Presence Attestation: The Missing Link in Dynamic Trust Bootstrapping

Zhangkai Zhang*
Beihang University
zhangzhangkai315@gmail.com

Xuhua Ding
Singapore Management University
xhding@smu.edu.sg

Gene Tsudik
University of California, Irvine
gts@ics.uci.edu

Jinhua Cui
Singapore Management University
jhcui@smu.edu.sg

Zhoujun Li
Beihang University
lizj@buaa.edu.cn

ABSTRACT

Many popular modern processors include an important hardware security feature in the form of a DRTM (Dynamic Root of Trust for Measurement) that helps bootstrap trust and resists software attacks. However, despite substantial body of prior research on trust establishment, security of DRTM was treated without involvement of the human user, who represents a vital missing link. The basic challenge is: how can a human user determine whether an expected DRTM is currently active on her device?

In this paper, we define the notion of "presence attestation", which is based on mandatory, though minimal, user participation. We present three concrete presence attestation schemes: sight-based, location-based and scene-based. They vary in terms of security and usability features, and are suitable for different application contexts. After analyzing their security, we assess their usability and performance based on prototype implementations.

KEYWORDS

trusted computing, attestation, dynamic root of trust, human-in-the-loop, device I/O

1 INTRODUCTION

Many currently popular x86 and ARM processors are equipped with a special hardware feature, called *Dynamic Root of Trust for Measurement* (DRTM), e.g., Intel TXT [12], AMD SVM [2], and ARM TrustZone[1] [3]. DRTM is designed to withstand software attacks, even from the operating system level. When activated at runtime, it securely measures and launches some software which may further measure and load another layer of software. Such iterations of measure-then-launch form a trust chain rooted in DRTM allowing a remote entity (verifier) to establish trust in the system, after checking integrity of the latter's software stack.

Popularity of DRTM has also fueled some new directions in system security research. Several designs [4, 5, 19, 32, 33] have been proposed to cope with OS-level threats directly by using various DRTM instantiations to ensure security of the Trusted Computing Base (TCB).

However, most prior efforts either overlooked or side-stepped an important factor – *the human user*. As noted by Parno et al. [23], it is challenging for a human user to "bootstrap trust" in DRTM itself, since she is not assured that the chain of trust is indeed rooted in **her** DRTM. The main reason is the difficulty for a human user to establish an authenticated and secure channel with her own DRTM. Although DRTM is trusted, the user cannot determine whether her DRTM is indeed engaged. In particular, malware can impersonate the user's DRTM using a so-called "cuckoo attack" [23].

Two mitigation approaches are proposed and discussed in [23]. The first relies on a hardware-based secure channel, e.g., a special-purpose I/O interface through which an external verification device directly interacts with DRTM. The second establishes a cryptographically secured communication channel and requires the user to have prior knowledge of the public key identifier of her DRTM.

The former offers stronger security and better usability. For example, an LED light securely wired to TrustZone can confirm to the user that her DRTM is active. Most COTS x86 and ARM devices are not equipped with such a feature and implementing it would require cooperation of hardware manufacturers. The latter approach also requires manufacturers' cooperation (though to a lesser extent) in order to export the DRTM's public key identifier to the user via either: (1) an out-of-band channel, e.g., etched or printed on the exterior of the device itself, or (2) a special protected interface. In any case, dealing with DRTM's public key identifier represents added burden for the user.

We observe that a user's trust in her device is based not only on physical availability of DRTM. Rather, to a greater extent, it is based on availability and security of *software* directly measured and loaded by DRTM, which constitutes the TCB of the user's device. For ease of presentation, we refer to the initial software in the trust chain following DRTM as the *trust anchor*.

In this paper, we use the divide-and-conquer approach to bootstrapping user's trust in her own device, without imposing aforementioned burden on manufacturers and users:

*The work was mainly done when the author visited SMU as a student intern.
[1]We slightly expand the notion of DRTM as TrustZone functions differently from the TXT and SVM.

CCS'17, , Oct. 30–Nov. 3, 2017, Dallas, TX, USA.

ACM ISBN ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134094

(1) The user checks whether a genuine DRTM has launched the trust anchor and is currently interacting with her.[2]
(2) The user verifies whether the trust anchor launched in the preceding step actually resides on her own device.

The second step is cryptographically bound to the first by an ephemeral secret key. Also, the implicit logical link between the two steps is based on trust anchor's runtime security being ensured by DRTM. We argue that this is a widely adopted trust assumption in many DRTM-based secure systems. If DRTM cannot ensure runtime security of the trust anchor, i.e., the very first software in the chain, its measurements are of little value since they only momentarily reflect static software integrity[3].

Based on the general approach outlined above, we design and implement three *presence attestation* schemes. By taking advantage of hardware DRTM's security assurance and software capability of the trust anchor, these schemes allow a human user to establish trust in her device after confirming that DRTM and the trust anchor are active. Proposed schemes are based on different physical properties: sight, scene and location. The sight-based scheme achieves strongest security since it resists analog cuckoo attacks, while the other two offer better usability commensurate with slightly weaker security. We implement all three protocols and assess their security and performance.

Organization. The next section describes the system setting as well as the adversary model, and overviews the proposed approach. Section 3 details the sight-based attestation scheme, followed by location-based and scene-based variants in Section 4. Implementation details are presented in Section 5 and experimental results are reported in Section 6. Section 7 discusses related work and Section 8 concludes the paper.

2 SYNOPSIS

This section sets the stage for the rest of the paper by describing the assumed environment, (including the devices and the adversary) and overviewing the notion of presence attestation.

2.1 System Model & Problem Definition

We consider the following system model widely used in the literature. A human user (Alice) physically controls a computing device Dev equipped with a hardware DRTM denoted as ROT_D instantiation of which is dependent on Dev's platform architecture. For example, if Dev is a smartphone with an ARM processor, ROT_D is the processor's TrustZone component, including code running in it. At runtime, ROT_D loads a trust anchor denoted by TA. For example, TA could be a bare-metal micro-hypervisor launched on Dev in order to protect a security-sensitive application. Figure 1 shows the chain of the actions leading to application-level security by tapping into the strong security assurance provided by the built-in DRTM.

The chain of actions is also the chain of trust propagation. Trust is bootstrapped from ROT_D, followed by TA and then, the secured application, in the sense that another device can verify the trust chain and establish trust in Dev. Unfortunately, Alice cannot establish trust on any of them. Note that the system is *not* trusted

[2]The main goal here is to ascertain whether **any** DRTM is involved in the interaction, not necessarily the one on the user's device.
[3]For the same reason DRTM is not used in the literature to directly launch the OS, which is vulnerable to runtime attacks.

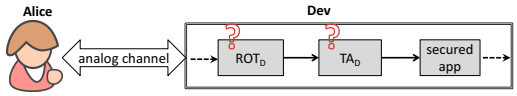


Fig. 1: Trust bootstrapping expected by Alice: TA **corresponds to the first software component in the trust chain rooted at** ROT_D.

prior to the launch of ROT_D. Malware residing on Dev can pretend to launch ROT_D and produce user-perceptible effects (e.g., audio, visual, etc.) identical to the real ROT_D. In other words, Alice cannot reliably determine whether trust is indeed bootstrapped in Dev.

The problem at hand is how to help Alice to securely verify *whether* ROT_D *is currently active on Dev.* Since she physically controls Dev, Alice is certain that ROT_D is installed on Dev. Nonetheless, she cannot determine ROT_D's status at runtime, due to lack of an authentic and secure channel between herself and her DRTM. Even worse, current DRTM manufacturers do not offer a way for Alice to identify DRTM on her device. Alice may not even know anything about ROT_D's public key certificate issued by the manufacturer.

Assumptions. We assume that each DRTM is secure against software attacks; this is, in fact, one of the main design goals of DRTM. Albeit, actual DRTM products are not assumed to be free of vulnerabilities. We assume that there exists a public key infrastructure (PKI), with each DRTM assigned a unique public/private key-pair and a credential, i.e., a public key certificate (PKC). Each DRTM can produce cryptographic evidence (e.g., in the form of a signature) to authenticate itself and thus prove its genuineness, by using its private key which is securely stored along with a credential. Nonetheless, as mentioned earlier, Alice is not assumed to have any prior knowledge of ROT_D's public key.

We also assume that the trust anchor launched by the DRTM is secure against kernel-level attacks, since hardware places the trust anchor in a more privileged environment than the OS. Other software components that follow the trust anchor in the chain are not relevant to our work.

2.2 Portrait of The Adversary

The anticipated embodiment of the adversary is malware running on Dev with kernel privileges. This malware controls all software and hardware resources accessible to the OS (except the TCBs which DRTM depends on). We parameterize the adversary with two aspects: (1) collusion with external entities, and (2) capability of using analog devices.

Local vs. *Collusive.* A **local** adversary is represented by standalone malware running on Dev with no runtime collusion with other devices. Although it might have the ability to communicate, it is not assisted by any external entity.

A **collusive** adversary consists of malware resident on Dev and a remote accomplice residing on at least one other device denoted by M, equipped with its own DRTM denoted by ROT_M. (We assume that M is a genuine device, of the same type as Dev, that has not been physically attacked; specifically, ROT_M is inviolate.) The local and remote adversarial components interact over a network. They might be physically near each other and communicate directly, e.g.,

via Bluetooth. Alternatively, they can be far apart and communicate over a Wi-Fi or cellular interface. The adversary does not use a wired connection, since Alice can easily unplug all cables connected to Dev prior to attestation. Also, although Alice can, in principle, muffle or jam all Dev's wireless communications, doing so requires additional specialized equipment.

A collusive adversary can mount a cuckoo attack (see Figure 2) described by Parno, et al. [23] which defeats TPM-based attestation. It is a special form of the man-in-the-middle (MITM) attack, whereby Dev forwards the attestation challenge to an accomplice device which then produces a valid response. The cuckoo attack is based on verifier's inability to determine the exact hardware origin of the attestation response. This stems from the fact that the communication channel between the verifier and Dev is not authenticated. It is also not perceivable by the human user.

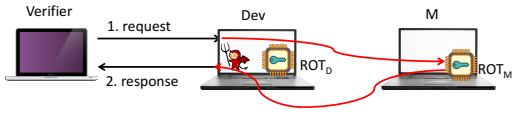


Fig. 2: Cuckoo attack example [23]. ROT$_D$ in Dev is not active. Malware in Dev uses ROT$_M$ in an accomplice device to produce a legitimate attestation response to the verifier which intends to assess Dev's trustworthiness.

Software-only vs. *Analog.* We also consider whether the adversary's reach extends into the physical (analog) world. A software-only adversary does not use anything beyond software, i.e., its attacks are performed entirely by malware. Such an adversary cannot eavesdrop on (or influence) analog signals emitted from any device.

An analog adversary exploits the physical environment and uses additional equipment to eavesdrop on, intercept, transmit, or modify analog data. Since Dev is physically controlled by Alice, an analog adversary must be a collusive one. The analog attack is conducted on the remote accomplice's side. A concrete example of an accomplice might be a screen that displays photos or videos transmitted by malware in Dev.

Caveat. We do not consider physical attacks that modify any hardware behavior or physically extract secrets (by brute force or via side-channels) from a DRTM. Such attacks fundamentally undermine DRTM security and are beyond the scope of this paper.

2.3 Overview of Presence Attestation

To bootstrap Alice's trust in her device Dev, we propose an attestation scheme to verify presence of an active ROT$_D$ on Dev. We assume that, during attestation, Alice is assisted by a trusted computer, denoted as Verifier.[4] Verifier engages Dev over a digital or analog channel to verify ROT$_D$'s presence. We believe that Verifier is a necessary component since Alice can neither securely communicate with ROT$_D$ nor perform necessary computations.

Strawman Approach: Before describing the proposed scheme, we consider an intuitive "strawman" approach and show how it fails, which highlights the subtlety of the problem at hand. The strawman approach is simple:

> **ROT$_D$ securely controls Dev's display and uses it to output its public key for Alice to validate**

This approach is insecure since Alice does not have a means to verify whether the displayed public key is indeed from ROT$_D$. Since Alice has no prior knowledge of this public key, the adversary can display an arbitrary value on Dev's screen without invoking ROT$_D$ and Alice can not visually authenticate its origin. The main challenge stems from Alice having no identifiable information with respect to ROT$_D$.

The proposed *presence attestation* scheme splits the problem into two parts: (1) *existence* problem, i.e., whether some DRTM interacts with Verifier, and (2) *residence* problem, i.e., whether the DRTM that participates in the interaction actually resides on Dev. The main benefit of this approach is the flexibility of taking advantage of DRTM security assurances and software trust anchor's capabilities. The scheme proceeds as follows:

Phase 1. Existence Checking. Verifier and Dev engage in an enhanced static integrity attestation protocol. Its outcome allows Verifier to determine whether it is interacting with a true DRTM which has launched a trust anchor. Nonetheless, it does not confirm that DRTM is indeed ROT$_D$ on Dev, due to the possible presence of the collusive adversary. At the end of the protocol, Verifier and the DRTM share a secret key k, needed in Phase 2 .

Existence checking determines whether Verifier is interacting with a genuine DRTM based on the latter's ability to generate a valid signature, i.e., verifiable using the public key from the DRTM's certificate. It extends conventional integrity attestation with verification of ownership of the private key and establishment of a fresh shared secret key.

Existence checking is based on the following logic: an entity that can, based on a challenge, produce a signed response verified using the public key (contained in a valid DRTM's certificate issued by a recognized *CA*) must know the corresponding private key. Recall that, according to our trust and adversary model, the adversary cannot extract the private key from **any** DRTM.

Interaction between Verifier and Dev is shown in Protocol 1. Because of the threat of cuckoo attacks, the response received by Verifier could be computed by ROT$_M$ instead of ROT$_D$, though Verifier exchanges data with Dev, from the *communication* perspective.

Caveat 1. Successful completion of Step 3 implies that Verifier shares a fresh random secret key k with ROT$_D$. In case of a cuckoo attack, ROT$_M$ obtains k; however, neither Dev nor (untrusted) software on M knows k.

Caveat 2. The certificate in Step 1 does not provide sufficient identification information for Verifier to authenticate Dev, since Alice does not have sufficient knowledge to link this certificate to Dev.

Phase 2. Residence Checking. The trust anchor attests to its physical environment and sends to Verifier its attestation token, integrity of which is protected by k. Using the analog interface provided by Verifier, Alice checks the physical environment vouched for by the trust anchor and determines whether it is indeed TA in Dev.

[4] In a typical setting, we expect that Dev would be Alice's smartphone or similar-class device and Verifier – Alice's laptop or desktop.

Existence Checking ()

1. Verifier verifies DRTM's public key certificate provided by Dev. If valid, it extracts public keys PK_e and PK_v.
2. Verifier$\xrightarrow{c}$ Dev. Verifier generates random number r, random secret key k, and computes $c = \mathbf{E}_{PK_e}(r||k)$.
3. Verifier$\xleftarrow{\sigma}$ Dev. $\mathsf{ROT_D}$ computes $r||k = \mathbf{D}_{SK_d}(c)$ and generates signature $\sigma = \mathbf{S}_{SK_s}(r||\mathsf{H}_k(TA))$ where TA is the code and data image of the loaded trust anchor.
4. Verifier$\xrightarrow{b}$ Dev Verifier verifies the signature by computing $b = \mathbf{V}_{PK_v}(\sigma, r||\mathsf{H}_k(TA))$ where $\mathsf{H}_k(TA)$ is HMAC of expected trust anchor image. If σ is successfully verified, $b = 1$ is returned to Dev. Both parties proceed to Phase 2. Otherwise, Verifier terminates attestation and displays an error message to Alice.

Protocol 1: Phase 1: Existence Checking; $\mathsf{ROT_D}$'s public and private key pairs for encryption and signing are: (PK_e, SK_d) and (PK_v, SK_s), respectively.

The main challenge in designing a residence checking protocol is that Verifier, as a computer, cannot *identify* its protocol peer. All commonly used identifiers – such as MAC or IP addresses as well as IMEI numbers – are subject to modification attacks. Although a hardware identifier can be unforgeable, it requires the manufacturer to explicitly convey it to Alice who must then keep it at all times. On the other hand, although Alice can easily identify Dev physically, she can neither engage in any protocol with Dev, nor hold its (trusted) logic identifier.

Our general idea is to challenge the alleged DRTM (and its trust anchor) to respond with attestation for a physical feature of its hosting device's physical environment. Verifier first checks response integrity to ensure that it indeed originates from the alleged trust anchor. Then, Alice assesses whether this matches the expected physical environment.

There are two prerequisites: First, the physical property in use should be unique to the environment and not reproducible by the adversary without detection. Therefore, some intuitive physical properties such as velocity and altitude are not ideal. Second, the physical property must be securely capturable by the trust anchor, despite the threat from the untrusted kernel on Dev. This requirement rules out certain TA choices, e.g., software launched by Intel SGX can not act as TA, since it lacks I/O capability.

In the following sections, we introduce three types of residence checking based on scene, sight and location, with different security and usability.

3 SIGHT-BASED RESIDENCE CHECKING

The residence checking protocol is designed to resist attacks by the *collusive and analog* adversary. This section focuses on protocol logic only. The operational model of analog I/O devices is simplified to consist of rendering (delivering) data from/to main memory at fixed-length intervals. The next section presents implementation details of complex low-level I/O operations.

3.1 Basic Protocol

After a successful existence attestation phase, Verifier is assured that a genuine DRTM and its TA took part in the interaction. The next step is to check whether they both reside on $\mathsf{ROT_D}$ or $\mathsf{ROT_M}$.

Using the general approach described in Section 2.3, we propose to use the line-of-sight between Verifier and Dev to verify DRTM residence. Before initiating attestation, Alice positions Dev's camera to face Verifier's screen. The line-of-sight channel between Verifier and Dev is then used to convey Verifier's attestation challenge to Dev. Our rationale is based on the physical property of line-of-sight: since the attestation challenge is sent and received via the analog channel, a man-in-the-middle adversary must perform the same type of analog I/O operations to relay the challenge. It therefore needs to take a longer time than an uncorrupted device would take. Hence, our scheme needs to measure the interval between sending a challenge in Verifier and arrival of that challenge at Dev.

3.1.1 Design Considerations. Although the basic idea is quite simple, the scheme is challenging to realize, for several reasons:

First, it is very difficult to precisely measure the desired time interval. The two events take place on Verifier and Dev, respectively. Since they are not expected to have perfectly synchronized clocks, we cannot obtain precise start and end times in the two devices.

Second, displaying an image on the screen and capturing that image are independent events. The camera periodically generates the image frame according to its frame rate. The image may be displayed anytime during the camera' operation cycle denoted by τ_{cam}. In the rest of the paper, we refer to this as the *D2C* (display-to-camera) interval, as illustrated in Figure 3. Thus, the difference between the shortest and longest D2C intervals is about τ_{cam}, Naturally, we must consider the shortest interval for the benefit of the attacker, and the longest – for the normal case.

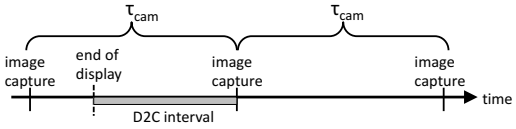


Fig. 3: The shadowed box represents the D2C interval. Its length is uniformly distributed in $[0, \tau_{cam}]$.

Caveat: The D2C interval can *not* be precisely measured by software, because the exact moment when the image is fully displayed is not known to the CPU. Unless τ_{cam} is significantly short, variance of the D2C interval should not be neglected.

Third, an image frame is essentially a matrix of pixels, displayed row-wise. If the adversary can derive the entire frame data after observing the first few rows, it does not need to wait for the entire frame to be displayed, and can hence take less time than expected. For example, after scanning the top row of a standard barcode, the adversary can infer the encoded binary string. Hence, images used between Verifier and Dev must not allow such shortcut attacks. For ease of presentation, we refer to qualifying images as *full-view* images.

Finally, the recipient must detect whether the camera captured the *entire* image of the displayed challenge. As shown in Figure 4,

the image produced by the camera might only take part of the challenge image, because the display has not yet rendered the entire screen when the camera took the picture. This partial capture could result in: (1) false positives with a genuine device, or (2) false negatives, since the adversary's cost is reduced by relaying only part of the challenge. We refer to this issue as the *full-screen* problem.

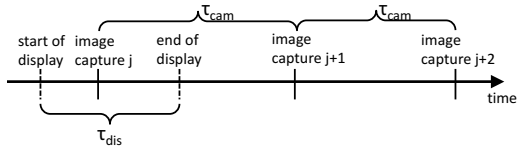


Fig. 4: Illustration of the full-screen problem: j-th capture only gets the top half of the displayed challenge, while both (j+1)-st and (j+2)-nd captures produce the entire image of the challenge.

3.1.2 Protocol Details. The residence checking scheme uses the analog display-to-camera (D2C) channel, where the former is the sender and the latter – the receiver. Success of the scheme hinges on this channel's *analog transmission latency*, as defined below.

Definition 3.1. Analog Transmission Latency (ATL) of the D2C channel is the interval between (1) the time *when the sender's display starts to render the image on the screen*, and (2) the time *when the complete image is stored by the camera in the receiver's memory.*

Since both display and camera run at a constant rate, ATL of a D2C channel is expected to be relatively constant, except for any variance due to the D2C interval. Moreover, any analog relay in a D2C channel significantly increases its ATL. The idea behind the basic sight-based residence checking scheme is as follows:

TA (which is verified in the existence checking phase) asks Verifier to display a challenge image. It then captures that image and reports it along with measured *raw latency*. To compute ATL, Verifier refines raw latency by removing time intervals which are within raw latency, but not within ATL. For ease of presentation, we refer to them as *noise intervals*. If the image is genuine and the refined ATL is lower than the pre-defined threshold, Verifier asserts that TA indeed resides on Dev. Details are shown in Protocol 2.

As shown, TA and Verifier communicate over the normal network and the D2C channel. We use: *network-send*, *network-receive*, *camera-receive* and *display-send* to denote two sets of communication primitives for the respective channel. While the first two are standard network operations, the last two are elaborated below:

display-send To send an image, trusted software (e.g., TA in Dev or the kernel on Verifier) writes it directly to the frame buffer. The hardware automatically retrieves this data and renders it on the screen. If trusted software runs in an untrusted device, it cordons off the frame buffer to prevent any read access from the untrusted kernel.

camera-receive Trusted software responds to the camera's interrupt which signals the arrival of a new batch of image blocks. It immediately copies buffer contents to its own protected buffer and reconstructs the image. A camera-receive operation returns successfully if the received frame is a full-screen image. If trusted software runs on an untrusted device (e.g., Dev), it cordons off the camera buffer to prevent any write access from the untrusted kernel.

Residence Checking (Δ_{max}, P)

Before execution, Verifier generates a random picture P. TA in Dev is prepared for camera-receive.

5. (Dev) TA in Dev reads its clock to get current time T_s, and network-sends (T_s, σ_1) to Verifier, where $\sigma_1 = \mathsf{H}_k(T_s)$
6. (Verifier) Challenge: It network-receives (T_s, σ_1) and verifies integrity of T_s against σ_1. It display-sends P and measures network latency, τ_{net}, as well as noise interval, δ_V.
7. (Dev) Response: After network-sending (T_s, $H_k(T_s)$), TA camera-receives eP'. It reads the clock to get current time T_e, and network-sends (T_e, P', σ_2) to Verifier, where $\sigma_2 = \mathsf{H}_k(T_e - T_s, P')$.
8. (Verifier) Verification: After network-receiving (T_e, P', σ_2), Verifier performs the following steps:
   (a) Verifies integrity of T_e and P' against σ_2.
   (b) If $T_e - T_s - \delta_V - \tau_{net} < \Delta_{max}$ and $P' \equiv P$, declares that DRTM and TA reside on Dev. Otherwise, they reside on M.

Protocol 2: (Cont. from Protocol 1) Basic sight-based residence checking protocol: $\Delta_{max} = \tau_{dis} + 2\tau_{cam}$. Notation $P' \equiv P$ means that they are visually equal.

Note that it is difficult to directly measure ATL because the start and end involve two distinct devices. In our protocol, TA times two events and obtains $T_e - T_s$, which is the raw latency. All noise intervals for Protocol 2 are in Verifier. They include network transmission (τ_{net}) and time (δ_V) between packet arrival and Verifier's display starting to render the image.

Maximum Latency with No Attack. Figure 5 illustrates time lapses measured by raw latency ($T_e - T_s$) in Protocol 2. The channel's ATL refers to the period between t_1 and T_e, as shown in Figure 5. It

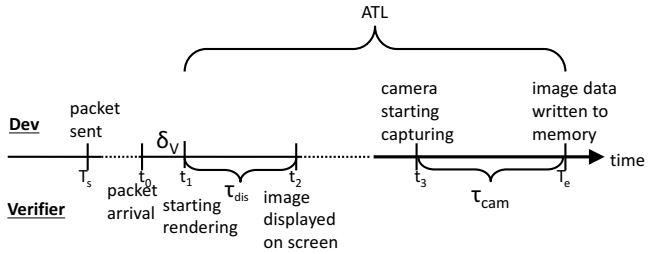


Fig. 5: Sequence of events between T_e and T_s. Dashed lines indicate variable times, while solid lines denote constant times. t_0, t_1, t_2, t_3 denote events of: packet arrival at Verifier, image rendering start, image displayed on screen, and image capture, respectively.

includes three intervals, all of which involve hardware operations:

- $t_1 \rightarrow t_2$: Time for Verifier's screen to display the image. This depends on the display frame rate. Let τ_{dis} denote the time to render the entire screen, which is close to the inverse of the refresh rate.
- $t_2 \rightarrow t_3$: D2C interval, at most τ_{cam}, which is the inverse of the camera's frame-per-second (fps) rate.
- $t_3 \rightarrow T_e$: Time for the camera to deliver the image frame to the DMA buffer; its length is also τ_{cam}.

Therefore, $\Delta_{max} = \tau_{dis} + 2\tau_{cam}$ is the longest possible ATL for intact Dev in the basic sight-based residence checking protocol.

3.1.3 Security Analysis. We first show that a collusive and software-only attack cannot pass presence attestation, and then analyze analog attacks. In the analysis below, we ignore CPU time consumed by code execution. Since no heavy computation is involved, the sum of CPU time in both Verifier and Dev is two orders of magnitude less than the time for analog operations.

Software-Only Attacks. Residing on Dev and M, the adversary is represented by malware with no hardware assistance. This malware controls the kernels of both Dev and M. If neither Dev nor M launches genuine DRTM-s, the adversary cannot pass existence attestation.

Suppose that the adversary launches DRTM and TA on M. The cuckoo attack allows it to successfully pass the existence checking phase. Since the resulting secret key k is securely held by TA, it is infeasible for malware to access k. Note that TA reads the challenge image directly from the camera's DMA buffer which can only be written to by the camera. Hence, malware cannot "feed" the image to TA, and a well-formed response cannot be returned to Verifier.

Analog Attacks. An analog adversary enhances the plain cuckoo attack by attacking the D2C channel. In essence, the adversary sets up an "image relay" that acts as the man-in-the-middle between Verifier and TA in M.

Ahead of time, the adversary sets up another computer (accomplice) with a display facing M's camera. TA on M successfully runs Protocol 1 and sends a request to Verifier via the network. When Verifier displays a challenge image, Dev's camera captures it, passes it to malware which then forwards it (via the network) to the accomplice, to be displayed. Finally, TA in M captures the image and computes channel latency.

Figure 6 illustrates minimum channel latency for this attack. t_1, t_2, t_3 are defined the same as in Figure 5. To give the adversary maximal advantage, we assume that D2C intervals (as defined in Section 3.1.1) are of negligible length, meaning that both $t_3 - t_2$ and $t_7 - t_6$ are close to zero. The interval between t_4 and t_5 depends on the adversary's network channel between Dev and the accomplice. We consider $t_5 - t_4$ to be negligible, assuming that the network used by the adversary is invisible to Verifier. We also disregard (as negligible) the CPU time taken by the adversary. Let $\bar{\Delta}_{min}$ denote minimum channel latency under the analog cuckoo attack. Hence, we have:

$$\bar{\Delta}_{min} = 2\tau_{dis} + 2\tau_{cam} \text{ and } \bar{\Delta}_{min} - \Delta_{max} = \tau_{dis}$$

Thus, in the best case, the adversary still needs τ_{dis} longer time than the longest delay incurred by uncorrupted Dev. Since τ_{dis} is the inverse of the display refresh rate (e.g., 60Hz for many modern displays), it is large enough to be detected by Verifier.

3.2 Extension: Iterative Checking

We extend the basic scheme to detect the presence of the analog adversary, armed with a high-end display with τ_{dis} of only a few milliseconds. The basic idea is to amplify a relatively short delay of one-round analog transmission into a substantially higher latency.

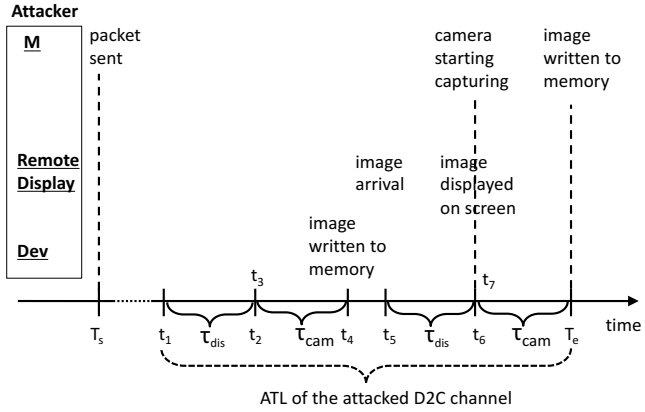


Fig. 6: Example of ATL of D2C channel under analogy cuckoo attack.

The extended protocol requires multiple *consecutive* rounds of D2C channel transmission, which clearly incurs additional user burden.

For effective latency amplification, a new round of transmission must begin *after* completion of the prior round, i.e., there should be no pipelining. Therefore, the receiver has to securely notify the sender about completion. We use the same D2C channel for the returning notification, such that a man-in-the-middle attack would incur higher delays. Hence, the extended scheme requires both Verifier and Dev to have their cameras facing the other's display. The resulting scheme is described in Protocol 3. The notification sent from TA to Verifier in Step 5 is a visualized HMAC, that allows Verifier to verify authenticity. Thus, the adversary cannot impersonate TA by sending a fake notification, which would lead to pipelined transmission and defeat the purpose of amplification.

Noise intervals on Dev and Verifier– of lengths δ_D and δ_V – refer to the time from receipt of a valid image frame produced by the camera, until the display starts rendering, on respective devices. These intervals, as well as images of h_i-s, are described in detail in Section 5.

Maximum Latency under No Attack. At completion of Protocol 3, TA display-sends $(h_0, \cdots, h_n)$ and Verifier display-sends $(P_0, \cdots, P_{n-1})$. Note that starting time T_s is right after receiving h_0. Thus, only transmission times for $(h_1, \cdots, h_n)$ are factored into $T_e - T_s$. In total, the protocol involves $2n$ D2C transmissions and maximum latency of accumulated ATL is: $\Delta^I_{max} = 2n(\tau_{dis} + 2\tau_{cam})$, assuming identical hardware at both end-points.

Minimum Latency for Adversary. The adversary can use both a display and a camera to relay images between Verifier and TA. Minimum delay in relaying P_i, denoted as $\bar{\Delta}^{v2d}_{min}$, is the same as in the basic protocol. Thus, we have:

$$\bar{\Delta}^{v2d}_{min} = \tau_{dis} + 2\tau_{cam} + \bar{\tau}_{dis}$$

where $\bar{\tau}_{dis}$ is the frame rate of the adversary's special (i.e., faster refresh rate) display. Minimum delay in relaying h_i, denoted as $\bar{\Delta}^{d2v}_{min}$, is computed as:

$$\bar{\Delta}^{d2v}_{min} = 2\tau_{dis} + \tau_{cam} + \bar{\tau}_{cam}$$

Iterative Residence Checking (Δ^{ite}_{max}, n, P_0, P_1, $\cdots$, P_{n-1})

Before execution, Verifier generates n random images $\{P_0, P_1, \cdots, P_{n-1}\}$, and TA computes $\{h_0, \cdots, h_n\}$ where $h_i = \mathsf{H}_k(i, \text{Verifier}||\text{TA})$ for $i \in [0, n]$, and converts them into $n+1$ images. Both Verifier and TA are ready for camera-receive with counter $i = 0$.

5. (Dev) TA in Dev display-sends the image for h_i. Next, based on the value of i:
   - $i < n$, TA camera-receives P'_i and saves it. It computes $i = i + 1$, and goes to Step 5.
   - $i = n$, goes to Step 7.
6. (Verifier) Challenge: It camera-receives and decodes h_i, and verifies integrity. Abort, if not valid. Next, based on the value of i:
   - $i = 0$: It reads its clock to get T_s, sets $i = i + 1$ and display-sends P_0. Go to Step 6.
   - $i < n$: it display-sends P_i and sets $i = i + 1$. Go to Step 6.
   - $i = n$: It reads its clock to get T_e. Go to Step 8.
7. (Dev) Response: TA network-sends $(P'_0, \cdots, P'_{n-1}, \delta_D, \sigma_2)$ to Verifier, where δ_D is duration of noise intervals in TA, and $\sigma_2 = \mathsf{H}_k(P'_0, \cdots, P'_{n-1}, \delta_D)$.
8. (Verifier) Verification: Verifier performs the following steps:
   (a) It network-receives $(P'_0, \cdots, P'_{n-1} - \delta_D, \sigma_2)$, and verifies its integrity.
   (b) If $T_s - T_e - \delta_D - \delta_V < \Delta^I_{max}$ and $P'_i \equiv P_i$, for all $i \in [1, c]$, it declares that the DRTM and TA reside on Dev. Otherwise, they reside on M.

Protocol 3: (Cont. from Protocol 1) Iterative Sight-based Residence Checking: $\Delta^I_{max} = 2n(\tau_{dis} + 2\tau_{cam})$

where $\bar{\tau}_{cam}$ is the frame rate of the adversary's camera. Total minimum latency $\bar{\Delta}^I_{min}$ is then computed as:

$$\bar{\Delta}^I_{min} = n(\bar{\Delta}^{v2d}_{min} + \bar{\Delta}^{d2v}_{min}) = 3n(\tau_{dis} + \tau_{cam}) + n(\bar{\tau}_{dis} + \bar{\tau}_{cam})$$

Hence, we have:

$$\bar{\Delta}^I_{min} - \Delta^I_{max} = n(\tau_{dis} - \tau_{cam}) + n(\bar{\tau}_{dis} + \bar{\tau}_{cam}) \quad (1)$$

Equation 1 implies that, even if the adversary uses the fastest display and camera, such that $\bar{\tau}_{dis} \approx \bar{\tau}_{cam} \approx 0$, our scheme can still detect the fastest attack, provided that $\tau_{dis} > \tau_{cam}$. Note that neither τ_{dis} nor τ_{cam} is chosen by the attacker. Moreover, it can be amplified linearly with multiple iterations. In practice, a typical smartphone display has a 60 Hz refresh rate, while a high-end camera can reach a rate of about 120 frames per second. In such a setting, $\tau_{dis} - \tau_{cam}$ alone is about 8 msec. Hence, the iterative residence checking scheme can detect an analog cuckoo attack.

4 SECURITY AND USABILITY

Residence checking protocols proposed in Section 3 require a line-of-sight channel between Verifier and Dev. In the basic protocol, Alice needs to position Dev to point at Verifier, while the extended protocol requires both Verifier and Dev have to be properly positioned. Moreover, Alice has to inform Verifier about relevant parameters, including τ_{dis} and τ_{cam}. This results in strong security, which is, unfortunately, commensurate with relatively poor usability. In this section, we consider software-only attacks and propose two protocols with better usability, though weaker security.

4.1 Scene-Based Residence Checking

Scene-based attestation also assumes that Dev has a camera. The scheme is essentially a challenge-response protocol between Alice and TA. Alice picks a random physical object in its physical proximity or environment and uses it as a challenge. TA is expected to respond with the object's image directly produced from its own camera. Assuming integrity of the hardware, and that the analog channel is not attacked by the software-only adversary, a correct response implies that TA's hosting device's camera indeed "sees" the challenge object. Therefore, the device in question must be Dev.

Scene-based Residence Checking Protocol.

5. After receiving b from Verifier, TA prepares for residence attestation by clearing the camera's DMA buffer and locking it.
6. After sending b, Verifier notifies Alice to prepare the challenge. Alice randomly chooses a physical object in her current environment. For example, she might use the whiteboard or a piece of paper to write one or more random number(s) or draw arbitrary pictures, or simply select a random scene in her immediate vicinity. Alice then points Dev at the chosen object (from a close distance) and takes a photo with Dev.
7. Verifier $\overset{\sigma, \mathtt{I}}{\leftarrow}$ Dev. TA obtains the image of the object by directly fetching raw bytes from the camera's DMA buffer. Let $\mathtt{I}$ denote the image data. TA computes $\sigma = \mathsf{H}_k(I)$.
8. Verifier verifies validity of σ using k and $\mathtt{I}$. If verified, it displays $\mathtt{I}$ on its screen; otherwise, Verifier displays an error message.
9. Alice manually checks whether $\mathtt{I}$ displayed by Verifier matches her chosen object. If so, she concludes that the trust anchor is indeed on Dev and protected by ROT_D.

Protocol 4: (Cont. from Protocol 1) Scene-based Residence Checking Protocol

Details are described in Protocol 4, which runs after Protocol 1. Basically, TA clears the camera's DMA buffer on its hosting platform, and locks it (Step 5), such that *only* the camera can write into it. It ensures that data later fetched from the buffer is indeed delivered by the camera, and not by malware. Thus, TA's response (in Step 7) faithfully reflects the physical environment of the hosting device.

Discussion. Protocol 4 is secure against software-only cuckoo attacks. Under such an attack, TA on M only takes input from hardware, and a software-only adversary cannot feed TA any images captured by Dev's camera.

Theoretically, this protocol can be defeated by the analog adversary using the scenario described in Section 3. Nonetheless, we believe that, in practice, the protocol may withstand analog attacks *to some extent*. There is notable difference in effort between (adversarial) ability to photograph: (1) a physical object in Alice's and Dev's private environment, and (2) an image displayed on the screen in a separate (adversary-controlled) environment. This difference is based on several factors, such as: ambient lighting, distance between the camera lens and the target, as well as reflections of various nearby objects. The adversary that photographs displayed images would quite likely sacrifice fidelity of the real object, or include objects that are not in Dev's environment.

Furthermore, the scene-based scheme can be extended to replace a photo with a short video clip. The camera in Alice's environment

would record a normal clip, while the clip produced in the adversary's environment would show the refreshing of the screen when the camera's frame rate is higher than that of the screen. Therefore, Alice would be able to decide whether the result is indeed obtained from her environment. Unfortunately, this would significantly complicate the design of TA due to the video clip being generated from raw data. We leave this issue for future work.

4.2 Location-Based Residence Checking

Location-based residence checking assumes that Alice is aware of her present location and Dev has a GPS[5]. The basic idea is to use Alice's present location as an implicit challenge to TA which is expected to report a matching location by securely obtaining its hosting device's GPS data. Since GPS signals can be spoofed by the adversary with physical equipment, this scheme is secure just against software-only attacks.

Similar to the scene-based attestation protocol, TA needs to clear the GPS DMA buffer and lock it, such that any data in the buffer is faithfully reported by GPS. The protocol is presented in Protocol 5.

Location-Based Residence Attestation.

5. After receiving b from Verifier, TA prepares for residence attestation by locking the DMA buffer used by the device's GPS.
6. Verifier $\overset{\sigma,L}{\leftarrow}$ Dev. TA obtains its present location coordinates by directly fetching raw bytes from the GPS DMA buffer. Let L denote the location data. TA computes $\sigma = \mathsf{H}_k(L)$.
7. Verifier verifies validity of σ using k and L. If verified, it highlights L on the displayed map; otherwise it displays an error message on the screen.
8. Alice manually checks whether the highlighted location matches her present environment. If positive, Alice concludes that the trust anchor is indeed on Dev and protected by $\mathsf{ROT_D}$.

Protocol 5: (Cont. from Protocol 1) Location-based Residence Checking Protocol

The main advantage of this scheme is its minimal human involvement, since Alice does not need to move or reposition Dev. The protocol can even be combined with the existence checking protocol without time gaps in Protocol 2, 3 and 4.

Caveat. The present scheme is *not* secure against the adversary positioned very near Alice, since, in that case, TA on the adversary's device reports the same location as Dev.

4.3 Other Analog Attacks

We now consider another potential attack setting for the analog adversary. In it, the adversary takes advantage of at least one extraneous camera (denoted by CAM) in the immediate vicinity of Alice, Dev and Verifier. The attack is more similar to shoulder-surfing, than to a cuckoo attack. The main idea is that, if properly positioned, CAM can take a reasonably accurate photo of the screen at about the same time as Dev's camera. Therefore, if CAM is wired to M (or they are one and the same), the proposed sight-based schemes can be defeated, since the adversary no longer needs any analog relay. There are two flavors of this attack:

[5] Although it is possible to locate a device via Wi-Fi and cellular signals, the complex analysis is ill-suited for the trust anchor due to bulky code.

[1]: The adversary has *prior physical access* to the premises where presence attestation will take place. Placement of CAM and M must be proactive and precise to anticipate the exact location of the presence attestation process. Although possible, this attack is complicated. If a ceiling-mounted camera is used, its angle must be sufficient to subsume Verifier's screen. Also, any screen privacy film used on Verifier would make it nearly impossible take a photo. Of course, if during attestation Alice is physically accompanied by a live (real-time) adversary Eve, who surreptitiously takes a photo using her device (CAM+M), all bets are off.

[2]: The adversary that has *no physical access* to the premises. However, it takes advantage of cameras common in many office and workplace environments. Having control over a multitude of CAMs, the adversary is not limited to targeting only one space, such as Alice's office. On the other hand, recall that M is assumed to be of the same type as Dev, which is different from a typical IoT-style camera exemplified by pre-installed CAM. Specifically, CAM would most probably lack a DRTM. Therefore, a successful attack would require negligible communication delays between M and CAM, which is highly unlikely.

5 IMPLEMENTATION

We implemented Protocols 1–5 for a typical setting, where Dev (e.g., a smart-phone) has an ARM processor, while Verifier is a commodity x86-based computer. Specifically, we use a laptop as Verifier, and, as Dev, we use an ARM development board with an LCD screen, a USB camera and a GPS unit. There are no technical barriers for adapting this implementation to other platform settings, though some low-level hardware-dependent modifications would be necessary. This initial prototype implementation is rather complex, since it involves system security techniques, as well as intricate I/O mechanisms used by the display and the camera, as well as relevant image processing techniques.

5.1 DRTM and Trust Anchor

We built the *DRTM agent* in Dev's secure world as well as a micro-hypervisor running in Dev's HYP mode of the non-secure world. The agent, as well as the supporting ARM TrustZone hardware, are collectively considered to represent DRTM, while the micro-hypervisor acts as TA.

The hypervisor is measured and launched by DRTM agent at runtime, instead of during boot-up. Dynamic hypervisor launching on ARM platforms is supported by ARM specifications and has been implemented by Cho, et. al. [10]. Basically, the user-space application issues a system call to the underlying kernel, which in turn issues an SMC call to DRTM in the secure world. DRTM measures the hypervisor image and deploys it. Besides loading the hypervisor into memory, deployment includes setting up Stage I and II page tables and installing the hypervisor call (HVC) handler. Then, the DRTM agent returns to the kernel, which issues an HVC and traps to the hypervisor. The main drawback of this approach is that DRTM takes up too much responsibility and its code base is significantly expanded, due to hypervisor deployment.

Our implementation employs a more direct approach with the same security strength, yet with a smaller DRTM code base. When handling the SMC call for hypervisor launch, the DRTM agent still

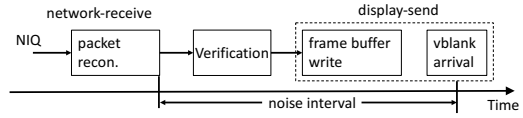
loads the measured hypervisor. However, it manipulates the hardware context, such that the SMC return goes to HYP mode, instead of the SMC calling site in the untrusted kernel. Specifically, `lr_mon` and `spsr_mon` registers are configured such that execution of the `eret` instruction in the secure world causes CPU to switch to HYP mode and returns to the hypervisor code previously prepared by the agent. Once the hypervisor takes control, it configures all paging structures and the HVC handler. In a similar way, the hypervisor creates a new hardware context, such that execution of `eret` in HYP mode returns to the untrusted kernel's SMC calling site. Note that, when the CPU runs in secure world, or in HYP mode, the untrusted kernel cannot preempt its execution.

Compared with the standard mechanism in Cho, et. al. [10], our system saves one CPU mode switch and the DRTM agent is hypervisor-agnostic, since it does not handle internals of the hypervisor, such as paging structures. Therefore, the logic is much simpler and the code-base for dynamic loading is smaller. Our implementation requires only 90 lines of assembly and 230 lines of C code for dynamic measured launch.

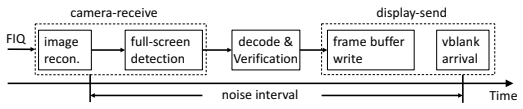
5.2 Sight-based Residence Checking Schemes

Since sight-based Protocols 2 and 3 are time-critical, we carefully implemented them for Verifier and TA to ensure that incurred CPU time is *minimized* and as *stable* as possible. Both requirements are crucial to protocol security.

5.2.1 The Verifier. We implement a protocol agent which runs as a kernel thread to execute residence checking protocols. Since Verifier is fully trusted, we do not consider security issues in the implementation. Figure 7 depicts the main steps of Verifier as well as the breakdown of Verifier's noise intervals in Protocol 2 and 3. To accurately measure these noise intervals, the protocol agent stalls all other CPU cores in Verifier and sets itself as non-preemptive, such that it does not yield CPU to other threads.



(a) Verifier in Protocol 2



(b) Verifier in Protocol 3

Fig. 7: Breakdown of noise intervals in Verifier.

Below, we discuss the details of *display-send* and *camera-receive*. Some tasks, such as random picture generation and HMAC computation, are not time-sensitive since their CPU time is not factored into raw latency.

Camera-receive. The high-level scheme of USB camera I/O is as follows: To produce an image frame, the camera can transfer one or multiple groups of image blocks. An interrupt called *FIQ* is generated upon each block group transfer. The corresponding handler reads the blocks and assembles them into a frame.

Verifier camera-receives h_i in Protocol 3. The FIQ interrupt handler is hooked, such that, whenever a frame is successfully reconstructed, the protocol agent reads its birth time.

Caveat A camera's frame rate is not the frequency of block-group transfer. It refers to the number of image frames reconstructed by software in a second. In other words, τ_{cam} is essentially the time between two image reconstructions.

Another issue is the full-screen problem described in Section 3.1.1. Our method is based on two consecutive frames being approximately the same. The rationale is that, after Dev's display is fully rendered from the top-left to the bottom-right corner, it appears as a still picture to the camera. Therefore, images captured by the camera at different times should be visually the same and their pixel values should be close to each other[6]. As illustrated by $(j+1)$-st and $(j+2)$-nd captures in Figure 4, if two consecutive images have little variation, the former is the first image for the full screen and its birth time is exactly the starting time of the noise interval of Verifier in Protocol 3.

After successful image reconstruction from the camera buffer, Verifier recovers h_i from the image. Details are described in Section 5.2.3; this involves image encoding techniques.

Display-send. Since security is not a concern here, we take advantage of the existing graphic framework to display P_i in both protocols. The crux of Verifier's display-send operation is to precisely measure the end-point of the noise interval. We leverage Intel Direct Rendering Management (DRM). The graphics card driver prepares two frame buffers which are used alternatively: one is used by the hardware for displaying, and the other – for update. A periodic *vblank* interrupt triggers the driver to switch to the updated buffer for displaying.

The vblank interrupt handler is instrumented to check whether P_i has been placed in the frame buffer. If so, the hander reads the clock which marks the end of Verifier's noise interval.

5.2.2 The Device. The implementation on Dev is more complex than on Verifier, because it is built into the ARM hypervisor and Dev is *not* a trusted device. On one hand, the implementation must ensure protocol security with high accuracy and little time variation. On the other hand, we must refrain from significantly expanding hypervisor code size and logic. Our strategy is to maximize the use of the untrusted kernel to perform non-sensitive tasks. The hypervisor, acting as TA, runs the attestation protocol(s) involving two main primitives as follows:[7]

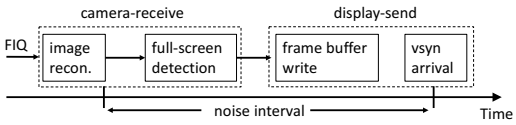


Fig. 8: Implementation details of Dev in Protocol 3.

[6]The two images are not byte-wise identical due to inaccuracy of the image sensing.
[7]Note that the hypervisor also stalls other CPU cores and acquires the highest scheduling priority for precise and stable time measurement.

camera-receive. The hypervisor uses the camera-receive to obtain random images from Verifier. It uses the same technique to solve the full-screen problem as Verifier. Note that there is no noise interval for Dev in the basic sight-based protocol, since TA's operations after reconstruction of P are not included in raw latency. The breakdown of the noise interval in the iterative sight-based protocol is described in Figure 8. The only difference between Figure 7(b) and Figure 8 is that the latter does not need to verify data authenticity. Similar to camera-receive of Verifier, the start of the noise interval in the iterative protocol is the time when the random image from Verifier is completely assembled.

The main security requirement is that the origin of the image must be Dev's camera. In other words, any update to the camera buffer can only come from the camera itself. Any software's write access to the buffer breaks the security premise of residence checking. Since memory operations are orders of magnitude faster than camera's analog operation, the adversary with write accesses to the buffer can significantly reduce its ATL and easily pass the time checking.

For this purpose, the hypervisor uses Stage II page table to configure the camera's frame buffer as read-only. The untrusted kernel can still read the buffer. However, any write to the buffer (from kernel or user spaces) is blocked by hardware. To deal with DMA attacks from the kernel, the hypervisor also configures the SMMU page tables used by peripheral devices, such that no device (except the camera) can access the buffer. For ARM platforms without SMMU page tables, the hypervisor intercepts I/O operations and inspects the DMA descriptors.

The FIQ interrupt handler in Dev's kernel is also hooked such that its execution is trapped to the hypervisor, which then assembles blocks in the camera buffers into an image frame. Although the kernel is untrusted, protection of the FIQ handler is actually not necessary. Any tampering with the handler leads the hypervisor to mistakenly assemble image blocks. However, this does not feed any counterfeit data (i.e., not from the camera) to the hypervisor. Thus, the adversary gains no advantage, except protocol disruption.

display-send. Our prototype of Dev is a development board with a low-end graphic card. Thus, there is no hardware signal synchronizing LCD refreshing and kernel frame buffer update. Since the display fetches the frame buffer data at fixed intervals, we simulate it by setting a timer with the same frequency. Note that the simulation has *no* effect to the basic protocol, which does not use display-send in Dev. We acknowledge that it introduces a constant offset between obtained δ_D and real values for the iterative protocol. Nonetheless, their distribution and average values are the same.

The security requirement of Dev's display-send is that displayed data must not be read by the adversary from memory. The adversary is thus forced to use the analog channel, which increases ATL. Therefore, the hypervisor properly configures the Stage II page tables and the SMMU tables to set the frame buffer as inaccessible.

5.2.3 Image Related Issues. As noted in Section 3, we can only use full-view images. Below we describe how full-view images are chosen and generated in our implementation.

P and P_i-s. Images displayed by Verifier to Dev are random, i.e., (almost) every region of the image is randomly generated. These images are prepared by Verifier before running presence attestation, in order to avoid the runtime overhead.

Note that the hypervisor is not burdened with image recognition. Instead, it only reconstructs the picture produced by the camera using the data in the camera buffer and offset information stored in registers. More details are in Section 5.4, which describes the implementation of the scene-based protocol.

Our current implementation does not include the algorithm to evaluate $P' \equiv P$. Instead, Alice is required to physically check for similarity. Image processing techniques can be applied to automate the process. We leave this item for future work.

Image of h_i-s. We do not use randomly generated images for the hypervisor, for two reasons: First, it is costly for the hypervisor to generate random images, in terms of code size/complexity, storage size, and time. Second, it is difficult for Verifier to check authenticity of those images. Since software in Verifier is trusted and has the full-fledged capability of image recognition and decoding data, it is more advantageous for the hypervisor to display visualized HMACs.

The hypervisor splits h_i into two binary strings, and converts them into two barcodes. It then constructs an image with one barcode at the top rows of the screen and the other – at the bottom. The rest of the image is solid white color. This layout ensures that any camera receiver must wait until the full image is displayed on the screen, because the screen always renders pixel rows top-down. Once Verifier camera-receives the image, it extracts two binary strings from the barcodes and reconstructs h_i for verification.

Since the time to reconstruct h_i is within the time measurement, it must be both short and constant. Verifier's camera is configured to produce images using the YUYV format, so that the camera's raw data is not compressed. The protocol agent in Verifier scans the image to read out the barcodes. Specifically, the agent reads pixel colors in a row. If a short black pixel segment is encountered, the agent outputs '0', and if a long black pixel segment is detected, it outputs '1'.

5.3 Location-based Residence Checking

GPS Receiver of Dev receives GPS data stream from the satellite in the form of the National Maritime Electronics Association (NEMA) sentences. These sentences are a sequence of ASCII characters, starting with '$' and ending with a carriage return. In every sentence, the first word describes its type. Geographic location information of Dev, including longitude and latitude, is in the "GPRMC" type. The receiver writes sentences to the `Data Register` (`DR`) one byte at a time. Arrival of a byte is indicated by the `Flag Register` (`FR`) where a flag bit is automatically set when the byte in `DR` is read by software.

For Protocol 5, the hypervisor in Dev denies all write accesses to `DR` by configuring Stage II page tables and SMMU page tables (if available). This ensures that data in the register is genuinely produced by on-board GPS receiver.

After setting up the restriction, the hypervisor continuously checks `FR` and reads one byte from `DR`. If it detects the word "$GPRMC", it copies subsequent characters to its buffer until it encounters a carriage return. The captured sentence contains the location of Dev. Hypervisor computes an HMAC and sends both GPS message and HMAC to Verifier. Since message secrecy is not a concern and it

is not time sensitive, hypervisor calls a user-space application to send them over the network.

Most users do not know the exact longitude and latitude of their locations. Thus, Verifier displays the location reported by hypervisor on a map and Alice manually verifies its correctness. Since GPS location accuracy can be within 5 to 10 meters, Alice might only be able to recognize the location based on buildings and street blocks.

5.4 Scene-based Residence Checking

In our platform for Dev, the device has a USB-camera which uses five DMA buffers at fixed physical addresses. The camera periodically writes to these buffers with a batch of image blocks of variable lengths. An image frame consists of one or more blocks, depending on image size. Although an image frame has a header with a unique frame identifier, a block has no header. A block's offset within the image and its size are stored in an array of USB Host Channel registers. Software is responsible for assembling the blocks into an image frame with the assistance of the registers.

We place a hook into the kernel's FIQ interrupt handler, which is invoked by hardware when the camera completes one batch of block delivery. The hook traps to hypervisor only when the latter is in the residence checking session. With the assistance of USB channel registers, hypervisor reads new blocks and assembles those with the same frame identifier into an image frame.

Hypervisor computes an HMAC over the image frame. Similar to the location-based scheme, we use a user space application to forward the image frame and its HMAC to Verifier over the network.

6 EVALUATION

We experimented with the prototypes to assess performance and security. In our setting, Dev is a Raspberry Pi-2 model B development board with 900MHz quad-core ARM Cortex-A7 CPU and 1 GB RAM[8]. It is connected with a Microstack GPS module[9] and a Logitech HD webcam C525 with maximum 30 fps[10], both of which are via USB. Verifier is a Toshiba laptop with 2.4 GHz Intel Core i7-5500 CPU and 8 GB RAM, with the same web camera as Dev.

In the rest of this section, we evaluate performance of proposed protocols before assessing their security. We do it in this order, since execution time is crucial to security of the two sight-based protocols.

6.1 Performance

We implemented DRTM in the TrustZone. It mainly consists of a hypervisor loader (90 lines of assembly and 234 lines of C code), and a 187KB cryptographic library customized from the Mbed TLS[11] version 1.3.10. We also implemented the hypervisor as the trust anchor on the Raspberry PI board. Table 1 lists sizes of all major prototype components in Dev and Verifier. The TCB in Dev consists of DRTM code and the hypervisor. The development board that we use was not shipped with a public key certificate. To this end, we manually installed RSA public/private key-pairs into its TrustZone to simulate the DRTM credential.

It takes about 4.5 msec to dynamically launch a hypervisor, from the moment of the application issuing an SMC call to the CPU returning to user mode. The main cost is due to hash operations. We also measure performance of the existence checking protocol. It takes about 998.5 msec on Dev to execute Protocol 1, where the dominant cost is RSA decryption and signing, each performed with a 1024-bit key. With a 2048-bit RSA key, the time shoots up to 5.16 sec.

6.1.1 Sight-based Residence Checking. Table 2 lists performance constants that can be measured without running the protocols. The image to send refers to data that Verifier or Dev needs to place into graphic frame buffers, which is not the same as image file size. The image to receive refers to the image frame reconstructed by software using data delivered by the camera. Values of τ_{cam} and τ_{dis} are derived from the respective hardware specifications, and τ_{net} is based on measurement on a lightly-loaded LAN. LAN congestion during the first network-send operation in Protocol 2 can only induce false positives, since it increases ATL. In our setting, τ_{net} = 6.3 msec, τ_{cam} = 33 msec, and τ_{dis} = 16 msec.

We also assess average CPU time of display-send and camera-receive. For the former, we measured the time for software's frame buffer update, and waiting period between completion of frame buffer update and arrival of the display synchronization signal. We split CPU time of camera-receive into image reconstruction and full-screen detection. We measured the period between the interrupt of the first block of an image frame to the time of the frame being assembled in memory. The time for full-screen detection is between completion of frame reconstruction and the moment when it is determined as full-screen. Results are summarized in Table 3. Note that image reconstruction time is not counted within the noise interval. In the camera I/O model, the fps rate takes into account image reconstruction time.

To assess overall performance, Table 4 shows average turn-around time of the two protocols, We also measured noise intervals in both Dev and Verifier. The dominant component of δ_D is full-screen detection time in Table 3, and dominant components of δ_V are: (1) display-send in the basic protocol, and (2) full-screen detection in the iterative protocol.

6.1.2 Location-based and Scene-based Protocols. We measured the turnaround time (as reported in Table 5) for both location-based and scene-based residence checking protocols. As a slow-speed device, GPS sends stream data to the buffer at 10 Hz frequency, which is the performance bottleneck for the protocol. Nonetheless, this protocol can be combined with existence checking to reduce overhead.

Turn-around time of the scene-based protocol is the sum of hypervisor's image reconstruction, network transmission delay and Verifier's user-space execution time, including HMAC verification and invocation of graphics library functions.

In both protocols, Alice's manual verification of the location and the image is not factored into the turnaround time. In our experiments, it does not take a noticeable delay for the user to verify presence. As part of our future work, we plan to conduct a user study to better understand verification behavior.

[8] https://www.raspberrypi.org/products/raspberry-pi-2-model-b/
[9] http://www.microstack.org.uk/products/microstack-gps/
[10] http://www.logitech.com/en-sg/product/hd-webcam-c525
[11] https://tls.mbed.org

	CPU Mode	Sight-based (Basic)	Sight-based (Iterative)	Location-based	Scene-based
Dev	Hyp	248 (142)	380 (150)	248 (88)	248 (127)
	Usr	204	54	272	282
	Svc	109	89	141	197
Verifier	User	703	710	286	248
	Kernel	46	347	-	-

Table 1: Code size in Dev and Verifier (in SLOC). The numbers of assembly code lines are in the brackets.

	Verifier	Dev
Image to send (KB)	54	306
Image to receive (KB)	203	21
barcode detection (msec)	0.012	-
HMAC speed (ms/KB)	0.05	0.1

Table 2: Constants in performance measurement.

		Verifier	Dev
Display-send	frame buffer update	0.02	0.5
	waiting time	7.9	8.2
Camera-receive	Image reconstruction	31.9	14.9
	Full-screen detection	33.2	31.8

Table 3: Time for component steps in display-send and camera-receive (msec)

	Basic	Iterative ($n = 1$)
Protocol Turnaround	132	200.8
Verifier's noise interval δ_V	15.6	17.0
Dev's noise interval δ_D	-	44.7

Table 4: Turn-around for two sight-based protocols and noise intervals in Dev and Verifier (msec)

	Location-based		**Scene-based**	
	Dev	Verifier	Dev	Verifier
Turnaround	403.1	23.0	23.9	4.4

Table 5: Turn-around for location- and scene-based protocols (msec)

6.2 Security of Sight-based Protocols

Since security of sight-based protocols depends on ATL of the channel, we ran experiments to evaluate whether basic and iterative protocols correctly verify an uncorrupted device and detect analog cuckoo attacks.

6.2.1 Attestation under No Attack. As described in Section 3, the D2C interval is uniformly distributed in $[0, \tau_{cam}]$. Hence, ATL in the basic protocol follows the uniform distribution between: $\Delta_{min} = \tau_{dis} + \tau_{cam}$ and $\Delta_{max} = \tau_{dis} + 2\tau_{cam}$, which are 49 msec and 82 msec, respectively. Thus, the average is 65.5 msec and standard deviation is 9.5 msec.

ATL in the iterative protocol for $n = 1$ is the sum of two independent ATL-s for two separate D2C channels. Therefore, it follows Irwin–Hall distribution between $[\Delta^I_{min}, \Delta^I_{max}]$, where $\Delta^I_{max} = 2\Delta_{max} = 164$ msec, and $\Delta^I_{min} = 2\Delta_{min} = 98$. The average is $2 \times 65.5 = 131$ msec and standard deviation is $\sqrt{2} \times 9.5 = 13.4$ msec.

We ran each protocol 50 times and computed: average, maximum, minimum, and standard deviation of measured ATLs. As shown in Table 6, results corroborate our analytical ATL models in each protocol. Largest ATLs are below Δ_{max} and Δ^I_{max}. Furthermore, we observed no false positives.

	Basic		**Iterative** ($n = 1$)	
	Δ_{min}	49	Δ^I_{min}	98
	Δ_{max}	82	Δ^I_{max}	164
	Analytic Model	**Experiment**	**Analytic Model**	**Experiment**
ATL	65.5	64.0	131	127.5
max. ATL	82	81.7	164	156.3
min. ATL	49	50.1	98	108.5
std. of ATL	9.5	8.1	13.4	16.9

Table 6: ATL in the basic and iterative protocols (msec)

6.2.2 Attestation Under Analog Cuckoo Attacks. To mimic the attack on the basic protocol, we used a Sony Cybershot DSC-RX100 camera to simulate both Dev and the accomplice display. The Sony camera's lens faces Verifier's display, while the camera attached to the Raspberry board (acting as M) is faces the Sony camera's LCD screen. We turn the Sony camera into the video mode, such that it simultaneously plays the role of Dev and the accomplice screen. Compared with the attack setting in Section 3, this experiment setting involves no network transmission and no software execution.

Results are shown in Table 7. All attacks are detected by Verifier since incurred ATLs are above Δ_{max}. Under our analytic model, the smallest ATL the adversary can achieve is 98 msec. In fact, attack ATL follows the same distribution as the ATL of the iterative protocol with $n = 1$, since both scenarios have two transmissions over the D2C channel. Note that the adversary cannot predict (or manipulate) the D2C interval.

	Basic	**Iterative** ($n = 1$)
ATL	129.9	248.3
max. ATL	164.0	275.1
min. ATL	110.0	220.3

Table 7: ATL of basic and iterative protocols, manipulated by the adversary. Theoretical lowest values are: $\bar{\Delta}_{min} = 98$ msec and $\bar{\Delta}^I_{min} = 196$ msec.

To simulate the attack on the iterative protocol, we introduce another digital camera (Olympus OM-D EM-10). The Olympus camera's lens faces the Raspberry board's display while its LCD screen faces Verifier's camera. Both digital cameras then concurrently relay screen images from Verifier to M and vice-versa. As shown in Table 7, all ATLs are above Δ^I_{max}. In other words, no false negatives are observed.

Basic v.s. Iterative Table 7 also shows that the iterative protocol is stronger than the basic protocol. We compare the gap between observed smallest ATL and its largest legal value. It is 28 msec for the basic protocol and 56.3 msec for the iterative one. We also compare the observed smallest ATL against its lower bound. It is

only 2 msec for the basic protocol, while it rises to 24 msec for the iterative one. This is because the probability of reaching the lowest value in the iterative protocol ($n = 1$) is the square of the probability in the basic protocol.

7 RELATED WORK

The topic of this paper is related to several research areas. This section overviews related work in each.

DRTM. The first effort to take advantage of hardware DRTM is Flicker [20] which launches a tiny secure execution environment based on AMD SVM technology [2]. Subsequent results, notably TrustVisor [19] and XMHF [32], launch a bare-metal micro-hypervisor by using Intel TXT [12] which is also used by Intel's own trusted boot-loader. Recent advent of Intel SGX [11] represents a stronger form of DRTM. It was shown to be a powerful tool in some recent literature [6, 26]. ARM TrustZone [3] could also be considered as a special type of DRTM, although its TCB is bigger than those of its counterparts on x86 platforms. Code protected by TrustZone can dynamically measure and launch the hypervisor [10]. Azab, et. al. propose to use TrustZone to provide kernel's runtime security [4]. Although the aforementioned schemes are secure in their respective adversary models, none of them considers the role of the human user in trust establishment.

Attestation. Research on remote attestation starts with TPM-based[31] static attestation [25] which only allows a trusted remote verifier to check static code integrity of the untrusted remote prover. Many subsequent research efforts have extended remote attestation from code integrity to encompass more expressive and dynamic properties [1, 7, 9, 14].

Some static attestation methods are based purely on software, under some assumptions about underlying hardware performance. For example, SWATT [28] and VIPER [18] do not require a hardware root of trust. Instead, by relying on carefully crafted memory traversal algorithms that compute measurements, these schemes can detect malware presence by precise timings (under the assumption that malware attempts to hide its presence and is thus forced to copy itself in chunks, which takes extra time). Similar to our schemes, a malware-infested device takes longer time to attest than an intact one. However, our schemes are more reliable and effective, since they are based on analog operations with more significant delays.

Distance Bounding. Since our work involve measuring communication delays, it is also somewhat related to distance-bounding protocols [8, 24]. In principle, distance bounding protocols might be applicable to the presence attestation problem, since the verifier can use them to determine the upper bound on the distance to the prover. Nonetheless, such protocols are extremely sensitive to time and require high-precision clocks. Also, they cannot tolerate the variance caused by software execution.

Virtualization-based Security. Many security architectures have been proposed based on a bare-metal micro-hypervisor, including: SecVisor[27], TrustVisor [19], InkTag[15], and MiniBox[17] on x86 platforms, as well as: XNpro [22], OSP [10], and H-Binder [29] on ARM platforms. Compared with DRTM, the hypervisor is more versatile and adaptive. As shown by XMHF [32], the DRTM measures and launches a micro-hypervisor, and the latter (acting as a trust anchor) extends the security perimeter to protect higher-level software. This paradigm combines the advantages of both DRTM and the hypervisor.

User Involvement. Both Lange, et. al. [16] and Danisevskis, et. al. [13] describe a means for a human user to establish trusts in her device via a secure user interface. The main idea is to isolate a small bar at the top of the device's screen that shows whether the critical virtual machine is running. In these methods, the hypervisor is launched *before* kernel initialization, and it is trusted to be always present. In contrast, our focus is on *presence attestation* – a more difficult problem, since the hypervisor is launched *after* (potentially corrupted) kernel execution.

TrustICE [30] is a TrustZone-based isolation method which involves an LED light solely controlled by software in the TrusZone's "Secure World". As acknowledged in the beginning of this paper, this currently represents the strongest hardware-based approach. Unfortunately, it lacks compatibility and requires hardware vendors' cooperation. Another less related result is "Seeing-is-Believing" [21], wherein the human-aided camera-based channel is used to obtain a public-key credential from a smartphone and bootstrap a secure channel.

8 CONCLUSIONS

This paper investigated how a human user can ascertain DRTM presence on her own computing device. The threat of cuckoo attacks makes this a challenging problem due to the gap between the hardware and the human user. We tackled this challenge with a two-step approach: (1) assisted by a trusted verifier device, the user first checks for existence of a DRTM in the interaction, and then (2) uses the residence-checking protocol to decide whether DRTM indeed resides on her device. We proposed three flavors of presence attestation: sight-, location- and scene-based. The sight-based variant offers the strongest security, since it can detect analog cuckoo attacks, while the other two offer better usability, commensurate with slightly weaker security.

Future work is planned in two directions. First, we intend to better understand proposed schemes via usability studies. and thus assess user burden, as well as reliability and error-prone-ness. Second, we plan to explore new presence attestation techniques under weaker security assumptions and resistant to compromised verifier devices.

ACKNOWLEDGEMENTS

Authors are grateful to the ACM CCS'17 anonymous reviewers for their constructive suggestions. This research was supported, in part, by the Singapore National Research Foundation under the NCR Award: NRF2014NCR-NCR001-012. Gene Tsudik's research was supported by funding from: (1) the Department of Homeland Security, under subcontract from the HRL Laboratories, (2) the Army Research Office (ARO) under contract: W911NF-16-1-0536, and (3) the Fulbright Foundation. Zhoujun Li's work was funded by National High Technology Research and Development Program of China (No.2015AA016004), National Natural Science Foundation of China (61672081, 61602237, 61370126,U1636211,U1636208).

REFERENCES

[1] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadegi, and G. Tsudik. C-FLAT: Control-flow ATtestation for embedded systems software. In *Proceedings of ACM CCS*, 2016.

[2] AMD. Secure virtual machine architecture reference manual. Technical report, Advanced Micro Devices, 2005.

[3] ARM. ARM security technology - building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[4] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

[5] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning. SKEE: A lightweight secure kernel-level execution environment for ARM. In *Proceedings of NDSS*, 2016.

[6] M. Barbosa, B. Portela, G. Scerri, and B. Warinschi. Foundations of hardware-based attested computation and application to sgx. In *Proceedings of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

[7] E. F. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In V. Atluri, B. Pfitzmann, and P. D. McDaniel, editors, *ACM Conference on Computer and Communications Security*, pages 132–145. ACM, 2004.

[8] S. Capkun and J.-P. Hubaux. Secure positioning in wireless networks. *IEEE Journal on Selected Areas in Communications: Special Issue on Security in Wireless Ad Hoc Networks*, February.

[9] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A.-R. Sadeghi, and C. Stüble. A protocol for property-based attestation. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 7–16, New York, NY, USA, 2006. ACM Press.

[10] Y. Cho, J. Shin, D. Kwon, M. J. Ham, Y. Kim, and Y. Paek. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *USENIX ATC*, 2016.

[11] I. Corporation. Innovative instructions and software model for isolated execution. http://privatecore.com/wp-content/uploads/2013/06/HASP-instruction-presentation-release.pdf.

[12] I. Corporation. Intel Trusted Execution Technology (Intel TXT) software development guide, Dec 2009.

[13] J. Danisevskis, M. Peter, J. Nordholz, M. Petschick, and J. Vetter. Graphical user interface for virtualized mobile handsets. In *MOST*, 2015.

[14] K. Eldefrawy, A. Francillon, D. Perito, and G. Tsudik. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA*, San Diego, UNITED STATES, 02 2012.

[15] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[16] M. Lange and S. Liebergeld. Crossover: secure and usable user interface for mobile devices with multiple isolated OS personalities. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 249–257, 2013.

[17] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference*, 2014.

[18] Y. Li, J. M. McCune, and A. Perrig. VIPER: verifying the integrity of peripheral's firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[19] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient TCB reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P)*, 2010.

[20] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, Apr. 2008.

[21] J. M. McCune, A. Perrig, and M. K. Reiter. Seeing-is-believing: Using camera phones for human-verifiable authentication. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05*, 2005.

[22] J. Nordholz, J. Vetter, M. Peter, M. Junker-Petschick, and J. Danisevskis. Xnpro: Low-impact hypervisor-based execution prevention on arm. In *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices*, TrustED '15, pages 55–64, New York, NY, USA, 2015. ACM.

[23] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[24] K. B. Rasmussen and S. Capkun. Realization of rf distance bounding. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[25] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium*, pages 16–16, 2004.

[26] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Trustworthy data analytics in the cloud using sgx. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[27] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[28] A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, pages 272–, 2004.

[29] D. Shen, Z. Zhang, X. Ding, Z. Li, and R. Deng. H-binder: A hardened binder framework on android systems. In *Proceedings of SecureComm*, 2016.

[30] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Ieee/ifip International Conference on Dependable Systems and Networks*, pages 367–378, 2015.

[31] Trusted Computing Group. TPM main specification. Main Specification Version 1.2 rev. 85, Feb. 2005.

[32] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[33] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, S&P, May 2012.