sign the contracts slightly faster. This demonstrates that improved security in Tabellion does not come at the cost of usability.

2 BACKGROUND

The law of contracts enumerates several requirements for the correct formation of a contract between an offeror and an offeree [45]. Some requirements are beyond the scope of Tabellion as they are concerned with the content of the contract or with the circumstances of the parties involved. For example, the law of contracts requires *consideration*, which states that the contract must represent "bargained for" exchange by both sides of a contract [45]. As another example, the law requires that the parties have not signed the contract under duress and that the parties have legal capacity (e.g., they are of legal age at the time of assent) [45].

There are, however, key requirements that are related to the contract platform. One key requirement in the case of contracts that are required to be in writing and signed is *signature attribution*, i.e., that a signature is an authentic signature of the party being charged. Another key requirement is *mutual assent*, which requires that the two parties agree to the same contract. Mutual assent has clear conditions in the law: (*i*) there is an offer from the offeror, (*ii*) there is an acceptance by the offeree, (*iii*) there is no revocation of the offer from the offeror before the acceptance by the offeree, and (*iv*) there is no rejection or counter-offer from the offeree before acceptance [45]. In case of a counter-offer by the offeree, this counter-offer is considered a fresh offer, which may be accepted or rejected by the other party.

Another requirement is that a party has had an opportunity to read a contract. However, it is not a requirement that a party has actually read the contract. In the case of *O'Connor v. Uber Technologies, Inc.* [5], the plaintiffs asserted that there was no valid agreement because it was displayed on "*a tiny iPhone screen when most drivers are about to go on-duty and start work.*" The court rejected that argument because "*for the purposes of contract formation, it is essentially irrelevant whether a party actually reads the contract or not, so long as the individual had a legitimate opportunity to review it.*"

Electronic contracts. The Uniform Electronic Transactions Act (UETA) and the Electronic Signatures in Global and National Commerce (ESIGN) Act permit the use of electronic signatures and contracts. ESIGN states that: "*a signature, contract, or other record relating to such transaction [any transaction in or affecting interstate or foreign commerce] may not be denied legal effect, validity, or enforceability solely because it is in electronic form*" [1]. This means the courts would not reject a contract simply on the basis of it being signed electronically. However, it is up to the contract platform designer to provide a secure solution—one that can be effectively defended in courts. For example, simply printing a contract, signing it using a wet signature, and scanning it does not provide strong evidence for attribution as it is easy to copy/paste (or even forge) the scanned signature. While no platform can guarantee that its contracts will be legally valid with certainty, a platform can increase the odds of success by providing strong and secure evidence for the correctness of its contracts.

Legal cases on electronic contracts. We next discuss a few legal cases to demonstrate the shortcomings of existing electronic contract platforms. First, existing platforms do not provide strong evidence for signature attribution. For example, in *In re Mayfield*, a recent California bankruptcy case [6], the court was unconvinced that an electronic signature created using DocuSign was adequate as it could be manipulated or forged with ease, stating: "*This brings the court to another important problem with Counsel's arguments: they do not address the ease with which a DocuSign affixation can be manipulated or forged. The UST [United States Trustee] asks what happens when a debtor denies signing a document and claims his spouse, child, or roommate had access to his computer and could have clicked on the 'Sign Here' button.*"[2]

Second, existing platforms have failed to provide evidence that the parties agreed to the same contract. For example, in the case of *Adams v. Quicksilver, Inc.* [4, 10], the plaintiff challenged the validity of an arbitration agreement: "*The system provided no audit trail for the signing process, though, so it couldn't be determined when the agreement was signed.*"

Third, existing platforms do not provide evidence that the parties had an opportunity to read the contract. This is evident in the case of *Labajo v. Best Buy Stores* [3, 62]: "*When Christina [the plaintiff] accepted the free subscription, she signed an electronic signature pad at Best Buy [the defendant]. Christina claimed that there was no disclosure telling her she would be charged for the magazine. But whether or not there was disclosure doesn't matter. What matters is the fact that Best Buy couldn't prove that she saw and approved the disclosure.*"

3 ATTACKS ON LEGAL CONTRACTS

We define potential attacks on a legal contract system by either a malicious offeror or a malicious offeree.

Repudiation attack[3]**.** In this attack, either the offeror or offeree denies having agreed to the contract (when in fact they did). This attack can come in three different forms. First, one party may deny having legally signed[4] the contract. Second, an attacker may deny mutual assent. That is, the parties might disagree on the terms of the contract or on the version of the contract they signed as a result of negotiations (i.e., counter-offers followed by revisions). Moreover, the offeror may claim to have revoked the offer before acceptance by the offeree or the offeree may claim to have rejected or countered it before acceptance. Third, the offeree may claim that they were not given an opportunity to read the contract.

Impersonation attack. In this attack, one party attempts to impersonate someone else and sign a contract on their behalf. There are two forms of this attack. First, the attacker spoofs the victim's authentication on the victim's device. Second, attacker spoofs the victim's identity.

Confusion attack. In this attack, the offeror attempts to fool the offeree into legally signing a contract different from what the offeree thinks they are legally signing. To perform this attack, an attacker needs to target and/or compromise the contract viewer on the offeree's device in order to misrepresent the contract to the offeree. In one special form of this attack, called the Dalì attack [40, 41, 70], the attacker submits a file for the contract in an interpreted document format (e.g., PDF) with dynamic content, which shows different content on the offeror's and offeree's devices.

[2]We note that in this case, the court was unwilling to accept software-generated electronic signatures as substitutes for wet signatures due to the California bankruptcy court's local rules of practice. As mentioned, the ESIGN Act provides that a signature cannot be rejected solely because it is in electronic form. However, that provision of the ESIGN Act does not apply to "*court orders or notices, or official court documents.*" [1]

[3]This is different from the concept of repudiation in the law of contracts, which refers to actions demonstrating that one party to a contract refuses to perform a duty or obligation owed to the other party.

[4]There are two types of signatures in our context. We use "legal signature" or "electronic signature" to refer to a user's assent to a contract, and "cryptographic signature" or "digital signature" to refer to a computer-generated hash that is signed by a private key.

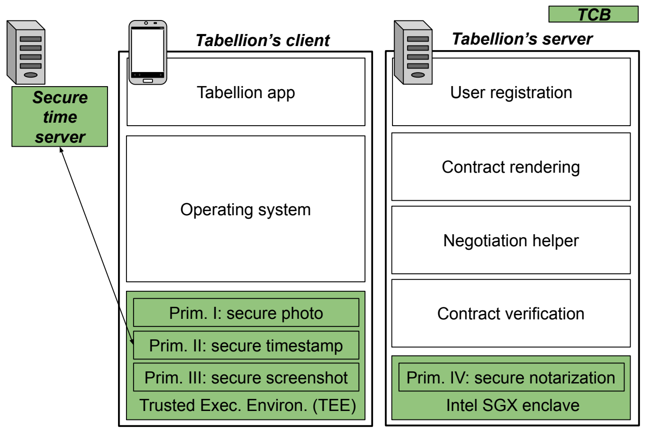


Figure 1: ***The client and server in Tabellion.***

4 TABELLION: PRINCIPLES AND DESIGN

We present Tabellion, a legal contract platform that provides strong evidence for the legal formation of a contract. Note that while we focus on two-party contracts for clarity, Tabellion can similarly handle multi-party contracts. Tabellion comprises two components: a client that runs on the mobile devices of the offeror and the offeree, and a centralized server that mediates the contract formation. Figure 1 illustrates these two components. Each incorporates some code in their TCB to implement the required secure primitives. Each also incorporates a large amount of untrusted code to implement the functionality needed to form a contract. In this section, we introduce the primitives (§4.1), discuss the contract formation protocol (§4.2), the resulting self-evident contract (§4.3), and the contract verification process (§4.4).

4.1 Secure Primitives

Based on the requirements of a valid and legal contract, we define a set of secure primitives to generate strong evidence for the contract.

Primitive I. Tamper-proof camera-captured photo (i.e., secure photo). Signature attribution in written agreements requires evidence of the identity of the signatory. Photos taken of the user can provide such evidence. Indeed, several existing identity-based systems, such as Voatz [34], capture photos and videos of the user and use them for identification. We define photos as the main primitive since videos are simply a collection of photos.

The key property of this primitive is that the photo must be captured by the camera hardware as opposed to being fabricated by software. The other property of this photo is that the photo must be tamper-proof after capture. We achieve these by reading the photo directly from the camera in the TEE and by cryptographically signing the photo. For the latter, we use a per-user per-device private key, which is generated in the TEE once the user registers with Tabellion on a mobile device (discussed in §4.2).

Primitive II. Tamper-proof global timestamp (i.e., secure timestamp). The mutual assent requirement of the law of contracts requires evidence of both an offer and an acceptance of that offer. Moreover, one needs to show that there was no revocation of the offer from the offeror before the acceptance by the offeree and that there was no rejection or counter-offer from the offeree before acceptance. To achieve this, we require the client devices to be able to attach a global timestamp, e.g., a timestamp with respect to a global clock, to each action (i.e., offer, acceptance, revocation, rejection, and counter-offer). The timestamps must be tamper-proof. That is, an attacker should not be able to spoof or modify them. We achieve this using a novel clock synchronization protocol that allows the TEE on the device to securely synchronize its clock with a trusted time server.

Primitive III. Tamper-proof user-confirmed screenshot (i.e., secure screenshot). The requirement of having the opportunity to read a contract in the law of contracts requires evidence of the contract having been presented to the offeree. We use screenshots of the contents displayed on the client device to achieve this goal. However, not all content displayed on the device is seen by the user. Therefore, we ensure that the user *confirms* seeing the content captured in the screenshot. We also ensure that the screenshots are tamper-proof.

To achieve these, we ask the user to authenticate with the system in order to confirm seeing the displayed content. Upon successful authentication, we use the aforementioned per-user per-device private key to cryptographically sign the screenshot. Note that it is critical that the acts of seeing the content on the display and providing authentication are atomic. If not, an attacker can show some content to the user but have the user unknowingly confirm seeing a different content.

Different authentication solutions can be used. We use biometric authentication, e.g., fingerprint authentication, as it is easy to use, is available on most modern mobile devices, and has high accuracy [2, 29].

Primitive IV. Secure notarization of the contract. This primitive securely connects all the evidence in a contract together so that the evidence cannot be maliciously deleted or reused for another contract, and so that new evidence cannot be added to a contract after it is finalized. We achieve these goals with a secure enclave in Tabellion's server, which acts as a notary by binding all the evidence together and cryptographically signing them.

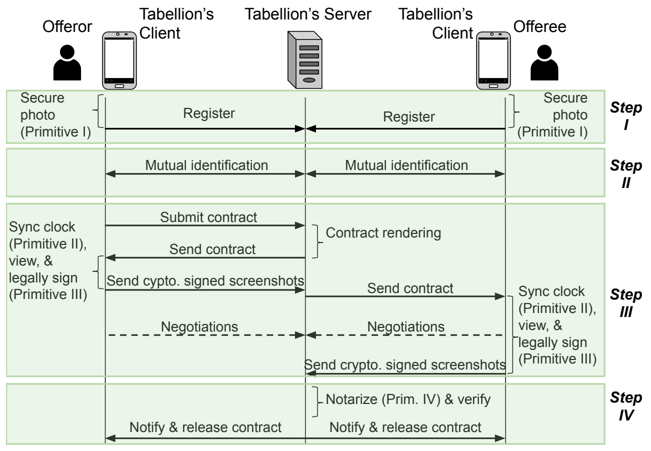

Offeror
Tabellion's Client
Tabellion's Server
Tabellion's Client
Offeree
Secure photo (Primitive I)
Register
Register
Secure photo (Primitive I)
Step I
Mutual identification
Mutual identification
Step II
Submit contract
Contract rendering
Sync clock (Primitive II), view, & legally sign (Primitive III)
Send contract
Send cypto. signed screenshots
Send contract
Sync clock (Primitive II), view, & legally sign (Primitive III)
Negotiations
Negotiations
Send crypto. signed screenshots
Step III
Notarize (Prim. IV) & verify
Notify & release contract
Notify & release contract
Step IV

Figure 2: *Tabellion's contract formation protocol.*

4.2 Contract Formation Protocol

Figure 2 illustrates the four steps of the protocol.

Step I: Registration. Both the offeror and offeree first register with Tabellion. In this step, Tabellion's client uses Primitive I to take a photo of the user. It then sends the cryptographically signed photo to the server. The client also sends some additional information to the server, which is needed for self-evident contracts (§4.3) including the device TEE certificate and the measurement of the TEE code (which is the certified hash of the TCB code).

The user needs to register with Tabellion once upon installation on a new device. During registration, the client TEE creates a *per-user per-device key pair*. It uses the private key of this pair to sign the user's photo (Primitive I) and uses the same key later to sign the screenshots captured of the content of the contract (Primitive III). It also sends the public key to Tabellion's server for verification.

The TEE code uses biometric-based authentication for the use of the key. That is, the user has to use their biometric (e.g., fingerprint) to confirm the securely captured photo and/or screenshot so that they are signed by the TEE. This strongly binds the user's photo and the confirmed screenshots to each other. That is, if a screenshot and a photo are signed with the same per-user per-device private key, one can conclude that the person in the photo confirmed and signed the contract in the screenshots (see §8.2 for a discussion of impersonation attacks and Tabellion's solutions).

Note that, instead of using a separate registration phase, it is possible to securely capture the user's photo for each contract or even for each page in the contract that the user confirms. However, this approach would impose a usability burden.

Step II: mutual identification. When the offeror requests Tabellion to initiate the process of forming a contract with the offeree, Tabellion asks the two parties to confirm each other's identities. To initiate a contract, the offeror names the offeree using a unique identifier, e.g., an email address registered with Tabellion. At this point, Tabellion asks the offeree whether they are expecting a contract from a named offeror (e.g., using an email address as the identifier). Once approved by the offeree, Tabellion uses the aforementioned securely-captured photos of the two parties to show to them. To ensure privacy, Tabellion does not exchange the photos before both parties approve having the intention of forming a contract with each other.

Step III: Legally signing a contract. In this step, the offeror first submits a contract to the server, which sends it to both parties to collect their legal signatures. Tabellion's server renders the contract into its own customized format, as described in §6.1, before sending it to the parties. The server also sends the certificate of the notary enclave to both parties so that they can include it in their signed screenshots (needed to prevent reuse of the screenshots). To legally sign the contract, Tabellion's mobile application first asks the TEE to use Primitive II to synchronize its clock (which will be needed to generate secure timestamps). It then uses Primitive III to display the contract to the user page by page and capture screenshots of content seen on the display. Note that this step may involve negotiations between the parties as discussed in §6.3. Finally, the application shows a special last page to each of the users that explicitly asks them to assent to (and hence sign) the contract (again using Primitive III).

Step IV: Notarizing the contract. Tabellion's server uses secure Primitive IV to cryptographically sign the contract as well as all the collected evidence. It then verifies the contract and releases it to both parties.

4.3 Self-Evident Contracts

Contracts in Tabellion are self-evident. That is, each user, and if needed, the court or an adjudicator, can independently verify the contract compliance with applicable law requirements.

A contract in Tabellion is formulated as

$$\{ \bigcup_{\mathtt{U \in users}} (\{\mathtt{Photo_U}\}^{\mathtt{Pr_U}}, \{\mathtt{Pu_U}\}^{\mathtt{PrD_U}}, \mathtt{CertD_U}, \mathtt{MeasureD_U}, \bigcup_{\mathtt{i \in pages}} (\{\mathtt{Screenshot_{i,U}}, \mathtt{ts_{i,U}}, \mathtt{Cert_N}\}^{\mathtt{Pr_U}})), \mathtt{Cert_N}, \mathtt{Measure_N}, \mathtt{ts_N}\}^{\mathtt{Pr_N}}$$

where $\{\mathtt{A}\}^{\mathtt{Pr}}$ indicates that A is cryptographically signed by private key Pr, and $\bigcup_{\mathtt{U \in users}} (\ldots)$ and $\bigcup_{\mathtt{i \in pages}} (\ldots)$ represent union of users and contract pages, respectively.

The formula shows the components of a contract. $\{\mathtt{Photo_U}\}^{\mathtt{Pr_U}}$ is the photo captured using Primitive I, where U denotes either the offeror or the offeree. The photos are signed by the corresponding party's per-user per-device private key ($\mathtt{Pr_U}$). To verify this key, the contract also includes the corresponding public key ($\{\mathtt{Pu_U}\}^{\mathtt{PrD_U}}$), which itself is certified by a device-specific private key in the party's corresponding device TEE ($\mathtt{PrD_U}$), and hence the contract also includes the certificates of the device TEEs of the two parties ($\mathtt{CertD_U}$). The device TEE certificate is simply the public key of the device certified by the device vendor. More specifically, this is the device-specific ARM TrustZone certificate [30]. The contract also includes the measurements of the TEE code in the devices of the two parties ($\mathtt{MeasureD_U}$). These measurements let the verifier know what software was running in the TEEs and are signed by the aforementioned device-specific keys.

The next components are the screenshots collected from the parties. The number of screenshots from each user can be different as the contract may be formatted differently for each user (§6.1). A signed screenshot also includes a timestamp captured with Primitive II ($\mathtt{ts_{i,U}}$), highlighting when the screenshot was confirmed by the user (as we discuss in §5.2, the timestamp comes with a confidence interval). It also includes the certificate of the notary ($\mathtt{Cert_N}$). The latter is to ensure that each signed screenshot can be used for one contract only. Without it, an attacker may take a screenshot from a contract signed by a victim and try to include that in a different contract by the same victim.

The last couple of components are related to the notary. This includes the certificate of the notary, which is the certificate of the Intel SGX enclave [24]. It also includes the measurement of the code in the enclave code ($\mathtt{Measure_N}$) and the time of notarization ($\mathtt{ts_N}$). Finally, the contract is signed by the notary's private key ($\mathtt{Pr_N}$).

4.4 Contract Verification Process

In Tabellion, one can verify the contract compliance with applicable law requirements as follows. To verify signature attribution, one needs to check that the same per-user per-device private key ($\mathtt{Pr_U}$) is used to sign the photo ($\{\mathtt{Photo_U}\}^{\mathtt{Pr_U}}$) and the screenshots ($\{\mathtt{Screenshot_{i,U}}, \ldots\}^{\mathtt{Pr_U}}$). Moreover, as described in §8.2, in Tabellion, we require a certain gesture to be performed in the photo to detect awareness. Therefore, one needs to check the presence of this gesture too.

To verify mutual assent, one needs to (*i*) check the content of the contract screenshots ($\mathtt{Screenshot_{i,U}}$) of the two users to make sure they both have the same content and (*ii*) check the timestamps of the screenshots ($\mathtt{ts_{i,U}}$), which include negotiations details (§6.3), to verify the order of the actions.

To verify that parties had an opportunity to read the contract, one needs to check that all contract pages are signed with the per-user per-device private key ($\{\mathtt{Screenshot_{i,U}}, \ldots\}^{\mathtt{Pr_U}}$).

In addition, one needs to perform several more correctness checks. More specifically, one needs to check the public key of each user ($\mathtt{Pu_U}$), to make sure all the cryptographic signatures by the clients are valid; check the certificate of the devices ($\mathtt{CertD_U}$), to make sure that the users used a device verified by its vendor; check the certificate of the notary ($\mathtt{Cert_N}$), to make sure a real enclave was used for notarization; check the software measurements of the device TEEs and the notary ($\mathtt{MeasureD_U}$ and $\mathtt{Measure_N}$), to make sure they used the expected code; check the inclusion of the notary certificate in the signed screenshots ($\{\ldots, \mathtt{Cert_N}\}^{\mathtt{Pr_U}}$), to make sure the screenshots were not reused from another contract; check the notarization timestamp ($\mathtt{ts_N}$), to make sure it is larger than the timestamps of all screenshots; and check the notary signature on the contract ($\mathtt{Pr_N}$), to make sure the contract is correctly sealed.

5 SECURE REALIZATION OF PRIMITIVES

Tabellion's self-evident contract assumes secure and untampered execution of the primitives. Therefore, it is critical to implement these primitives with a small amount of code so that their TCB remains small. A small TCB makes the primitives less prone to software bugs (which can get exploited by attackers). Moreover, a smaller code base can be easily inspected for safety and even certified. Table 1 shows that, while Tabellion needs a large amount of code (~15,000 LoC) to implement all of its functionality, the size of trusted code is small (~1,000 LoC). In this section, we discuss

Tabellion's Component	Trusted Code		Untrusted Code	
	Component	Size	Component	Size
Client	Primitive I	166	Mobile App	9919
	Primitive II	104		
	Primitive III	80		
	Shared	291		
Server	Primitive IV	185	Rest	4180
Combined		**826**		**14099**

Table 1: Tabellion's trusted and untrusted code size. The sizes are reported in LoC. We count the lines of the code we added, but not the existing code, e.g., TEE OS or Android libraries.

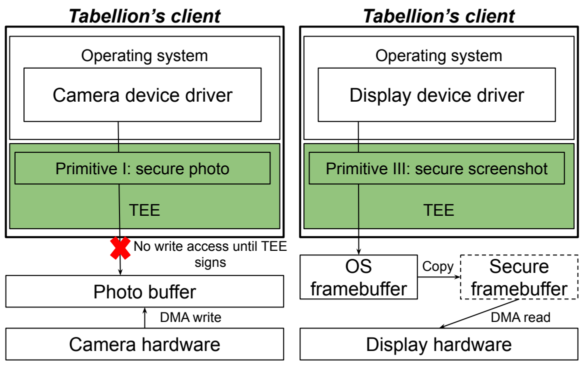


Figure 3: ***(Left) Secure realization of Primitive I. (Right) Secure realization of Primitive III.***

some of the important challenges we faced and solved to achieve this goal.

5.1 Primitive I: Secure Photo

Challenge. One straightforward way to implement this primitive is to exclusively control the camera in the TEE (a feature supported by ARM TrustZone). In this case, the TEE can directly take the photo and sign it. Unfortunately, this approach significantly bloats the TCB as it requires moving the camera device driver to the TEE. For example, in the Nexus 5X smartphone, the size of the camera driver is 65,000 LoC.

Solution. Our key idea to solve this problem is for the TEE to protect the camera photo buffer (rather than the whole camera driver) in memory from the time that the camera is about to capture the photo until when it is cryptographically signed. To protect the camera photo buffer, Tabellion write-protects the buffer pages before the camera device populates them with the photo data using Direct Memory Access (DMA). Moreover, to prevent the untrusted OS from storing a fake image in the camera photo buffer before protection, Tabellion zeroes out the buffer right after protection. Figure 3 (Left) illustrates this solution.

We implement two APIs in the TEE for this purpose. The application calls the normal OS photo capture API and the OS kernel uses the TEE API to capture a secure photo and returns it to the application. The kernel calls the first API, `prepare_photo_buffer`, to register a memory buffer to be used for secure photo capture. This API takes one argument, `photo_buf_paddr`, which is the physical address of the photo buffer in the OS physical address space. This API write-protects the buffer and zeroes out its contents. The kernel then waits for the camera hardware to populate this buffer through DMA. Next, the kernel calls the second API, `show_photo_buffer`, which displays the photo on the screen. Finally, the kernel calls the third API, `sign_photo_buffer`, which waits for the user to confirm the photo on the screen, and then cryptographically signs the photo and returns it using shared memory.

Note that to protect against the attacker using another DMA-enabled device to write to the photo buffer, Tabellion can use IOMMUs available in ARM SoCs, similar to SchrodinText [36].

5.2 Primitive II: Secure Timestamp

The TEE needs to be able to use a secure clock synchronized with a global clock. Network Time Protocol (NTP) is a popular protocol that can be used for clock synchronization. For security, we assume and use an integrity-protected channel (i.e., signed messages) to communicate with a secure NTP server, such as [14], to prevent a man-in-the-middle attack, which may tamper with the messages. This makes sure that the OS (or an attacker in the network) cannot change the content of the messages.

Challenge. Unfortunately, this security provision is not enough and an attacker can still mount an *asymmetric delay attack*. That is, the OS (or an attacker in the network) can delay the outgoing and incoming messages (from the TEE to the secure NTP server) in order to tamper with NTP calculations. We next describe this attack in more detail and then provide our solution.

In NTP, the client calculates its clock offset from the NTP server's clock as $\frac{(t_2-t_1)+(t_3-t_4)}{2}$, where t_1 and t_4 are timestamps captured by the client when it first sends a message to the NTP server and when it receives a response, and t_2 and t_3 are the timestamps captured by the NTP server when it first receives a message from the client and when it sends a response (which sends t_2 and t_3 to the client). This offset can then be used to synchronize the client clock. The NTP protocol assumes that the time to send a message from the client to the NTP server is the same as the time to send a message from the server to the client (hence the divide by two). An attacker can inject an *asymmetric delay* into one of these messages, e.g., using a compromised OS on the client or a compromised network link, in order to tamper with the calculated offset.

Solution. To address this challenge, Tabellion uses a novel secure clock synchronization strategy, built on top of NTP, which we call *delay-resistant NTP*. Our solution defeats the asymmetric delay attack by calculating a confidence interval, which represents the maximum and minimum possible offsets assuming arbitrary delay in any of the messages. Tabellion tags each action with its timestamp and confidence interval. For mutual assent, in addition to ordering the timestamps, Tabellion's contract verification requires that confidence intervals be non-overlapping.

To calculate the interval, we assume two extreme cases, one where only the request from the client to the NTP server forms the full round trip time (and the response takes no time) and one vice versa. It is possible to show that $\mathtt{offset}_{max} = \max(t_3 - t_4, t_2 - t_1)$ and $\mathtt{offset}_{min} = \min(t_3 - t_4, t_2 - t_1)$. Therefore, the confidence interval (ci) is calculated as $\mathtt{ci} = |(t_3 - t_4) - (t_2 - t_1)|$.

We add two APIs to the TEE for this primitive. They allow the application to initiate the protocol and to communicate with the

NTP server. The application calls the first API, `sync_clock_init`, to initiate the protocol. This API returns a nonce from the TEE (used to protect against replay attacks). The application then forwards the nonce to the NTP server and forwards the response from the NTP server to the TEE with a call to the second API, `sync_clock_complete`. This API takes one argument, `server_ts`, which is a shared buffer for passing the two server timestamps in NTP protocol and the server's signature (RSA with a 1024 bit key).

Note that in addition to a secure synchronization mechanism, the TEE needs a secure hardware timer to keep track of time after synchronization. For that, we use a secure hardware timer available in TrustZone.

5.3 Primitive III: Secure Screenshot

Challenge. The TEE needs to securely capture the content on the display. A straightforward approach to achieve this is to give exclusive control of the display subsystem to the TEE. Unfortunately, doing so requires moving the display subsystem driver to the TEE, which on the HiKey board, encompasses at least 8,000 LoC. This bloats the TCB.

Solution. Our key idea in Tabellion is to secure the buffer used for displaying content (i.e., framebuffer) in the TEE rather than the whole display software stack. At a high-level, the primitive is realized as follows. When invoked, the TEE *freezes* the framebuffer, not allowing any more updates. It then waits for the user's authentication using biometrics. Once the user confirms, the TEE signs a copy of the framebuffer and *unfreezes* it. This process guarantees that the displayed content and the authentication are atomic.

Disallowing updates to the framebuffer can break the display stack in the OS. Therefore, the TEE copies the contents of the framebuffer to a newly generated framebuffer (i.e, secure framebuffer), which is only accessible in the TEE. It then shows the secure framebuffer on the display by programming its address into the memory-mapped display controller register that holds the address of the framebuffer. Furthermore, to prevent a compromised OS from overwriting this register (in order to use a different framebuffer), Tabellion also removes write permission from the corresponding register page of the display controller. With this solution, the OS is allowed to update its own framebuffer but doing so does not change the content shown on the display. Upon unfreezing the display, Tabellion points the display controller back to the untrusted framebuffer and enables writes to the aforementioned display controller register. Figure 3 (Right) illustrates this approach.

Note that when the TEE removes the permission of the above register page, any write to this register page of the display controller would fault. Indeed, there are other registers on the page as well, all of which will be write-protected. To avoid these faults, we made minimal changes to the display controller driver in the OS to skip the writes while the display is frozen.

To provide this primitive, the TEE exposes three APIs. It expects the application (through the OS) to call these APIs. First, to show a contract page, the application displays the page and then makes a call to TEE's *freeze_framebuffer*, which freezes the framebuffer showing the contract page. The TEE waits for the user's confirmation using biometrics. Note that in commodity mobile devices, biometric devices are controlled in the TEE [16], therefore, we assume so in our design. Once the user confirms, the application makes a call to the second API, *sign_framebuffer*, which captures a screenshot in the TEE and cryptographically signs it, appending the secure timestamps and the notary certificate passed with the API. This API takes the notary certificate as input and returns the signed framebuffer using shared memory. Finally, the application makes a call to the third API, *unfreeze_framebuffer*, which unfreezes the framebuffer.

As one last provision, we require each page shown through this primitive to stay on the display for a minimum of 2 seconds. This prevents a user from confirming a page by mistake without having enough time to read it.

5.4 Primitive IV: Secure Notarization

Challenge. Collecting one piece of evidence poses a challenge for the notary. Specifically, at the time of notarizing the contract, the notary enclave does not know for certain whether the offeror has revoked the offer or not. The law of contracts recognizes the offeror's right to revoke the offer as long as it is not signed by the offeree and this revocation is otherwise permitted under the terms of the offeror's offer. For example, the offeror might have revoked the offer 5 minutes prior to the offeree legally signing the contract (hence not satisfying the requirement of mutual assent), but the revocation evidence might not reach the notary in time.

Solution. To address this problem, the enclave requires a confirmation from the offeror that there have been no revocations, and if there has been one, it requires the secure screenshot confirming that. The enclave, through the rest of the server code, inquires about any pending revocations in the offeror and waits until it receives the response.

Note that this solution, while secure, might cause a practical problem. That is, no response from the offeror's device can stall the notarization of the contract indefinitely. To prevent indefinite blocking, a possible approach is to wait for no longer than a configurable period of time, e.g., 24 hours. This allows the offeror's device to send the response in most practical cases.

6 FULLY FUNCTIONAL PLATFORM

The secure primitives that we design and build for Tabellion, while effective in providing strong evidence for the contract, pose challenges for building a fully functional contract platform. We discuss all the challenges we faced and solved by developing ~14,000 untrusted lines of code in our client and server.

6.1 Readable Contracts

Challenge. Contracts are often in PDF or Word formats, with each page containing a large amount of text. When viewed on the display of a mobile device, users need to continuously *pinch and zoom* to fully view the content. Doing so creates challenges for the use of Primitive III (secure screenshot) to capture all the content seen by the user.

Solution. We solve this problem by providing a service in Tabellion's server that renders the contract into a readable one when viewed fullscreen on the mobile device of each user. This service accepts a contract in an intermediate format (Markdown language in our prototype). It generates the contract pages specifically for the screen sizes of the mobile devices used by the offeror and the offeree. Moreover, the service follows some formatting guidelines for the rendered document to make sure the content is easily readable. These guidelines include adequate line spacing, margins, clear background color, and font colors. Indeed, it is possible to add a feature to the service so that it can apply user-specific requests, e.g., a font size larger than the default one. In addition, the server clearly marks every page with a page number, which helps verify the presence of all required screenshots in the contract.

We note that contract rendering in the server is not part of the TCB of the system. This is because Tabellion asks the user to read through and sign the generated contract.

6.2 Contract Submission

Challenge. As mentioned, Tabellion's contract generator receives the contract in an intermediate format, such as Markdown. This creates a burden for the offeror, who may prefer another format to prepare the contract in.

Solution. To address this problem, we provide a mostly-automatic converter. More specifically, we allow the offeror to submit the contract in PDF format. Our converter then extracts the content from the PDF file and converts it to a mobile app UI page with explicit headers and text sections. The offeror is then allowed to review the extracted content and edit it, if needed. Once finalized, the converter produces a Markdown file and sends it to the server. Figure 4 shows this solution with an example.

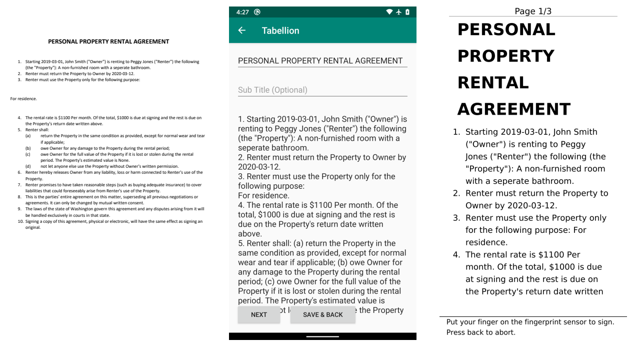
We do, however, note that our prototype cannot currently handle complex PDF pages (i.e., those with images and tables). We leave addressing this limitation to our future work.

6.3 Contract Negotiations

Challenge. When parties negotiate and vary the terms of a contract, the parties will need to mutually assent to the new terms if they are materially different from the terms of the original contract. In order to demonstrate mutual assent, one needs to provide evidence of these negotiations. Unfortunately, creating an out-of-bound channel to allow the parties to perform negotiations, e.g., messaging apps or email, require other primitives to securely capture the negotiations.

Solution. We have implemented a fully-functional negotiation interface in Tabellion using the existing primitives. To achieve this, our Tabellion application allows the offeree to enter a comment on the offer. The application then shows the comment on a new UI page to the offeree and asks them to confirm the comment, similar to how they confirm seeing a contract page. Tabellion then uses Primitive III to capture a confirmed screenshot of the revision request and includes it in the contract. Finally, this revision request is shown to the offeror, who can revise the contract and submit again, using the Android UI page described earlier and seen in Figure 4 (Middle). Note that the details of negotiations can be easily verified in a Tabellion contract by inspecting the confirmed screenshots of the contract pages and the revision requests along with their timestamps.

Opportunity. The combination of the previous three solutions has enabled us to add an important capability to Tabellion that



PERSONAL PROPERTY RENTAL AGREEMENT

1. Starting 2019-03-01, John Smith ("Owner") is renting to Peggy Jones ("Renter") the following (the "Property"): A non-furnished room with a seperate bathroom.
2. Renter must return the Property to Owner by 2020-03-12.
3. Renter must use the Property only for the following purpose:

For residence.

4. The rental rate is $1100 Per month. Of the total, $1000 is due at signing and the rest is due on the Property's return date written above.
5. Renter shall:
   (a) return the Property in the same condition as provided, except for normal wear and tear if applicable;
   (b) owe Owner for any damage to the Property during the rental period;
   (c) owe Owner for the full value of the Property if it is lost or stolen during the rental period. The Property's estimated value is None.
   (d) not let anyone else use the Property without Owner's written permission.
6. Renter hereby releases Owner from any liability, loss or harm connected to Renter's use of the Property.
7. Renter promises to have taken reasonable steps (such as buying adequate insurance) to cover liabilities that could foreseeably arise from Renter's use of the Property.
8. This is the parties' entire agreement on this matter, superseding all previous negotiations or agreements. It can only be changed by mutual written consent.
9. The laws of the state of Washington govern this agreement and any disputes arising from it will be handled exclusively in courts in that state.
10. Signing a copy of this agreement, physical or electronic, will have the same effect as signing an original.

PERSONAL PROPERTY RENTAL AGREEMENT

1. Starting 2019-03-01, John Smith ("Owner") is renting to Peggy Jones ("Renter") the following (the "Property"): A non-furnished room with a seperate bathroom.
2. Renter must return the Property to Owner by 2020-03-12.
3. Renter must use the Property only for the following purpose:
For residence.
4. The rental rate is $1100 Per month. Of the total, $1000 is due at signing and the rest is due on the Property's return date written above.
5. Renter shall: (a) return the Property in the same condition as provided, except for normal wear and tear if applicable; (b) owe Owner for any damage to the Property during the rental period; (c) owe Owner for the full value of the Property if it is lost or stolen during the rental period. The Property's estimated value is
ot l e the Property

Page 1/3

PERSONAL PROPERTY RENTAL AGREEMENT

1. Starting 2019-03-01, John Smith ("Owner") is renting to Peggy Jones ("Renter") the following (the "Property"): A non-furnished room with a seperate bathroom.
2. Renter must return the Property to Owner by 2020-03-12.
3. Renter must use the Property only for the following purpose: For residence.
4. The rental rate is $1100 Per month. Of the total, $1000 is due at signing and the rest is due on the Property's return date written

Put your finger on the fingerprint sensor to sign. Press back to abort.

Figure 4: ***(Left) A contract sample in PDF. (Middle) Extracted contract in Tabellion presented in an Android UI activity, which allows edits. (Right) Contract rendered by Tabellion and viewed on the device. The original contract is a single page with small fonts, which is barely readable on a smartphone screen in fullscreen mode. The converted contract has 3 pages and is easily readable.***

no existing platform supports: *negotiation integrity tracking*. More specifically, after the offeror edits the original offer and submits it, Tabellion's server compares the edited contract with the old version and identifies the contract pages that are affected by changes. It then asks the offeree to only view and modify these edited pages. This capability provides important usability benefits, especially when dealing with long contracts. In today's platforms, this is left to the offeree. That is, the offeree can decide to view the parts of the revised contract that they think have been updated. However, they bear the risk of not seeing other changes added (possibly maliciously) to the contract. Alternatively, they can re-read the whole revised contract again, which is time-consuming, especially if there are multiple rounds of negotiations.

6.4 Automatic Contract Verification

Tabellion's contracts are self-evident. Yet, the verification process is not easy and requires several checks. Therefore, to enhance the usability of Tabellion, we provide an automatic contract verifier. We use this verifier in our own server to verify the contract once it is formed and before notifying the users. We note that contract verification in the server is not part of the TCB of the system. This is because each user can independently verify the contract as well.

Challenge. The rendering of the contract specifically for each user creates a challenge for automatic verification. That is because the contract pages (but not the content) might be different for each user (e.g., different number of pages, different page dimensions, and different font sizes). To check that both parties assent to the same contract terms, we cannot trivially compare the two sets of screenshots pixel by pixel.

Solution. To enable automatic verification, Tabellion's server releases some metadata alongside the notarized contract. This metadata includes information about the code used to render to the contract from the intermediate language (e.g., Markdown), the contract source in that intermediate language, and the format used for each mobile device (i.e., screens size, font size, etc.). The verifier uses the same generator code to render the contract pages from the source to the final pages for each mobile devices (which are PNG images as described in §7). It then compares these rendered images, pixel by pixel, with the contract screenshots signed by the users' devices. If the images fully match, verification is successful.

7 IMPLEMENTATION

Tabellion's client. We build Tabellion's client on a HiKey LeMaker development board. The TEE in this board is the Xen hypervisor (version 4.7) and the OPTEE OS (version 3.3) running in ARM's TrustZone secure world. We implement Primitives I and III (other than the cryptographic signatures) in the Xen hypervisor. We implement cryptographic signing operations as well as Primitive II in OPTEE. We use RSA with 2048 bit keys for digital signatures in the client.

We note that virtualization hardware extension is available in most of the ARM mobile SoCs that are used in current mobile platforms and, hence, adding a hypervisor is feasible. Indeed, some mobile manufacturers have already added a hypervisor layer for security purposes. For example, Samsung uses a hypervisor for real-time kernel protection as part of Samsung Knox [30].

We use a USB camera and a USB fingerprint scanner with the board and program them in the normal world. We use Android Open Source Project (AOSP) Nougat for the untrusted OS. The TCB size in this prototype is the trusted code that we added (Table 1) and the existing trusted code in TrustZone secure world and hypervisor (which can be as low as a few tens of thousands of lines [64]).

We also provide a secondary prototype of Tabellion for commodity mobile devices. We use this prototype for our user study and for energy measurements (§9). The main difference is the implementation of secure primitives. On commodity mobile devices, we cannot program the TEE, therefore, we emulate these primitives in the mobile app itself. In this prototype, we use the smartphone's camera and fingerprint scanner.

Tabellion's server. We process the contract in Markdown format and generate the contract pages as images in PNG format. We also attach instructions and page numbers to contract page (Figure 4 (Right)).

We implement the notary enclave in an Azure Confidential Compute Standard DC4s VM. This VM runs on top of the 3.7GHz Intel XEON E-2176G processor, which supports Intel SGX. We program the enclave using the open source Confidential Consortium Framework (CCF). For the measurement of the TCB and the enclave certificate, we use the Intel SGX Data Center Attestation Primitives (DCAP) libraries, which leverage Elliptic Curve Digital Signature Algorithm (ECDSA). We use RSA with 4096 bit keys for digital signatures by the notary in the enclave.

8 SECURITY EVALUATION

8.1 Threat Model

We assume that the Tabellion client's TEE is uncompromised. We assume that the attacker can access the victim's device (e.g., by stealing it) but cannot compromise its TEE. We do not trust Tabellion's application running in the user's device. We assume that the enclave in Tabellion's server is uncompromised. However, we do not trust the rest of the server components. We also assume that the attacker cannot leverage side channels to perform side-channel attacks on our TCB in the client TEE [59] and SGX enclave [43, 79].

We assume a secure NTP server, such as [14], with which clients can synchronize their clocks (§5.2). We trust the hardware of mobile devices, e.g., the camera, and the SGX feature of processors in the server. A self-evident contract includes certificates from the mobile device vendor (e.g., Samsung) and the enclave vendor (e.g., Intel). We trust these vendors.

Tabellion's prototype does not currently provide availability or confidentiality guarantees. Lack of the availability guarantee means that Tabellion's services, e.g., the registration service, may not to be available to users or that a contract signed with Tabellion may be lost. Lack of the confidentiality guarantee means that an attacker can access the content of a contract. These guarantees can be provided using existing solutions, e.g., encryption.

8.2 Security Analysis

We next analyze various attacks introduced in §3 and discuss whether they would succeed or fail against Tabellion.

Repudiation attack. We introduced three forms of this attack. In the first form, the attacker denies the signature. This would fail against Tabellion as the contract provides the photo of the user, signed with a key, which is authenticated with the user's biometrics and which also is used to confirm the contract pages. Moreover, the validity of the key can be verified by inspecting its certificate and the certificate of the mobile device. In the second form, the attacker denies mutual assent. This would fail as the screenshots confirmed by the user clearly show the contract and negotiation terms. Moreover, all screenshots are securely timestamped, which provides strong evidence of the order of actions. In the third form, the attacker denies that there was an opportunity to read the contract. This would fail as the contract includes signed screenshots of all the content viewed and confirmed by the user.

In addition, in either of these attack forms, the attacker may deny the strong evidence by Tabellion and claim their device or Tabellion's server was compromised. Tabellion's small TCB in its client and server provides strong protection against such claims. Moreover, the self-evident contract provides strong evidence that the expected code executed in the device TEE and enclave by providing their code measurements and certificates.

Impersonation attack. We introduced two forms of this attack. In the first form, the attacker must spoof the victim's authentication on the victim's device. This is challenging in Tabellion as it requires defeating TEE-protected biometric sensors with sophisticated anti-spoofing (e.g., Apple's Touch ID [35]).

An attacker in close proximity to the victim may attempt to defeat a fingerprint-based authentication by pressing the victim's fingers against the fingerprint scanner. While this is a difficult attack already, we note that it is feasible only if fingerprint is used as the sole biometric signal. Tabellion's design is conducive to using different or multiple biometric signals, e.g., Apple's Face ID [35]. Note that these modern biometric sensors are accessible in the mobile device TEE [25, 30] and hence using them does not require adding more trusted code.

For the second form (where the attacker tries to spoof the victim's identity), we see five attack variants on Tabellion. We first briefly introduce these variants and then describe how Tabellion protects against them. The first variant is using an *existing or deep-faked*

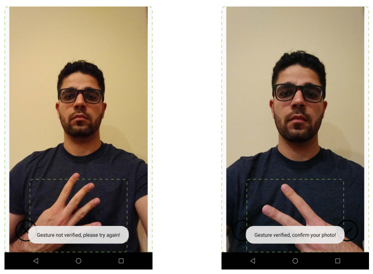

Gesture not verified, please try again!
Gesture verified, confirm your photo!

Figure 5: *Custom gesture (V sign) required in Tabellion's photos. (Left) The user performs an incorrect gesture. (Right) The user performs the correct gesture. In both cases, the user is notified accordingly. In (Left), the notification says "Gesture not verified, please try again!" In (Right), it says "Gesture verified, confirm your photo!"*

photo of the victim. The second one is taking *a photo of an unaware victim.* The third one is taking *a photo of a 3D-printed object looking like the victim.* The fourth one is taking *a photo of an existing or deep-faked photo of the victim shown on a display.* The fifth one is taking *a photo of a doppelganger.*

Tabellion defeats the first attack variant by its use of a secure photo, which guarantees that the photo is captured by the camera hardware. The second variant is challenging for the attacker as it requires physical proximity to the victim. Yet, Tabellion further defeats this attack by mandating a requirement for the photos used for registration: *specialized photo.* More specifically, Tabellion requires the user to perform a custom gesture while taking the secure photo in order to demonstrate *awareness.* The contract is not valid if the requirement is not satisfied and Tabellion's server automatically checks the requirement using an open source framework [21]. Figure 5 shows this solution in practice.

The third and fourth variants are also difficult for the attacker. Yet, Tabellion can make these attacks even harder by detecting *liveness*, using one of the several existing solutions [12, 35, 57, 58, 78]. For example, it can require the user to take multiple photos from different angles of their face, which can be used to detect liveness [12]. Alternatively, it can use an iris scanner or secure face recognition on modern phones, e.g., Apple's Face ID, which projects a grid of 30,000 infrared dots on a user's face and takes an infrared picture [35].

The fifth variant is also challenging for the attacker since it requires a doppelganger. Yet, Tabellion can defend against this attack by asking the user to show an official ID in the photo, which can be automatically checked using services such as Checkr [13].

Confusion attack. In this attack, a malicious offeror leverages vulnerabilities in the contract viewer in the offeree's device to mislead the offeree. Tabellion's rendering of the contract in its server neutralizes the Dalì attack (which is a specialized form of the confusion attack, discussed in §3) since the attacker cannot directly send a file to the offeree's device. Moreover, Tabellion's use of TEE to securely display contract pages defeats a powerful attacker who may be able to take control of the contract viewer on the victim's device (e.g., by compromising Tabellion's app or the OS) and try to change the contract content shown to and confirmed by the victim.

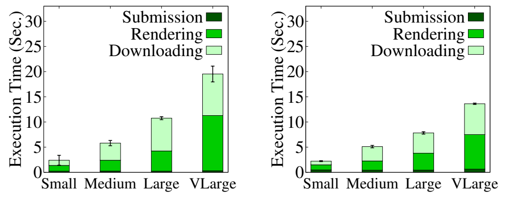


Figure 6: *Tabellion's execution time of offer (as defined) for (Left) HiKey and (Right) Nexus 5X.*

8.3 Case Analysis

In §2, we discussed a few legal cases where existing electronic contract platforms have failed to provide strong evidence. While it is difficult, if not impossible, to predict the legal validity of a contract with certainty, we believe Tabellion, if used for forming those contracts, would have provided stronger evidence. First, in *In re Mayfield* [6], the court discussed the ease with which the signature on DocuSign could be forged. It mentioned that "*what happens when a debtor denies signing a document and claims his spouse, child, or roommate had access to his computer and could have clicked on the 'Sign Here' button.*" In contrast, Tabellion provides a secure photo of the signatory and protects against impersonation attempts, as discussed.

Second, in *Adams v. Quicksilver, Inc.* [4, 10], the problem was that "*it couldn't be determined when the agreement was signed.*" Tabellion's use of secure timestamps for every action in the signing process provides strong evidence for when each party signed the contract.

Finally, in *Labajo v. Best Buy Stores* [3, 62], the problem was that "*Best Buy couldn't prove that she [Christina] saw and approved the disclosure.*" Tabellion's use of secure screenshots to capture all the content seen by a user provides strong evidence for this requirement.

9 EVALUATION

9.1 Performance Evaluation

We present the execution time of using Tabellion (measured on the client device). We include results for our main prototype on the HiKey board and our prototype on a Nexus 5X smartphone.

Figure 6 shows the *execution time of offer*, defined and measured from when the offeror submits a contract to Tabellion until when the contract is ready for them to sign. This includes the time needed to send a request to the server, render the contract pages in the server, and download them to the device. The figure shows the results for four contracts of different lengths. These contracts, labeled as small, medium, large, and very large, result in 6, 12, 24, and 48 pages for HiKey and 4, 8, 16, and 32 pages for Nexus 5X (the numbers are different in the two platforms since they have different screen sizes). As can be seen, even for very large contracts, the overall time is less than 20 seconds.

The same figure also shows the breakdown of the execution time. It shows it is mostly due to transfer of contract pages from the server to the device and due to rendering of the contract. Our rendering pipeline can be improved, as it currently renders the contract in several stages (Markdown to HTML, HTML to PDF, and finally PDF to PNG).

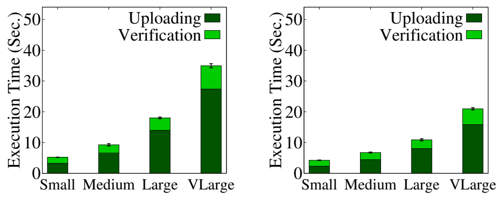


Figure 7: ***Tabellion's execution time of acceptance (as defined) for (Left) HiKey and (Right) Nexus 5X.***

Figure 7 shows the *execution time of acceptance*, defined and measured from the time that the offeree submits their signed contract to Tabellion until when the contract is notarized and verified (excluding the last inquiry to the offeror, which might take very little time or an arbitrarily long time depending on the reachability of the offeror, as discussed in §5.4). The results show that the execution time is around 35 seconds for the very large contract. As the results show, most of the execution time is due to uploading the signed screenshots to the server.

We next measure the execution time of secure primitives on the HiKey board (Primitives I-III) and on the server (Primitive IV). More specifically, we measure the execution time of a single use of a secure primitive. For each primitive, we measure the execution time several times and report the average and standard deviation. Figure 8 shows the results. It shows the execution times of these primitives are small. Note that Primitive III enforces at least a two second display freeze (§5.3).

9.2 Energy Measurement

We measure the energy consumption of Tabellion in our secondary prototype with a Nexus 6P smartphone. We perform this experiment just to demonstrate that Tabellion does not drain the battery, which would cause inconvenience to the user. We measure the energy on Nexus 6P as opposed to Nexus 5X, which we use in other experiments. This is because the fuel gauge in Nexus 6P, unlike Nexus 5X, includes a charge counter [26]. We measure the energy for the offeror submitting and signing a very large contract (28 images on Nexus 6P). Our results show that the overall energy consumption is on average 68.97 (standard deviation of 4.67) milliwatt-hours, which is 0.5% of the overall battery capacity on this smartphone assuming a 3.7 V voltage.

9.3 User Study

We perform an IRB-approved user study[5] to evaluate four usability aspects of Tabellion: effectiveness of presentation, usage convenience, readability of contracts, and time spent on contracts. Note that we do not evaluate other usability aspects of the system, such as registration.

We recruit 30 participants in our study (22 undergraduate and 8 graduate computer science students; 21 male and 9 female). We ask the participants to read and sign contracts on a Nexus 5X smartphone. After each contract, we ask them to answer two multiple-choice questions about the contract's details and ask them to rate the convenience and readability of the contract platform. Example questions are true/false questions about a specific statement

[5]UC Irvine IRB HS# 2019-5017

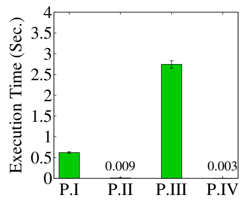


Figure 8: ***Execution time of secure primitives.***

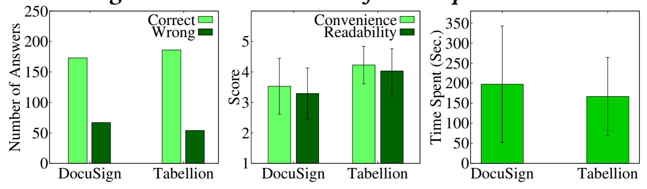


Figure 9: ***(Left) Correct/wrong answers for questions in the user study. (Middle) Convenience and readability of platforms rated by participants. (Right) Average time spent on the contracts in the user study.***

in the contract or a question asking about a specific value, e.g., the total cost of a service. Each participant uses two different platforms, Tabellion and DocuSign [18], and signs four contracts on each. We choose the contracts out of a repository of eleven contracts and rotate the contracts among platforms and participants to avoid any systemic bias (i.e., each contract is signed by several users on each platform). These contracts include sale, loan, Non-Disclosure-Agreement (NDA), and rental contracts that we created using contract samples from Docsketch [15].

Effectiveness of presentation. To measure the effectiveness of presentation and user understanding, we count correct and wrong answers to the questions for the contracts. Figure 9 (Left) shows the results. It shows that the participants fared slightly better in Tabellion, demonstrating that Tabellion's rendering of the contract specifically for a mobile device and the use of good formatting guidelines (§6.1) are effective.

Convenience and readability. Using a Likert Scale, we ask the participants to rate the convenience and readability of the platforms. Figure 9 (Middle) shows that the participants slightly preferred the convenience and readability of Tabellion over DocuSign for signing the contracts. More specifically, on average, the participants scored the convenience and readability of the contracts to be 4.22 and 4.02 for Tabellion and 3.53 and 3.28 for DocuSign.

Time spent on contracts. Figure 9 (Right) shows that participants spent slightly less time to view and sign the contracts on Tabellion (166.68 seconds on Tabellion and 197.46 on DocuSign, on average).

All of these results show that the security benefits of Tabellion do not come at the cost of usability. However, one might wonder whether Tabellion's usability decreases for long contracts. We report the results for the longest contract in our study, which was an NDA contract, consisting of 7 PDF pages for DocuSign and 25 pages for Tabellion. Our results show that participants scored the convenience and readability of the NDA contract, respectively, to be 4 and 3.6 for Tabellion and 2.6 and 3.1 for DocuSign. Moreover, they spent 355 seconds for this contract on Tabellion and 490 seconds on DocuSign. This shows that the usability of Tabellion is good even for long contracts.

10 RELATED WORK

Attacks on Electronic Signatures. Several papers study different forms of attacks on electronic signatures [49, 54]. Dalì attack [40, 41, 70] is a confusion attack enabled by the same file being interpreted differently with different file extensions (§3). Other attacks use malicious fonts [52] and JavaScript code [53] as the source of the dynamic representation. Tabellion protects against these attacks as discussed in §8.2.

TEE for Protected UI. Several systems take advantage of ARM TrustZone and virtualization hardware in mobile devices to protect the UI. VButton [56] provides a framework for attesting the user operations in different apps. TruZ-View [81] provides UI integrity and confidentiality protections without porting the UI renderer to the TEE. AdAttester [55] provides attestation for the user's clicks on ads in Android apps. SchrodinText [36] protects the confidentiality of select textual content on the UI. Yu et al. [82] and Zhou et al. [85] design trusted path solutions for I/O devices, such as display, GPU, and keyboard, to give an application direct and secure access to these devices. In contrast to all of these systems, Tabellion is concerned with legal contracts and hence provides novel secure primitives to help generate strong evidence for the legal and valid formation of contracts.

TEE for System Security. Several systems use the TEE for security purposes. DelegaTEE [61] uses TrustZone and SGX to provide delegation of credentials. TruZ-Droid [80] enhances the functionality of TrustZone TEE by providing a binding between an Android app and a Trusted Application (TA) so that the Android app callback functions can be triggered from a TA. TrustFA [83] designed a remote facial authentication method that, unlike Tabellion, moves the camera and display driver into the TEE, which increases the TCB size. Trusted sensors [47, 60] attest sensor data to applications. Gilbert et al. [47, 48] attest the integrity of sensor data. fTPM [71] implements the TPM functionality within TrustZone. Samsung Pay [31] authenticates users via fingerprints in TEE for confirming a transaction. Android Protected Confirmation [8] provides APIs for applications, such as a banking app, to get the user's confirmation on certain important messages, such as those used for transferring money. SecTEE [84] and SANCTUARY [39] use TrustZone to provide enclave for applications running in ARM SoC-based devices. Several works also use SGX enclave to provide different security guarantees [38, 51]. None of these systems provide a platform for legal contracts and hence do not address its challenges.

UI Protection. AdSplit [74] and AdDroid [69] isolate applications from ad services, while maintaining a uniform UI. LayerCake [72] enables UI components to be securely embedded in Android. They do not, however, protect against attacks on legal contracts.

Untrusted Operating System. Overshadow [44] and InkTag [50] protect an application from an untrusted operating system using a hypervisor. Ditio [66] makes the OS untrusted using a hypervisor- and TrustZone-based TEE when auditing sensor usage in a device. Viola [63, 65] uses the hypervisor to protect against unauthorized accesses to memory-mapped sensor pages. Similar to Tabellion, these systems leverage the TEE to make the OS untrusted. However, they do not address the challenges of securing legal contracts.

eNotary. There are eNotary [19] services whereby a user can get a legal signature notarized electronically. These services, however, are not pure electronic solutions because they require a human notary to be present. Indeed, in the case of DocuSign's solution [17], the notary needs to be in the same physical location as the signatory. We note that requiring a notary can help provide strong evidence for the contract, but it is expensive and time consuming.

Smart Contracts. Smart contracts are self-executing computer programs on Blockchains that enforce terms of an agreement between parties automatically [68]. For example, a smart contract in a loan agreement can perform automatic payments via the Internet at pre-specified dates. However, current smart contract solutions do not provide the strong security guarantees that Tabellion does, namely providing strong evidence for the legal validity of contracts. Addressing these issues require carefully designing and securing the interactions of the user with the contract platform. We believe that several techniques provided in Tabellion are complementary to smart contracts.

Secure NTP. Solutions such as NTS [7] and NTPsec [27] improve the security of time synchronization by verifying the authenticity of the time server and protecting the integrity of the of the synchronization packets. NTS suggests multiple approaches for mitigating the asymmetric delay attack such as using multiple servers, defining a time threshold on the client device, or using multiple communication paths. However, these solutions do not fully solve the problem, need extra hardware, and are difficult to deploy within a TEE. Tabellion, on the other hand, calculates a confidence interval without requiring extra hardware and is deployable in a TEE.

Delay-resistant Systems. TimeSeal [37] designs a secure timer for SGX. It uses counting threads to improve the resolution of SGX timer and addresses the scheduling attacks to the counting threads. It also addresses the delay attacks between the application enclave and the Platform Service Enclave (PSE) that provides the trusted timer in SGX. However, it does not address the NTP asymmetric delay attack. cTPM [42] provides secure time for TPM. It uses a trusted cloud server to update the time on the local TPM device. It addresses the delay attack on NTP by using a global timeout value. However, this technique needs to re-do synchronization if the delay is more than the timeout value. Sandha et al. [73] evaluate accuracy of time synchronization on smartphones using different hardware solutions but do not discuss attacks on synchronization.

11 CONCLUSIONS

We presented Tabellion, a system solution for secure legal contracts on mobile devices. Tabellion generates strong evidence for the valid and legal formation of contracts and provides self-evident contracts. It uses four secure primitives implemented with only ~1,000 LoC. It also provides a fully functional contract platform, built with ~14,000 lines of untrusted code. Through prototype measurements, analysis, and a user study, we showed that Tabellion is secure, achieves acceptable performance, and provides slightly better usability than DocuSign for viewing and signing contracts.

ACKNOWLEDGMENTS

The work was in part supported by NSF Awards #1617513 and #1718923. The authors thank the paper shepherd, Jeremy Andrus, and the anonymous reviewers for their insightful comments.

REFERENCES

[1] 2000. ELECTRONIC SIGNATURES IN GLOBAL AND NATIONAL COMMERCE ACT. PUBLIC LAW 106–229.
[2] 2004. NIST Study Shows Computerized Fingerprint Matching Is Highly Accurate. https://www.nist.gov/news-events/news/2004/07/nist-study-shows-computerized-fingerprint-matching-highly-accurate.
[3] 2007. *Labajo v. Best Buy Stores, LP*, 478 F. Supp. 2d 523 (S.D.N.Y. 2007).
[4] 2010. *Adams v. Quicksilver, Inc.* no. G042012 (Cal. App. 4th Div. Feb. 22, 2010).
[5] 2015. *O'Connor v. Uber Technologies, Inc.*, 150 F.Supp.3d 1095 (N.D. Cal. 2015).
[6] 2016. *In re Mayfield*: No. 16-22134-D-7, 2016 WL 3958982 (E.D. Cal. July 13, 2016). https://www.govinfo.gov/content/pkg/USCOURTS-caeb-2_16-bk-22134/pdf/USCOURTS-caeb-2_16-bk-22134-0.pdf.
[7] 2017. Network Time Security. https://tools.ietf.org/html/draft-ietf-ntp-network-time-security-15.
[8] 2018. Android Protected Confirmation. https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html.
[9] 2018. Global +$4 Billion Digital Signature Market by Deployment, Component, Industry and Region - Forecast to 2023 - ResearchAndMarkets.com. https://www.businesswire.com/news/home/20181001005761/en/Global-4-Billion-Digital-Signature-Market-Deployment.
[10] 2019. 7 landmark electronic signature legal cases. https://esignrecords.org/7-landmark-electronic-signature-legal-cases/.
[11] 2019. AdobeSign. https://acrobat.adobe.com/us/en/sign.html.
[12] 2019. BioID Liveness Detection. https://www.bioid.com/liveness-detection/.
[13] 2019. Checkr. https://checkr.com/product/screenings/.
[14] 2019. Cloudfare Secure Time Service. https://developers.cloudflare.com/time-services/nts/usage/.
[15] 2019. Contract Templates and Agreements. https://www.docsketch.com/contracts/.
[16] 2019. Device-side Security: Samsung Pay, TrustZone, and the TEE. https://developer.samsung.com/tech-insights/pay/device-side-security.
[17] 2019. DocuSign eNotary. https://www.docusign.com/products/enotary.
[18] 2019. DocuSign Website. https://www.docusign.com/.
[19] 2019. eNotary. https://en.wikipedia.org/wiki/ENotary.
[20] 2019. eSignLive. https://www.esignlive.com/.
[21] 2019. Gesture Recognition. https://github.com/Gogul09/gesture-recognition.
[22] 2019. Global Digital Signature Market to Reach $3.44 Billion by 2022 at 30.0% CAGR: Says AMR. https://www.globenewswire.com/news-release/2019/08/13/1901155/0/en/Global-Digital-Signature-Market-to-Reach-3-44-Billion-by-2022-at-30-0-CAGR-Says-AMR.html.
[23] 2019. HelloSign. https://www.hellosign.com.
[24] 2019. Intel Provisioning Certification Service for ECDSA Attestation. https://api.portal.trustedservices.intel.com/provisioning-certification.
[25] 2019. iOS Security – iOS 12.3. https://www.apple.com/business/docs/site/iOS_Security_Guide.pdf.
[26] 2019. Measuring Device Power. https://source.android.com/devices/tech/power/device.
[27] 2019. NTPsec. https://ntpsec.org/.
[28] 2019. PandaDoc. https://www.pandadoc.com/.
[29] 2019. Qualcomm's larger in-screen fingerprint sensor could seriously improve security. https://www.engadget.com/2019/12/03/qualcomm-3d-sonic-max-worlds-largest-in-display-fingerprint-sensor-specs-availability/.
[30] 2019. Samsung Knox Security Solution. https://images.samsung.com/is/content/samsung/p5/global/business/mobile/SamsungKnoxSecuritySolution.pdf.
[31] 2019. Samsung Pay. https://www.samsung.com/us/samsung-pay/.
[32] 2019. SignEasy. https://signeasy.com/.
[33] 2019. SignNow. https://www.signnow.com/.
[34] 2019. Votz. https://voatz.com/.
[35] 2020. Apple Platform Security, Spring 2020. https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf.
[36] A. Amiri Sani. 2017. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *Proc. ACM MobiSys*.
[37] F. M. Anwar. 2019. *Quality of Time: A New Perspective in Designing Cyber-Physical Systems.* Ph.D. Dissertation. UCLA.
[38] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, et al. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proc. USENIX OSDI*.
[39] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf. 2019. SANCTUARY: ARMing TrustZone with User-space Enclaves.. In *NDSS*.
[40] F. Buccafurri, G. Caminiti, and G. Lax. 2008. The Dalì Attack on Digital Signature. *Journal of Information Assurance and Security* (2008).
[41] F. Buccafurri, G. Caminiti, and G. Lax. 2009. Fortifying the Dalì Attack on Digital Signature. In *Proc. ACM Int. Conf. on Security of Information and Networks (SIN)*.
[42] C. Chen, H. Raj, S. Saroiu, and A. Wolman. 2014. cTPM: A cloud TPM for Cross-Device Trusted Applications. In *Proc. USENIX NSDI*.
[43] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.
[44] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. 2008. Overshadow: a Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proc. ACM ASPLOS*.
[45] M. A. Chirelstein. 2013. *Concepts and Case Analysis in the Law of Contracts, Seventh Edition.* Foundation Press.
[46] DocuSign. 2012. Going Mobile with Electronic Signatures. https://www.docusign.com/sites/default/files/Going_Mobile_with_Electronic_Signatures.pdf.
[47] P. Gilbert, L. P. Cox, J. Jung, and D. Wetherall. 2010. Toward Trustworthy Mobile Sensing. In *Proc. ACM Workshop on Mobile Computing Systems & Applications (HotMobile)*.
[48] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharkey, A. Sheth, and L. P. Cox. 2011. YouProve: Authenticity and Fidelity in Mobile Sensing. In *Proc. ACM SenSys*.
[49] Hernandez-Ardieta, J. L. and Gonzalez-Tablas, A. I. and de Fuentes, J. M. and Ramos, B. 2013. A taxonomy and survey of attacks on digital signatures. *Elsevier Computers & Security* (2013).
[50] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. 2013. InkTag: Secure Applications on an Untrusted Operating System. In *Proc. ACM ASPLOS*.
[51] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. 2018. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)* (2018).
[52] A. Jøsang, D. Povey, and A. Ho. 2002. What You See is Not Always What You Sign. In *Proc. AUUG*.
[53] K. Kain. 2003. Electronic Documents and Digital Signatures. *Master of Science Thesis, Dartmouth Computer Science Department, Technical Report TR2003-457* (2003).
[54] G. Lax, F. Buccafurri, and G. Caminiti. 2015. Digital Document Signing: Vulnerabilities and Solutions. *Information Security Journal: A Global Perspective* (2015).
[55] W. Li, H. Li, H. Chen, and Y. Xia. 2015. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *Proc. ACM MobiSys*.
[56] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan. 2018. VButton: Practical Attestation of User-driven Operations in Mobile Apps. In *Proc. ACM MobiSys*.
[57] Y. Li, Y. Li, Q. Yan, H. Kong, and R. H. Deng. 2015. Seeing Your Face Is Not Enough: An Inertial Sensor-Based Liveness Detection for Face Authentication. In *Proc. ACM CCS*.
[58] Y. Li, Z. Wang, Y. Li, R. Deng, B. Chen, W. Meng, and H. Li. 2019. A Closer Look Tells More: A Facial Distortion Based Liveness Detection for Face Authentication. In *Proc. ACM ASIA Conference on Computer and Communications Security (ASIACCS)*.
[59] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *Proc. USENIX Security Symposium*.
[60] H. Liu, S. Saroiu, A. Wolman, and H. Raj. 2012. Software Abstractions for Trusted Sensors. In *Proc. ACM MobiSys*.
[61] S. Matetic, M. Schneider, A. Miller, A. Juels, and S. Capkun. 2018. DelegaTEE: Brokered Delegation Using Trusted Execution Environments. In *Proc. USENIX Security*.
[62] E. Maxie. 2013. COURT CASE: LAWSUIT FILED OVER POORLY CONCEIVED ELECTRONIC SIGNATURE. https://www.signix.com/blog/bid/93126/court-case-lawsuit-filed-over-poorly-conceived-electronic-signature.
[63] S. Mirzamohammadi and A. Amiri Sani. 2016. Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems. In *Proc. ACM MobiSys*.
[64] S. Mirzamohammadi and A. Amiri Sani. 2018. The Case for a Virtualization-Based Trusted Execution Environment in Mobile Devices. In *Proc. ACM Asia-Pacific Workshop on Systems (APSys)*.
[65] S. Mirzamohammadi and A. Amiri Sani. 2018. Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems. *IEEE Transactions on Mobile Computing (TMC)* (2018).
[66] S. Mirzamohammadi, J. A. Chen, A. Amiri Sani, S. Mehrotra, and G. Tsudik. 2017. Ditio: Trustworthy Auditing of Sensor Activities in Mobile & IoT Devices. In *Proc. ACM SenSys*.
[67] J. Mulliner. 2018. What the Wells Fargo Mobile Research Reveals About E-Signatures. https://www.onespan.com/blog/what-the-wells-fargo-mobile-research-reveals-about-e-signatures.
[68] R. O'Shields. 2017. Smart Contracts: Legal Agreements for the Blockchain. *NC Banking Inst.* (2017).
[69] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. 2012. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.
[70] D. Popescu. 2012. Hiding Malicious Content in PDF Documents. *arXiv preprint arXiv:1201.0397* (2012).
[71] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten. 2016. fTPM: A Software-Only Implementation of a TPM Chip. In *25th USENIX Security Symposium (USENIX Security 16), Austin, TX*.
[72] F. Roesner and T. Kohno. 2013. Securing Embedded User Interfaces: Android and Beyond. In *Proc. USENIX Security Symposium*.

[73] S. S. Sandha, J. Noor, F. M. Anwar, and M. Srivastava. 2019. Exploiting Smartphone Peripherals for Precise Time Synchronization. In *Proc. IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS).*

[74] S. Shekhar, M. Dietz, and D. S. Wallach. 2012. AdSplit: Separating Smartphone Advertising from Applications. In *Proc. USENIX Security Symposium.*

[75] Dilani Silva. 2019. Demand for E-Signing From Mobile Devices on the Rise in Financial Institutions. https://www.onespan.com/blog/demand-for-e-signing-from-mobile-devices-on-the-rise-in-financial-institutions.

[76] M. Simpson. 2018. BDC app offers e-signature for loans, reducing in-person visits. https://www.itbusiness.ca/news/bdc-app-offers-e-signature-for-loans-reducing-in-person-visits/104429.

[77] M. H. Stanzione. 2020. 'Wet' Ink Signatures Requirements May Fade After Coronavirus. Bloomberg Law, The United States Law Week.

[78] D. Tang, Z. Zhou, Y. Zhang, and K. Zhang. 2018. Face Flashing: a Secure Liveness Detection Protocol based on Light Reflections. *arXiv preprint arXiv:1801.01949v2* (2018).

[79] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, Baris Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proc. USENIX Security Symposium.*

[80] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du. 2018. TruZ-Droid: Integrating TrustZone with Mobile Operating System. In *Proc. ACM MobiSys.*

[81] K. Ying, P. Thavai, and W. Du. 2019. TruZ-View: Developing TrustZone User Interface for Mobile OS Using Delegation Integration Model. In *Proc. ACM CODASPY.*

[82] M. Yu, V. D. Gligor, and Z. Zhou. 2015. Trusted Display on Untrusted Commodity Platforms. In *Proc. ACM CCS.*

[83] Dongli Zhang. 2014. TrustFA: TrustZone-Assisted Facial Authentication on Smartphone. *Technical Report* (2014).

[84] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng. 2019. SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE. In *Proc. ACM CCS.*

[85] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. 2012. Building Verifiable Trusted Path on Commodity x86 Computers. In *Proc. IEEE Symposium on Security and Privacy (S&P).*