PROXIMITEE: Hardened SGX Attestation and Trusted Path through Proximity Verification

Aritra Dhar
ETH Zurich
aritra.dhar@inf.ethz.ch

Ivan Puddu
ETH Zurich
ivan.puddu@inf.ethz.ch

Kari Kostiainen
ETH Zurich
kari,kostiainen@inf.ethz.ch

Srdjan Capkun
ETH Zurich
srdjan.capkun@inf.ethz.ch

***Abstract*—Intel's Software Guard Extensions (SGX) enables isolated execution environments, called enclaves, on untrusted operating systems (OS), and thus it can improve the security for various applications and online services. However, SGX has also well-known limitations. First, its remote attestation mechanism is vulnerable to relay attacks that allow the attacker to redirect attestation and the following provisioning of secrets to an unintended platform. Second, attestation keys have been shown to leak thus enabling attackers to fake the secure attested environment by emulating it. Third, there exists no secure way to let enclaves communicate with the I/O devices and as a consequence the user.**

To address these shortcomings, we propose a hardened variant of SGX attestation using proximity verification. We design and implement a system called PROXIMITEE, where a simple embedded device with a low TCB is attached to the target platform. The embedded device verifies the proximity of the attested enclave by using distance bounding and secure boot-time initialization, thus allowing secure attestation regardless of a compromised OS or leaked attestation keys. Our boot-time initialization can be seen as a novel variant of "trust on first use" (TOFU) that makes deployment of secure attestation easier, reduces the system's attack surface and enables secure revocation. We further leverage the embedded device to build a trusted I/O path between peripherals (e.g., keyboards, displays) and enclaves, by letting it securely mediate every I/O communication between them. Our prototype implementation shows that such proximity verification is reliable in practice.

I. INTRODUCTION

Trusted execution environments like Intel's SGX [6] enable secure applications and services on untrusted computing platforms. In essence, SGX isolates small security-critical applications, called *enclaves*, from all the other software running on the same platform, including the privileged Operating System (OS). The primary security guarantees of SGX are enclave's data confidentiality and code integrity. Although recent research has demonstrated that information may leak from enclave's execution through digital side-channels [11], [20], [27], or by leveraging platform vulnerabilities [13], [15], correctly implemented enclaves on timely patched platforms remain a valuable security enabler for many applications and services.

Limitations of SGX. Remote attestation is a mechanism that allows a remote verifier to ensure that an enclave was constructed as expected. When an enclave is created, the CPU measures its code and securely stores the measurement. During remote attestation, the CPU signs the measurement using a processor-specific key. The signed attestation statement can be bound to the enclave's public key which allows the remote verifier to establish a secure connection to the enclave.

While remote attestation guarantees that the attested enclave runs the expected code, it *does not* guarantee that the enclave runs on the expected computing platform. The untrusted OS can *relay* incoming attestation requests to another platform, as was previously observed in the context of TPM attestation [29]. Such relay attacks weaken the security guarantees of SGX in many deployment scenarios. For example, servers at data centers are typically well maintained (e.g., regularly patched) and physically guarded. If the attestation is redirected to another SGX platform that is outside the data center and in the physical control of the adversary, he has increased capabilities to attack the user's data, including physical attacks and software attacks that leverage vulnerable platform versions.

In a more severe attack, an adversary that has obtained leaked, but not yet revoked, attestation keys can *emulate* an SGX-enabled CPU on any platform. The emulated enclaves would successfully pass remote attestation, thus making it impossible for the remote verifier to distinguish between real and emulated enclaves. Emulation attacks allow the attacker to obtain *any secrets* provisioned to the attested enclave and control its execution, hence breaking the two primary security guarantees of SGX: data confidentiality and code integrity. The extraction of attestation keys could be achieved through physical attacks or, as recently demonstrated, by exploiting platform vulnerabilities [13]. Emulation attacks are particularly concerning because with just one or few leaked keys the adversary can emulate enclaves on many platforms.

A straightforward solution to emulation and relay attacks is to simply assume *Trust On First Use* (TOFU) [26]. That is, at the time of attestation the remote verifier assumes that the target OS is not compromised and it routes the communication to the legitimate enclave, with which a secret can be shared to securely. Unfortunately, OS compromise is common. A commonly suggested and more secure TOFU variant is to perform the attestation immediately after a fresh installation of the OS. However, such solution has significant drawbacks. In many application scenarios, reinstallation of the OS is impractical. For instance, service providers like banks cannot force their customers to reset their computers for secure attestation. Even if the platform owner, like a cloud platform provider, can reset servers at will, this approach is limited to enclaves that are known at the time of platform initialization. In Section III, we review such limitations in more detail.

Another well-known limitation of SGX is its lack of *trusted path*, that is, a secure communication channel between a user and an enclave. Since in SGX the untrusted OS mediates all enclave's communication with the I/O devices, e.g., keyboard

and display, the OS can read or modify any messages sent between the enclave and the user. The lack of trusted path prevents secure implementation of many useful applications that require, e.g., password input from the user to the enclave.

Prior research has proposed to overcome this problem by using a trusted hypervisor that handles all I/O operations [35]. The main drawback of this approach is that general-purpose hypervisors have a significant Trusted Computing Base (TCB) size and thus expose a large runtime attack surface.

Our solution. In this paper, we propose a system called PROXIMITEE that uses a small trusted embedded device and *proximity verification* for more secure and easy-to-deploy remote attestation. An additional benefit of our approach is that it enables automated platform revocation. We design and implement two variants of our solution.

Our first variant leverages distance bounding [10] to prevent relay attacks. During attestation, the remote verifier establishes a secure connection to the embedded device which performs the standard SGX attestation and verifies the proximity of the attested enclave. After this, the remote verifier can establish a secure connection to the enclave. The connection is mediated by the attached device that performs periodic distance bounding measurements and the channel stays active only as long as the device is connected to the same platform.

Proximity verification alone cannot address emulation attacks, as a perfect locally emulated enclave would pass any proximity test. Therefore, our second variant uses a combination of secure *boot-time initialization* and distance bounding. The target platform loads a small, single-purpose kernel from the attached embedded device and launches an enclave that seals a secret key known by the embedded device. After this initialization, the target platform can reboot onto regular OS. Subsequently, when attestation is needed, the enclave can verify the proximity of other enclaves on the same platform using SGX's local attestation. As above, using distance bounding, the embedded device guarantees that the attested connection stays alive only as long as the device is connected. Our second variant can be seen as a novel variant of trust on first use and it protects against relay and emulation attacks, and thus enables secure attestation regardless of a potentially compromised OS or leaked attestation keys.

Similar to second-factor authentication tokens, our embedded device acts as an authenticator of the platform that is being attested. The physical act of attaching the device enables secure attestation (enrollment) while detaching the device will prevent further communication with the attested enclave (revocation). Neither enrollment nor revocation require interaction with a trusted authority, and thus through the use of periodic proximity verification, our solution enables *secure offline enrollment and revocation*.

We also extend our approach to a trusted path mechanism, where the trusted embedded device can be attached as a *bridge* between I/O devices and the target platform. The embedded device verifies the proximity of the enclave using either of our two variants and securely mediates user input and output to and from them.

Main results. We implemented complete prototype of our solution using an *Arduino Due* microcontroller prototyping board. Our prototype shows that the TCB of such device can be made small: our implementation is 3.9 KLoC.

To evaluate the security of our proximity verification scheme, we simulated a powerful relay-attack adversary that can perform the required protocol computation instantly and is connected to the target platform with a fast and short network connection (one meter Ethernet cable). Our experiments and analysis show that against such adversary proximity verification can be made secure, efficient and reliable at the same time. The adversary's probability of performing a successful relay attack is negligible (1.24×10^{-54}), while legitimate verification succeeds with a very high probability (0.9999998). Importantly, the adversary cannot increase his success probability with repeated attempts, as attestation is triggered by the benign remote verifier and proximity verification performed by the trusted embedded device. In terms of efficiency, the initial proximity verification adds only a minor delay of 25 ms to the attestation protocol and the periodic proximity verification consumes only 0.26% of the available channel capacity between the embedded device and the enclave.

In comparison to previous TOFU solutions, our approach has noteworthy benefits. The first is easier deployment, as the target platform does not need to be reinstalled or manually configured. The second is increased security, as in our solution only a simple embedded device needs to be trusted instead of a regular OS kernel. The third benefit is improved platform management, as previously enrolled platforms can be easily and securely revoked without interaction with trusted authorities. In comparison to hypervisor-based trusted path solutions, we provide a lower attack surface.

Contributions. To summarize, in this paper we make the following contributions:

1. *New approach.* We propose a novel approach for hardened SGX remote attestation based on a simple trusted embedded device and proximity verification.
2. *Two attestation variants.* We design and implement two attestation schemes. Our first solution uses distance bounding to prevent relay attacks. Our second solution combines secure boot-time initialization with proximity verification to prevent emulation attacks that use leaked SGX attestation keys.
3. *Trusted path.* We extend our attestation mechanisms to build a trusted I/O path, where the trusted embedded device acts as a bridge between peripherals and enclaves.
4. *Evaluation.* We built prototypes of our solutions using a microcontroller prototyping board. We demonstrate that our hardened attestation is secure and reliable with small overhead.

Outline. The rest of this paper is structured as follows. Section II provides background on SGX. Section III explains remote attestation attacks and defines adversary model. Section IV outlines our approach and describes two attestation variants. In Section V we analyze the security of our solutions. Section VI extends our approach for a trusted path. Section VII describes our implementation and evaluation. Section VIII reviews related work and Section IX concludes the paper.

II. SGX BACKGROUND

Intel's SGX isolates application enclaves from all other software running on the system, including the privileged

OS [16]. Enclave's data resides in plain-text inside the CPU but is encrypted and integrity protected whenever it is moved outside the CPU chip (e.g., to DRAM). Although untrusted, the OS is responsible for the enclave creation. Initialization actions taken by the OS to start enclaves are recorded securely inside the CPU, essentially creating a *measurement* that captures the enclave's code. The enclave measurement is used later as part of the attestation process. Furthermore, enclaves have the ability to *seal* data to disk, which essentially allows them to securely store confidential data into non-volatile memories with the guarantee that only the same enclave running in the same CPU will be able to retrieve it later. Enclaves cannot directly execute system calls, therefore whenever these are required developers must carefully design their applications into two logical parts. Protected processing takes place within the enclave part, and an unprotected part (normal user-level process) handles non-sensitive operations such as file system access and I/O through the OS.

Local attestation. SGX allows one enclave to authenticate another enclave on the same platform [7], [8]. An enclave can ask the CPU to generate a report data structure, which includes the enclave's measurement and a cryptographic proof that the enclave exists on the platform. This report can be given to another enclave who can verify that the enclave report was generated on the same platform. The authentication mechanism uses a symmetric key system where only the enclave verifying the report and the enclave creating the report know the key.

Remote attestation. Remote attestation is a procedure, where an external verifier checks that certain enclave code is correctly initialized. Attestation is an interactive protocol between three parties: (i) the remote verifier, (ii) the attested SGX platform, and (iii) Intel Attestation Service (IAS), an online service operated by Intel. During manufacturing each SGX processor is equipped with a unique attestation key that IAS uses for attestation verification. Each SGX platform includes a system service called Quoting Enclave that has exclusive access to this key. In attestation, the remote verifier sends a random challenge to the attested platform that returns a QUOTE structure (capturing the enclave's measurement from its creation) signed by the attestation key which can be forwarded to the IAS. The IAS then verifies the signed QUOTE, checks that the attestation key has not been revoked, and in case of successful attestation signs the QUOTE.

The attestation key is a part of a group signature scheme called EPID (Enhanced Privacy ID) [14] that supports two signature modes. The default mode is privacy-preserving and it does not uniquely identify the processor to IAS; the signature only identifies a group like certain processor manufacturing batch. The linkable signature mode allows IAS to verify if the currently attested CPU is the same as previously attested CPU. If a *linkable* mode of attestation is used, IAS reports the same *pseudonym* every time the same service provider requests attestation of the same CPU [6].

Vulnerabilities. Recent research has demonstrated that the SGX architecture can be vulnerable to side-channel leakage. Secret-dependent data and code access patterns can be observed by monitoring shared physical resources such as CPU caches [11], [20], [27] or the branch prediction unit [23]. The OS can also infer enclave's execution control flow or data accesses by monitoring page fault events [36]. Many such attacks can be addressed by hardening the enclave's code, e.g., using cryptographic implementations where the data or code access patterns are independent of the key.

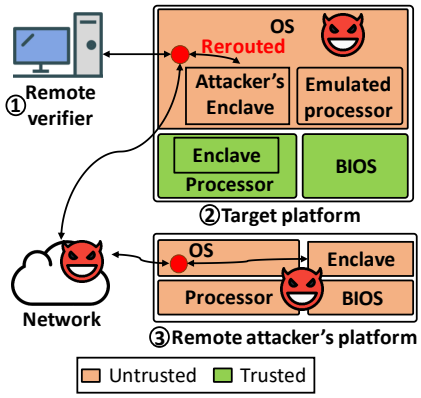

Fig. 1: **Attestation model and attacks.** The attacker controls the OS of the target platform and can relay attestation requests to another platform or emulate SGX processor on the target platform.

The recently discovered vulnerabilities Spectre [22] and Meltdown [24] allow application-level code to read memory content of privileged processes across separation boundaries by exploiting subtle side-effects of speculative execution. The Meltdown attack has been already adapted for attacking SGX enclaves [15]. The Foreshadow attack [13] demonstrates how to extract SGX attestation keys from processors by leveraging the Meltdown vulnerability.

III. Problem Statement

In this section, we specify a model for remote attestation, review attestation attacks, explain the lack of trusted path and outline limitations of known approaches.

A. Remote Attestation Model and Attacks

We consider a remote attestation model shown in Figure 1 that consists of three parties:

① *Remote verifier* is connected to the attested target platform either over network channel or local interface. We consider the remote verifier trusted.

② *Target platform* has a compromised OS that the attacker controls fully. The processor and its SGX protections are trusted, that is, we assume that the adversary cannot extract attestation or sealing keys from the target platform. We also assume that the BIOS (or UEFI) on the target is trusted.[1]. The adversary may have leaked attestation or sealing keys from *other* SGX processors and therefore is able to emulate another SGX processor. The attacker has no physical access to the target platform. The target platform is connected to a network that is controlled by the adversary.

③ *Attacker's platform.* The adversary has another SGX processors. All privileged software in this platforms is in

[1]This condition is not necessary for our first attestation variant presented in Section IV-D

permanent control of the adversary, and the adversary has physical access to the platform. The adversary may be able to extract the attestation and sealing keys from this CPU.

Relay attack. In a relay attack, the adversary reroutes attestation requests to his own platform, as illustrated in Figure 1. Anonymous attestation (cf. Section II) prevents identification of the physical platform from the signed attestation response received from the IAS server. Such lack of TEE identification is a long-standing open problem in the trusted computing literature [29]. SGX's linkable attestation mode allows the remote verifier to link two attestations of the same platform, but it does not prevent relay attacks during the first attestation.

Even if the adversary has not fully compromised the SGX protections on his own platform (i.e., managed to extract encryption and signing keys), relay attacks have undesirable implications including the fact that the adversary has an indefinite amount of time to execute physical or software-based side-channel attacks. Also, if platform vulnerabilities similar to Spectre and Meltdown [22], [24] are found *after* the attestation, the adversary can attack the attested enclave, and any secrets provisioned to it, through the vulnerability.

Emulation attack. The second attack that we consider applies to adversaries that have obtained at least one valid (i.e., not revoked by Intel) attestation key from other platforms. In this attack, the adversary emulates an SGX-processor on the attestation target platform using the leaked attestation key from another CPU (see Figure 1). Since the IAS successfully attests the emulated enclave, it is impossible for a user to distinguish between the emulated enclave and the real one. This allows the adversary to fully control the attested execution environment which means she can obtain any secrets that are provisioned to the emulated enclave and arbitrarily modify its execution. Emulation attacks break the two fundamental security guarantees of SGX: enclave's data confidentiality and code integrity.

B. Limitations of Known Attestation Solutions

To address such attacks on attestation, a common approach in recent research papers and systems is to assume some form of *Trust On First Use* (TOFU). For example, ROTE [25] assumes that the OS is installed in isolation before the start of the protocol; VC3 [32] uses a special-purpose enclave, Cloud QE (quoting enclave), in conjunction of SGX QE that generates a public/private key pair, outputs the public key and seals the private key which never leaves the Cloud QE before a new platform enters into service; SCONE [9] assumes a trusted OS at the time the Linux container is deployed; and SecureKeeper [12] assumes trust on first use while deploying the entry enclave that maintains client connections and encrypts all the messages between the clients and the enclave.

All of these approaches have noteworthy limitations that we discuss next by using a simple scheme as a straw-man solution. In our straw-man solution, a fresh OS installation is performed on the target platform to make sure that at the time of initialization and attestation there is no code injected by the attacker which would enable relay attacks or emulation. While protection against relay attacks could be obtained by physically detaching all the network interfaces,[2] emulation prevention can only be achieved if the OS is in a known, non-compromised state (e.g., clean installation). Once the target platform is started after installation, the platform's owner inputs a credential (e.g., a key certified by a trusted CA) that allows the remote verifier to authenticate the platform and perform remote attestation on it. The enclaves that require attestation can seal the credential for later use. Such solution has the following security and deployment limitations:

1. *OS reinstallation:* The increased security comes at the expense of practicality and ease of deployment, as the target platform OS needs to be reinstalled. For instance, service providers like banks cannot force their customer to reinstall the OS for adoption of more secure attestation.
2. *Manual configuration:* To enable secure attestation, a platform-specific credential needs to be provisioned to the target platform. Such manual interaction complicates platform enrollment, especially in scenarios like data centers where the number of enrolled platforms can be high.
3. *Pre-defined enclaves:* This approach only works for enclaves that are known and installed at the time of installation, as they need to securely seal the provided credential. However, in many scenarios, such as in cloud computing platforms, users need to install new enclaves after platform installation.
4. *Large temporary TCB:* Modern operating systems in which SGX enclaves run have a large TCB. Although the OS is exposed to attacks only for a short period of time (the first use), a large TCB is nonetheless undesirable.
5. *Online CA for revocation:* If the target platform needs to be revoked at a later point in time, interaction with the CA is again required. Therefore, the CA cannot be fully offline to reduce its attack surface.

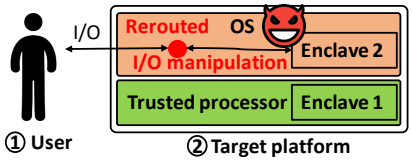

Fig. 2: **Trusted path model and attacks.** The OS can see and manipulate all the I/O operations done by the user. Additionally, the attacker can reroute user input to another enclave than the user intended.

C. Lack of Trusted Path

Another limitation of SGX is the lack of *trusted path*. As defined in [18], a trusted path (i) isolates the input and output channels of different applications to preserve the integrity and confidentiality of data exchanged with the user, (ii) assures the user of a computer system that she is truly interacting with the intended software, and (iii) assures the running applications that user inputs truly originate from the actions of a human (as opposed to being injected by other software).

As shown in Figure 2, an adversary which controls the OS can trivially read and modify all user's inputs, read and modify all enclave's outputs intended to the user, and direct user's inputs to a different enclave from the one intended by the user. Under the SGX security model, lack of trusted path prevents

[2]If the platform has a wireless interface, it could be placed in a Faraday cage to make sure that communication is disabled.

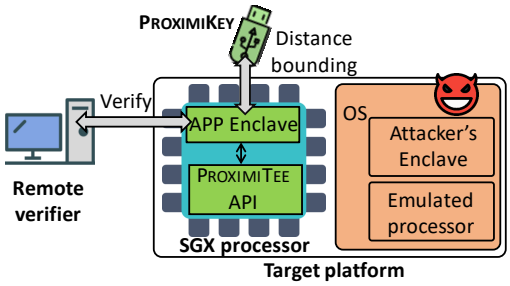

Fig. 3: **Approach overview.** A trusted embedded device ProximiKey is attached to the target platform, verifies the proximity of the attested enclave, and enables a secure connection to it.

the user from providing sensitive information like passwords to enclaves or confirming transactions performed by the enclave.

Limitations of known solutions. The commonly suggested solution for the trusted path is to leverage a trusted hypervisor to mediate all I/O [35]. The main drawback of general-purpose (commercial) hypervisors is their significant complexity and attack surface. While the research community has produced also small and formally-verified hypervisors, like the seL4 project [33], their adoption in practice can be problematic. In addition to the secure hypervisor itself, realization of a trusted path requires secure device drivers which can be difficult to implement and increase the TCB size. Formally-verified hypervisors are also typically severely restricted in terms of functionality and adding new functionality to can be very slow, as each new update needs to be formally verified (a process that can take years). For these reasons, minimal and formally-verified hypervisors are not commonly used in consumer devices or corporate systems that require rich functionality and updates.

D. Goals

Our main goal is to enable remote attestation and trusted path in a way that is more secure than current solutions and practical to deploy at the same time.

1. *Security.* Our attestation solution should address adversaries that control the OS and have leaked attestation keys. Our solution should not increase TCB of enclaves significantly (recall that in SGX's security model only the CPU is trusted besides the enclave's code) and it should minimize interaction with trusted authorities like CAs. Our trusted path mechanisms should have smaller attack surface than current hypervisor-based solutions.

2. *Deployment.* We focus on target platforms like consumer devices and corporate servers that should provide rich functionality and frequent updates similar to commercial off-the-shelf operating systems. We want to avoid solutions where the target platform owner has to reinstall the OS or perform manual configuration with each enrolled computing platform.

IV. ProximiTEE Attestation

In this section, we describe our solution ProximiTEE for hardened remote attestation. We start with an overview, explain our security assumptions, then outline example use cases, and finally present our two attestation variants.

A. Approach Overview

We propose a hardened SGX attestation scheme based on the intuitive idea of *enclave proximity verification*.[3] The overview of our approach is shown in Figure 3. Our solution utilizes the standard SGX remote attestation as a primitive, while enhancing it by performing proximity verification through a *trusted embedded device* (ProximiKey) that is physically attached to the target platform, e.g., over a USB interface. ProximiKey then facilitates the creation of a secure channel (e.g., `TLS`) between the remote verifier and the attested enclave. The operating system of the ProximiKey-connected platform relays all the data to and from the ProximiKey.

After the initial attestation, ProximiKey *periodically checks proximity* to the attested enclave. Thus, the established secure channel is contingent on the physical presence of the embedded device on the target machine and it stays active only as long as the device is plugged-in. The physical act of detaching the device automatically revokes the attested platform without any interaction with a trusted authority. Thus, our solution enables *secure offline enrollment and revocation*.

To use our solution, enclave developers add function calls to a simple ProximiTEE API that facilitates communications between the enclave and the ProximiKey and executes the proximity verification protocol.

B. Security assumptions

The embedded device ProximiKey is a trusted component in our solution. We deem this choice reasonable since ProximiKey implements only the strictly necessary functions and therefore it a has significantly smaller software TCB, attack surface, and hardware complexity compared to the host operating system and hypervisor solutions like [35]. We assume that its issuer certifies each embedded device. This certification takes place prior to the ProximiKey deployment, and therefore it can happen entirely offline. ProximiKey has a public and private key hardcoded into it, and it can store the issuer's certificate for its public key.

Concerning the security of the ProximiKey device we employ the same adversary model introduced in Section III for the SGX enclaves. While the user's device and its private keys are never exposed to the attacker, another ProximiKey can be in the physical possession of the attacker, which has as much time as she wants to fully compromise it (that is, for instance, get it to run arbitrary code, and extract its keys).

C. Example Use Cases

Figure 4 shows three example use cases for our solution.

① *User identification in online services.* In the first use case, we consider a remote verifier such as a bank or a company. The remote verifier issues a certified ProximiKey to the owner of the attested target platform, such as a customer

[3] We note that the idea of proximity verification for remote attestation has been proposed in the previous literature in the context of TPM attestation (see, e.g., [19], [34]), but never realized as a fully-functional system. In Section VIII we review these previous works in more detail.

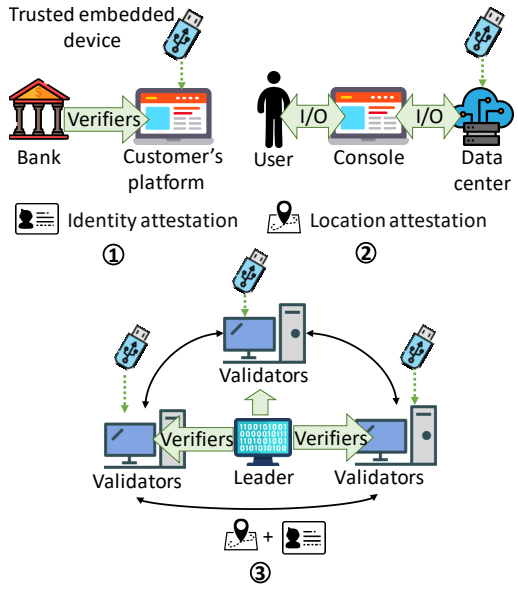

Fig. 4: **Use cases and types of attestation.** ① Remote verifier such as bank checks if the platform from where the customer logs in is connected to the bank-issued embedded device. ② The user establishes a secure path to a TEE on the remote server like data center. ③ Group leader verifies the TEEs for permissioned blockchain validator nodes.

of a bank[4] or an employee of a company. We call this type of attestation *identity-based attestation* as it enables the remote verifier to verify the owner of the attested platform. Binding the user's identity to a computing platform can be realized by plugging in the trusted embedded device to the platform. Our approach guarantees the attested user can communicate with the bank if and only if the PROXIMIKEY is physically attached to the user's platform. If the user switches to a new computing platform, the act of moving PROXIMIKEY to the new platform automatically revokes the old platform.

② *Outsourcing data to remote infrastructures.* In the second use case, the issuer of the trusted embedded device also physically enforces the location in which it can be deployed. For instance, a cloud platform provider can attach PROXIMIKEY to a server in a specific data center and publish the public key of the attached device to the users of the service. Such attestation enables the user to ensure that he is outsourcing data and computation to a server that resides in a specified location, thus providing *location-based attestation*. Similar to example ①, the enrollment of a platform can be done by just attaching the PROXIMIKEY to the platform and enforcing that PROXIMIKEY never leaves the facility of the location to which it is bound. Revocation (e.g., when a server is relocated to another data center or function) can be realized by merely detaching the PROXIMIKEY.

③ *Initialization of permissioned blockchains.* As our third example we consider a trusted authority that initializes a set of validators for a permissioned, SGX-hardened blockchain. The trusted authority issues one PROXIMIKEY

[4]Note that it is common for banks and to distribute hardware tokens to account holders, and by authenticating the user's input this new device could replace existing solutions.

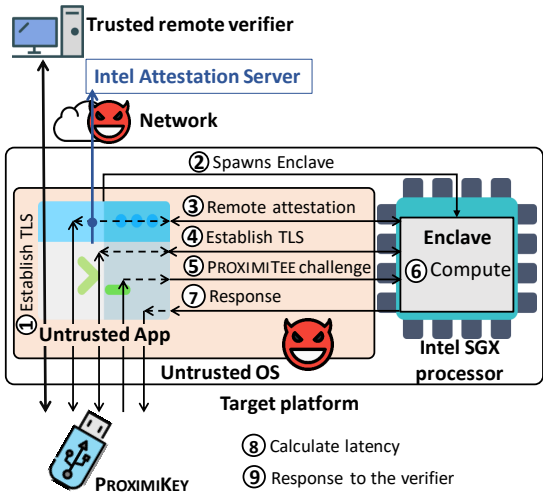

Fig. 5: **Attestation variant I: distance bounding.** The remote verifier establishes a secure channel to the PROXIMIKEY embedded device that performs standard attestation and verifies the proximity of the attested using distance bounding.

for each organization that operates one of the validator nodes which allows secure attestation of the validator platforms. Organizations are free to upgrade their computing platforms by attaching the PROXIMIKEY to a new platform which automatically revokes the old platform without the need to interact with the trusted authority. Furthermore, since PROXIMIKEY can only be active on one platform at the time, such a deployment enables the trusted authority to bound the "voting power" (e.g., identities in Byzantine consensus) of each validator organization in the blockchain consensus.

D. Attestation Variant I: Distance-Bounding

Next, we describe our first hardened attestation variant. The main idea behind this form of attestation is to use the trusted embedded device to perform both a standard remote attestation on the target enclave and then verify the proximity of the attested enclave using distance bounding. If both steps succeed, the embedded device enables the remote verifier to establish a secure connection to the enclave.

Attestation protocol. Figure 5 illustrates the attestation protocol that proceeds as follows:

① The remote verifier establishes a secure channel (e.g., TLS) to the certified PROXIMIKEY. An assisting but untrusted user-space application facilitates the connection on the target platform acting as a transport channel between the remote verifier and the PROXIMIKEY (and later also the enclave). As part of this first step, the remote verifier specifies which enclave should be executed.

② The untrusted application creates and starts the attestation target enclave.

③ PROXIMIKEY performs the standard remote attestation protocol to verify the code configuration of the enclave with the help of the IAS server (see Section II for attestation protocol details). In the attestation protocol, the device learns the public key of the attested enclave.

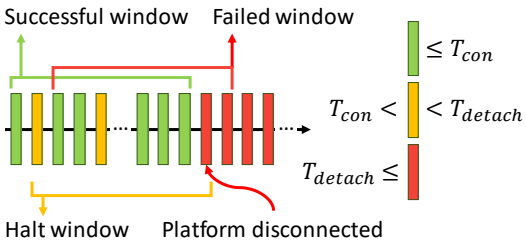

Fig. 6: **Sliding window.** The figure shows an example of sliding windows for periodic proximity verification.

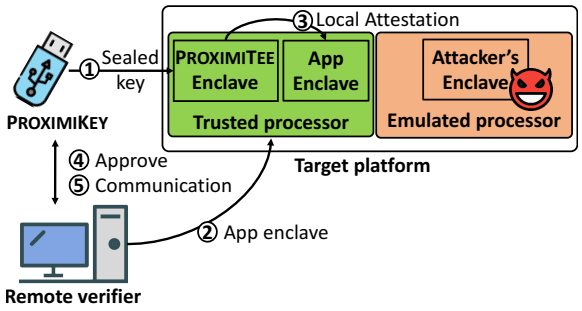

Fig. 7: **Attestation variant 2: boot-variant attestation.** After the boot-time initialization (refer to Figure 8) the PROXIMITEE enclave executes a local attestation with the verifier uploaded application-specific enclave.

④ PROXIMIKEY establishes a secure channel (e.g., TLS) to the enclave using that public key.

⑤ PROXIMIKEY performs a distance-bounding protocol that consists of n rounds, where each round is formed by steps ⑤ to ⑧. At the beginning of each round PROXIMIKEY generates a random challenge r and sends it to the enclave over the TLS channel.

⑥ The enclave increments the received challenge by one.

⑦ The enclave sends a response $(r + 1)$ back to the PROXIMIKEY over the TLS channel.

⑧ PROXIMIKEY verifies that the response value is as expected (i.e., $r+1$) and checks if the latency of the response is below a threshold (T_{con}). Successful proximity verification requires that the latency is below the threshold for a sufficient fraction (k, at least $k.n$ out of n) of responses.

⑨ If proximity verification is successful, the PROXIMIKEY notifies the remote verifier over the TLS channel (constructed in step ①). The verifier starts using the PROXIMIKEY TLS channel to send messages to the enclave.

Periodic proximity verification. After the initial connection establishment, PROXIMIKEY performs *periodic* proximity verification on the attested enclave. The PROXIMIKEY device sends a new random challenge r at frequency f, verifies the correctness of the received response and measures its latency. The latest w latencies are stored to a sliding window data structure, as shown in Figure 6.

As described in Section VII we generally observe three types of latencies in the presence of relay attacks. The first type of response is received faster than the threshold T_{con} (green in Figure 6), these responses can only be produced if no attack is taking place. In the second type of response the latency exceeds T_{con}, but it is below another, higher threshold T_{detach} (yellow), these are sometimes observed during legitimate connections and sometimes during relay attacks. And third, the latency is equal to or exceeds T_{detach} (red), these latencies are only observed while a relay attack is being performed. Given such a sliding window of periodic challenge-response latencies, we define the following rules for halting or terminating the connection:

1. *Successful window: no action.* If at least k responses have latency $\leq T_{con}$ and none of the response have latency $\geq T_{detach}$, we consider the current window *legitimate*. PROXIMIKEY keeps the established connection active, i.e., no action.

2. *Halt window: prevent communication.* If one of the responses have latency $\geq T_{detach}$, we consider the current window a "halt window" and the PROXIMIKEY stops forwarding further communication to the enclave until the current window is legitimate again.

3. *Failed window: terminate channel.* If two or more responses have latencies $\geq T_{detach}$, we consider the current window a "failed window" and the PROXIMIKEY terminates the communication and revokes the attested platform.

E. Attestation Variant II: Boot-Time Initialization

The usage of distance bounding prevents relay attacks. However, proximity verification alone cannot protect against processor emulation attack, as PROXIMIKEY cannot distinguish between the enclave running on the physical processor versus the enclave running on the emulated processor.

Now, we describe our second hardened attestation variant that is based on the idea of secure *boot-time initialization*. This solution can be seen as a novel variant of trust on first use that simplifies deployment (e.g., no OS reinstall) and increases security (e.g., reduced attack surface). Additionally, usage of the trusted device enables additional properties such as offline revocation.

Figure 7 illustrates an overview of this solution. During initialization, that is depicted in Figure 8, the target platform is booted from the attached PROXIMIKEY that loads a minimal kernel (PROXIMITEE kernel) on the target device. In particular, this kernel includes no network functionality. The kernel starts an enclave (PROXIMITEE enclave) that shares a secret with the device. This shared secret later bootstraps the secure communication between PROXIMIKEY and the PROXIMITEE enclave. The security of the bootstrapping relies on the fact that the minimal kernel will not perform enclave emulation at boot time. The PROXIMITEE enclave will later be used as a proxy to attest whether other (application-specific) enclaves in the system are real or emulated and on the same platform.

Boot-time initialization. The boot-time initialization process needs to be performed only once. This process is depicted in Figure 8 and it proceeds as follows:

① The platform owner plugs PROXIMIKEY to the target platform, restarts it to BIOS and selects the option to boot from PROXIMIKEY.

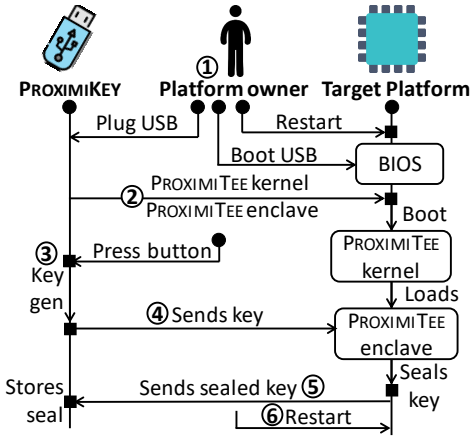

Fig. 8: **Boot-time initialization.** The ProximiKey uses a minimal kernel Linux image to boot and load ProximiTEE enclave on the target platform and seal a platform specific secret to the ProximiKey memory.

② ProximiKey loads the ProximiTEE kernel and boots from it. The ProximiTEE kernel starts the ProximiTEE enclave.

③ The user presses a button on ProximiKey to confirm that this is a boot-initialization process. This step is necessary to prevent an attack where the compromised operating system emulates a system boot.

④ ProximiKey sends a randomly generated key $\mathcal{K}$ to the ProximiTEE enclave.

⑤ The enclave returns the sealed key $\mathcal{S}$ corresponding to the key $\mathcal{K}$ ($\mathcal{S} \leftarrow \texttt{Seal}(\mathcal{K})$) to ProximiKey that stores the key and the seal pair $(\mathcal{K},\mathcal{S})$ on its flash storage.

⑥ ProximiKey blocks further initializations, sends a restart signal and boots the platform with the normal OS.

Attestation process. After initialization the target platform runs a regular OS. The attestation process is depicted in Figure 7 and proceeds as follows:

① ProximiKey sends the seal $\mathcal{S}$ to the ProximiTEE enclave that unseals it and retrieves the key $\mathcal{K}$. ProximiKey and the ProximiTEE enclave establish a secure channel (TLS) using $\mathcal{K}$.

② The remote verifier uploads a new application-specific enclave on the target platform.

③ The ProximiTEE enclave performs local attestation (cf. Section II) on the application-specific enclave that binds its public key to the attestation.

④ The ProximiTEE enclave sends the measurement and the public key of the application-specific enclave to ProximiKey. ProximiKey establishes a secure channel to the application-specific enclave and sends the measurement of the enclave to the remote verifier. The remote verifier then approves the communication to the application-specific enclave.

⑤ The remote verifier checks that the measurement of the application-specific enclave is as expected. If this is the case, it can communicate with the enclave through ProximiKey.

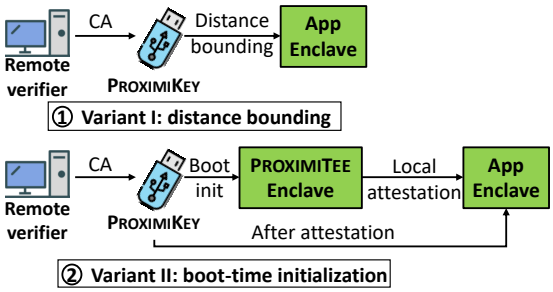

Fig. 9: **Security analysis overview.** Attestation is successful, if the remote verifier establishes a secure communication channel to the correct enclave.

Following communication. Similar to our previous variant, after the initial attestation all the communication between a remote verifier and the enclave is mediated by the ProximiKey that periodically checks the proximity of the attested enclave and terminates the communication channel in case the embedded device is detached.

V. Security Analysis

In this section, we provide an information security analysis. We analyze attestation security, revocation security, and attack surface, respectively. Full details on our experimental evaluation, including proximity verification parameters and system performance, are provided in Section VII.

A. Attestation Security

To analyze the security of the initial attestation, we must first define successful attestation. We say that the attestation is successful when the remote verifier establishes a connection to the "correct" enclave that (i) has the expected code measurement and (ii) runs on the computing platform to which ProximiKey is connected to.

Attestation variant I. In our first attestation variant, the task of establishing a secure channel to the correct enclave can be broken into two subtasks (see the top in Figure 9). The first subtask is to establish a secure channel to the correct ProximiKey device. In our solution, this is achieved using standard device certification. We assume that the adversary cannot compromise the specific ProximiKey used. If the adversary manages to extract keys from other ProximiKey devices, he cannot trick the remote verifier to connect to a wrong enclave, as the remote verifier will only communicate with a pre-defined embedded device.

The second subtask is to establish a secure connection from ProximiKey to the correct enclave. For this, we use proximity verification. ProximiKey verifies the proximity of the attested enclave through steps ⑤ to ⑧ of the protocol. These steps essentially check two things. First, through step ⑦, whether the messages are received from the correct enclave. This verification is performed by checking the correctness of the decrypted message, and it relies on the assumption that the attacker cannot break the underlying encryption and hence only the enclave that has access to the key that was bound to the attestation could have produced a valid reply. Second,

through step ⑧, whether the PROXIMIKEY and the enclave are in each other's proximity. This check relies on the assumption that a reply from a remote enclave will take more time to reach the PROXIMIKEY than a reply from the local enclave.

We evaluate the second aspect experimentally. In particular, we simulate a powerful relay-attack adversary that is connected to the target platform with short and fast network connection (e.g. a one meter long Ethernet cable). Since the attacker's platform might be faster than the target platform, we simulate an adversary that can perform instantly all the needed computation to participate in the proximity verification protocol.[5] We define adversary's success as the event where the proximity verification succeeds with an enclave that resides on the above-defined adversary's platform and denotes the probability of such event P_{adv}. We define legitimate success as the event that proximity verification succeeds with a local enclave and denote its probability P_{legit}.

In Section VII our experiments and analysis show that example parameter values $n = 50$, $k = 0.3$ and $T_{con} = 461\mu s$ enable reliable, secure and fast proximity verification. Proximity verification takes 25 ms, and therefore it adds only a minor delay to the initial attestation. The adversary's success probability is negligible: $P_{adv} = 1.24 \times 10^{-54}$. With the same parameters, legitimate proximity verification with a local enclave succeeds with high probability: $P_{legit} = 0.9999998$.

Since perfectly emulated SGX environment can pass any proximity test, this variant does not prevent emulation attacks enabled by leaked SGX attestation keys from other CPUs.

Attestation variant II. Our second variant also prevents emulation attacks. This variant relies on the integrity of the BIOS / UEFI to run once per platform the correct PROXIMITEE kernel which initializes the PROXIMITEE enclave.[6] The PROXIMITEE kernel is a single-purpose kernel that only supports a minimal set of features that is essential to run SGX which makes its attack surface small.

In this variant, the task of establishing a secure communication channel to the correct enclave is broken into three subtasks (see the bottom in Figure 9). The first subtask is the same as above.

The second subtask is to establish a secure communication channel from PROXIMIKEY to the PROXIMITEE enclave. For this, we use a secure boot-time initialization (i.e., trust on first use). PROXIMIKEY shares a key with an enclave that is started by the trusted PROXIMITEE kernel, hence at a time in which the attacker could not emulate any enclave. PROXIMIKEY knows when secure initialization takes place because the user (platform owner) indicates this by pressing a button which is an operation that the adversary cannot perform. The PROXIMITEE enclave seals the key during initialization. Different SGX CPUs cannot unseal each other's data, and therefore even if the adversary has extracted sealing keys from other SGX processors, she cannot unseal the key and masquerade as the legitimate PROXIMITEE enclave.

[5] A computationally-bounded attacker cannot break cryptographic primitives such as encryption, hash, and signature.

[6] There exist methods (e.g., [31]) to verify the integrity of the BIOS / UEFI in hardware, if for a particular scenario the administrator believes this prerequisite is not met.

The third subtask is to establish a secure communication channel from the PROXIMITEE enclave to the application-specific enclave. The security of this step relies on SGX's built-in local attestation functionality (cf. Section II). An adversary that has obtained leaked sealing attestation keys from other SGX processors, cannot produce a local attestation report that the PROXIMITEE enclave would accept, and therefore the adversary cannot trick the remote verifier to establish a secure communication channel to a wrong enclave.

B. Periodic Proximity Verification Security

To analyze the security of the periodic proximity verification that is used for platform revocation, we must first define what it means for the attacker to break the periodic proximity verification. The purpose of the periodic proximity verification is to prevent cases where the user detaches the PROXIMIKEY from the attested target platform and attaches it to another SGX platform before the previously established connection is terminated. Since we consider an adversary who does not have physical access to the target platform (recall Section III-A), we focus on benign users and exclude scenarios where the PROXIMIKEY would be connected to multiple SGX platforms with custom wiring or rapidly and repeatedly plugged in and out of two SGX platforms.

We define that adversary breaks the periodic proximity verification if the previously established connection is not terminated within a "short delay" after the PROXIMIKEY was detached from the attested target platform. For most practical purposes we consider an example delay of 1 ms sufficiently short. We denote the adversary's success probability in breaking the periodic proximity verification as P'_{adv}.

We define as false positive for periodic attestation the event where the connection to the legitimate enclave is terminated and the attested platform is revoked despite the PROXIMIKEY being connected to the platform. We denote the probability that this happens during a "long period" of usage as P'_{fp}. We consider an example period of 10 years sufficiently long for most practical usages.

In Section VII evaluate this experimentally, again by simulating a similar relay-attack adversary. We show that there exists at least a set of parameters that provide secure, reliable and inexpensive channel termination and platform revocation. The attacker's success probability can be made negligible: $P'_{avd} = 1.6 \times 10^{-50}$, while keeping the false positive probability low: $P'_{fp} = 1.12 \times 10^{-8}$. Such periodic proximity verification consumes only 0.0011% of the available channel capacity (USB 2.0 has a channel capacity of 480 MBits/s) between PROXIMIKEY and the enclave, so we consider its cost minor.

C. TCB and Attack Surface

Figure 10 illustrates a comparison of trusted components and attack surface between our straw-man solution (cf. Section III-B) and our hardened attestation variants. In the straw-man solution, the Trusted Computing Base consists of the target platform hardware, a regular operating system that needs to be trusted only at the time of first use, and a standard certification authority (CA). In our Variant I (distance-bounding), the TCB contains the target platform hardware, the embedded device (PROXIMIKEY), and a CA

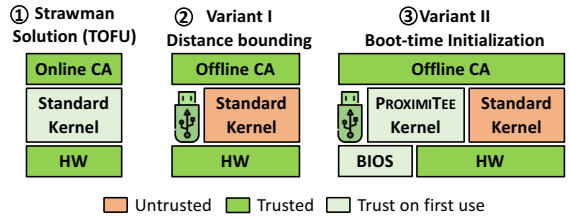

Fig. 10: **TCB comparison.** We illustrate the components that need to be trusted in our straw-man solution and two hardened attestation variants.

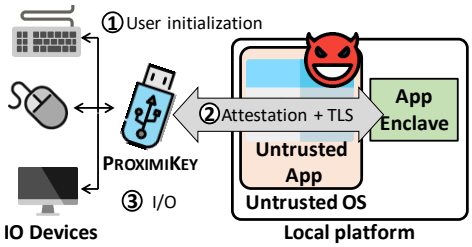

Fig. 11: **Trusted path to local enclave.** The IO devices are connected to ProximiKey that is connected to the local platform. The ProximiKey performs attestation using one of our variants and then mediates all IO communication.

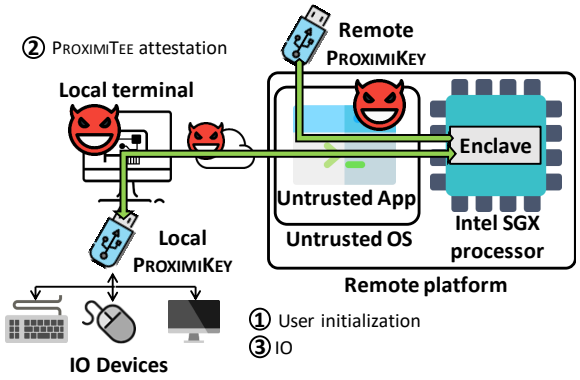

Fig. 12: **Trusted path to remote enclave.** This setup uses two embedded devices. The local ProximiKey is connected to the local platform and the remote ProximiKey is connected with the remote platform.

that can be fully offline. The OS is untrusted. In our Variant II (boot-time initialization), the TCB contains the target platform hardware, the embedded device, an offline CA and a small kernel that needs to be trusted at the time of first use. This small kernel requires no network functionality, and thus this component can be considered offline.

In Section VII we show that the complexity of our prototype implementation is small (3.9 KLoC) and thus both of our variants can provide significantly reduced TCB and attack surface over typical trust on first use solutions.

VI. ProximiTEE Trusted Path

In this section, we extend our attestation techniques to build a trusted path between the user and an enclave. Our main idea is to use the ProximiKey trusted embedded device as a *bridge* to securely mediate all user inputs and outputs between I/O devices and enclaves.

For trusted path we require that the embedded device has at least two communication interfaces, one for the target platform and additional ones for the I/O device(s), and minimal user interaction capabilities, e.g., a small display and a button. We also assume that the embedded device is either (i) pre-installed with a list of human-readable names for enclave code measurements, or (ii) it can fetch such mappings from a trusted server, similar to property-based attestation [30].

We consider the following mode of user interaction. The user must explicitly activate the trusted path and select the enclave with which she wishes to communicate. This can be done, for example, using the button and enclave names shown on the device screen. Alternatively, the interaction can be initiated by an untrusted application or the OS. In this case, the embedded device can show the human-readable enclave name on its screen that the user can verify.

Trusted path to local enclave. Now we describe the process of establishing a trusted path to an enclave on a local platform. As shown in Figure 11, the I/O devices are connected to the ProximiKey that is attached to a local computing platform. The trusted path creation proceeds as follows:

① The user activates the trusted path. The user selects which enclave to use using a button and display on ProximiKey.

② ProximiKey performs attestation of the chosen enclave using either of our two attestation variants. ProximiKey verifies that the measurement of the attested enclave matches the user's selection. ProximiKey establishes a secure channel (TLS) to the correct enclave.

③ ProximiKey captures all the input from the I/O devices and sends them to the enclave via the secure channel. Similarly, the enclave can send output to the user over the same channel.

Trusted path to remote enclave. Next, we describe how such trusted path can be extended to an enclave that resides on a remote platform from a local and untrusted platform. Figure 12 illustrates this scenario. Both the local and the remote platform have a ProximiKey device attached to them. The I/O devices are attached to the local ProximiKey. Trusted path creation proceeds as shown in Figure 12:

① The user initiates the trusted path by selecting an enclave as explained above.

② The local ProximiKey acts as the remote verifier in remote attestation using using one of our attestation variants. As the end result of the attestation process, the local ProximiKey has established a secure channel to the correct enclave via the remote ProximiKey.

③ The user can securely communicate with the enclave.

VII. Evaluation

In this section, we describe our prototype implementation and experiments. Then we analyze suitable parameters for proximity verification.

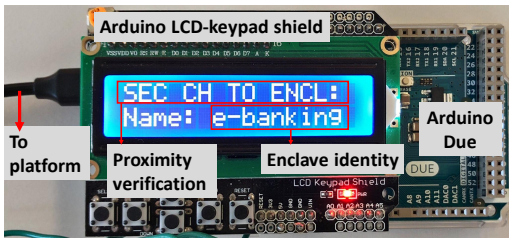

Fig. 13: **PROXIMIKEY prototype.** Our prototype is build on an Arduino Due prototyping board with LCD-keypad for indicating currently active enclave.

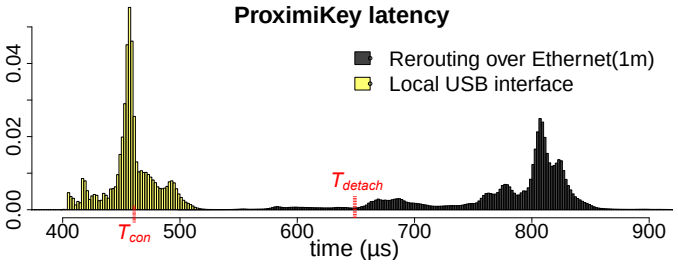

Fig. 14: **Challenge-response latency.** Latency frequency distributions when PROXIMIKEY prototype is connected to the target platform over the USB interface and for the simulated relay attack over a short Ethernet connection. T_{con} and T_{detach} thresholds are placed at $461\mu s$ and $649\mu s$, respectively.

A. Implementation

We implemented a complete prototype of the PROXIMITEE system. Our implementation consists of three primary components: (i) PROXIMIKEY prototype, (ii) PROXIMITEE enclave API which enables any application-specific enclave to communicate with the PROXIMIKEY and execute the PROXIMITEE protocol, and (iii) PROXIMITEE kernel prototype.

PROXIMIKEY. Our PROXIMIKEY prototype consists of one Arduino Due prototyping board equipped with an 84 MHz ARM Cortex-M3 microcontroller, as shown in Figure 13. The board communicated with the target platform over native USB 2.0 connection that provides high-speed 480 Mbps connection.[7]. We use the Arduino cryptographic library [2] for the TLS. The limited set of cipher suites in our implementation uses 128-bit AES (CTR mode) for encryption, AES-HMAC for message authentication code, Curve25519 for Diffie-Hellman for key exchange and SHA256 for the hash function. Our prototype implementation is approximately 200 lines of code, and the code size of the TLS and LCD controller is around 3.9 KLoC.

PROXIMITEE kernel. We have modified an image of Tiny Core Linux [3], and used it as the boot image for our attestation variant II (cf. Section IV-E). The image size of our modified Linux distribution is 14 MB (in contrast to 2 GB standard 64 bit Linux images build on the standard kernel). Our image supports bare minimum functionality and includes `libusb`, `gcc`, Intel SGX SDK, Intel SGX platform software (PSW), and Intel SGX Linux driver.

PROXIMITEE enclave API. The PROXIMITEE API for application-specific enclaves is written in C++ using the Intel SGX API. The API uses native SGX crypto library for TLS implementation. The prototype is around 200 lines of code.

B. Experimental Setup and Simulated Relay Attack

We conducted our experiments on three SGX platforms: two Intel NUC NUC6i7KYK mini-PCs and one Dell Latitude laptop, all equipped with SGX-enabled Skylake core i7 processors and Ubuntu 16.04 LTS installed on them.

We performed two types of experiments. First, we tested legitimate attestation of an enclave on the target platform (i.e., SGX platform to which our PROXIMIKEY prototype device was connected to). Second, we *simulated* a relay-attack adversary whose platform was connected to the target platform via a direct Ethernet cable. Since the adversary might have a faster processor than the one of the target platform, we simulate the worst case scenario, where the adversary's computation needed for the proximity verification protocol happens instantly. Instant replies were simulated by fixing the randomness for the challenges and having precomputed responses for that randomness on the attacker's machine. While on the one hand, we tried to make the adversary as powerful as possible, by placing her as close as possible to the victim and simulating high computational resources, we are aware that the OS kernel network stack can be optimized for network latency, and since the attacker has control over the OS, it would be in her interest to optimize it for minimum latency. However, given the space of parameters to play with, and since optimizing for latency is not trivial, we report results for a default Linux kernel configuration. Nevertheless, when possible we also present results for simulated minimum point-to-point latencies.

For the relay-attack case, we tested three different Ethernet cables of length 1m, 7m and 10m to evaluate the effect on the latency. For the legitimate proximity verification, we used a standard USB cable of length 1m and a USB extender cable or length 2m to evaluate the effect of USB cable length on the latency. We also tested two different SGX platforms and two different embedded device prototypes.

To measure latencies we used Arduino's native `micros()` that provides (10s of) microsecond level accuracy. To achieve more accurate time measurements, we also used a high precision 8 Ghz Keysight Infinium oscilloscope. We performed a total of 22 million rounds of the protocol for normal attestations and 15 million rounds for simulated attacks and measured the challenge-response latencies for each. We measure all of them inside the Arduino code. For cross-validation, we tested the PROXIMIKEY with the high precision oscilloscope and witnessed identical timing patterns.

C. Main Experimental Result

Figure 14 shows our main experimental result for the 1 meter Ethernet and 1 meter USB cables case. The left histogram represents challenge-response latencies in the benign case, and the right histogram represents the latencies in the simulated attack. As can be seen from the figure, the

[7]According to the USB specification [5], the speed between the host and the USB device is negotiated dynamically.

vast majority of benign round-trips take from 394 to 647 μs (average is 459 μs, 95% samples are in between 400 μs and 497 μs). The vast majority of the round-trip times in the simulated attack take from 530 to 2496 μs (average is 777 μs, 95% samples are in between 650 μs and 1500μs). The difference between the averages of the benign and attacking scenario is 318μs which corresponds to the average ping latency we observed between the two computers. However, the Linux kernel might queue some of the network packets. Thus we simulated an ideal kernel configuration by running `ping` in flood mode. This mode effectively fills all the network queues, hence providing an intuition of the latencies experienced with a kernel network stack optimized for minimum latency. We observed a minimum latency of 50μs and an average latency of 103μs, in flood mode. While in the following sections we build on the results presented in Figure 14, we also show that our scheme still protects against an attacker who can achieve the latencies observed with `ping` in flood mode. We further report additional experiment results including the effects of different Ethernet and `USB` cable lengths, different SGX platforms, and different PROXIMIKEY prototypes in Appendix A.

D. Initial Proximity-Verification Parameters

In Section IV-D we explained that the initial proximity verification is successful when at least fraction k of the n challenge-response latencies are below the threshold T_{con}. Now, we explain how to set these parameters based on the above experimental results. There are five interlinked parameters that one needs to consider: (i) the legitimate connection latency threshold T_{con}, (ii) total number of challenge-response rounds n, (iii) the fraction k , (iv) attacker's success probability P_{adv} that should be negligible, and (v) the legitimate success probability P_{legit} that should be high. We find suitable values for these parameters in the following order:

1. First we focus on the threshold T_{con}. The higher T_{con} is, the higher the legitimate success probability P_{legit} becomes, on the other hand, a too high value for T_{con} also makes P_{adv}, the attacker's success probability, high. Therefore, we are after a suitable value for T_{con} that keeps P_{legit} high while minimizing P_{adv} over a varied number of rounds n.
2. Based on such T_{con}, we pick a fraction k such that it maximizes the legitimate success probability P_{legit} and reduces the attacker's success probability P_{adv}.
3. Given T_{con} and k, we evaluate P_{adv} and P_{legit} over a varied number of rounds n and choose the minimum number of rounds that provides the required probabilities, since the fewer rounds, the faster the initial attestation is.

Finding suitable threshold T_{con}. Finding a suitable latency threshold T_{con} is a non-trivial task. A low threshold requires a high number of the challenge-response rounds, since the protocol (cf. Section IV) requires at least a fraction k of the observed responses to be less or equal to T_{con} and the lower threshold has very low cumulative probability value in the latency distribution (see Figure 14). Conversely, a high threshold value enables some latencies measured during an attack to be classified as legitimate local replies, hence increasing the chances of the attacker to break the proximity verification. To address this challenge, we perform a trial over multiple threshold candidates to evaluate their viability.

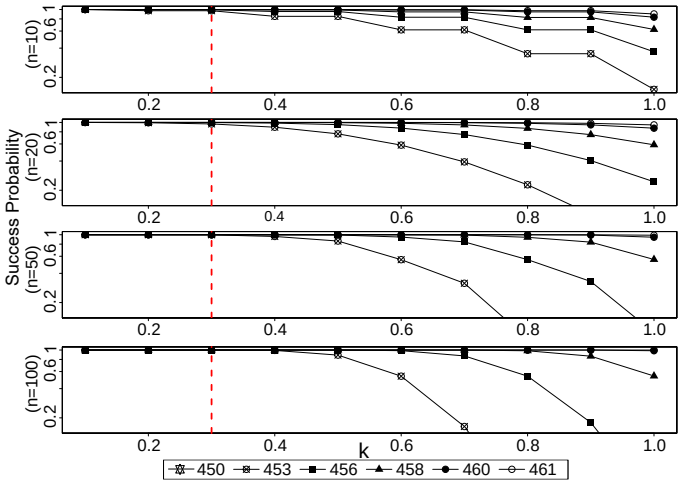

Fig. 15: **Effect of different threshold latencies (T_{con}).** The figure shows the success probability when no relay attack takes place. The threshold latency T_{con} value 461μs reaches to 0.99999985 success probability for number of trials at least 15 ($k.n$) out of 50 (n) challenge-response protocol. The red line denotes $k=0.3$.

Figure 15 shows the legitimate success probability P_{legit} for different number of rounds ($n \in \{10, 20, 50, 100\}$). We iterate through multiple threshold times ($T_{con} \in \{450\mu s, 453\mu s, 456\mu s, 458\mu s, 460\mu s, 461\mu s\}$), and 461$\mu s$ provides high success ratio for different values of k ($P_{legit} = 0.9999998$ $(n=50)$ and $P_{legit} = 0.99999999908$ $(n=100)$). We chose to test T_{con} up until 461μs because as can be observed in figure 14 for these values we almost never observe any latency response during an attacking scenario. It is possible to increment the latency further to improve the success probability (at T_{con} 471μs, $P_{legit}=0.999999999997$ $(n=50)$), but doing so will start increasing the probability for the attacker as well. After that, we estimate that any latency value less than or equals to the threshold T_{con} appears with the cumulative probability of $p_c = 0.6461$ ($p_c = Pr[396 \leq x \leq 461] = 0.6461$ where 396μs is the smallest latency experienced). Using the standard error of the mean, we estimated the error in our model, which is approximately $p_e = 1/(2\sqrt{\mathcal{N}})$ [8] where $\mathcal{N}$ denotes the number of samples drawn in our experiment. We have around $\mathcal{N} = 27$ million samples to construct the distribution. This makes the sampling error probability $p_e \approx 1.06 \times 10^{-4}$. The attacker's success probability $p_{\mathcal{A}}$ for a single round is simply the sampling error, i.e., $p_{\mathcal{A}} = p_e = 1.06 \times 10^{-4}$ due to the standard error of the mean as in our experiment we encounter zero samples within 461 μs for attack distribution. The legitimate enclave's success probability $p_{\mathcal{H}}$ for a single round is the cumulative probability above, i.e., $p_{\mathcal{H}} = p_c = 0.6461$, as can be seen from Figure 16.

Now, for both cases (simulated attack and benign case) we can model the complete challenge-response protocol of n rounds as a Bernoulli's trial where we look for at least kn responses within 461μs out of n. We can write this

[8] Hoeffding's inequality [21] describes that one needs at least $\frac{log(2/\alpha)}{2t^2}$ samples to acquire $(1-\alpha)$-confidence interval $E(\bar{X}) \pm t$, we set $1-\alpha = 95\%$.

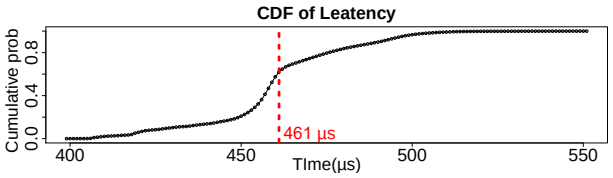

Fig. 16: **Cumulative distribution function for latencies.** We set the threshold T_{con} at 461 μs which has a cumulative probability of 0.6461 in the experiment where no rerouting attack takes place.

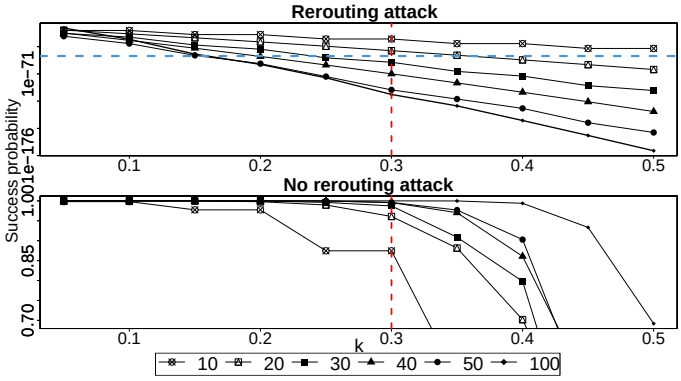

Fig. 17: **Finding suitable fraction** k**.** The graph shows the legitimate enclave's success probability in an ideal scenario and the attacker's success probability in rerouting attack scenario while the threshold time is $T_{con} = 461\mu s$. The x-axis of the graph shows the threshold k that denotes that at least nk challenge-response out of n has to be less or equal to $461\mu s$. The blue line in the upper graph denotes negligible success probability ($10^{-48} \approx 2^{-128}$) for the attacker.

cumulative probability as a binomial distribution:

$$\Pr[x \geq nk] = \sum_{i=nk}^{n} \binom{n}{i} (p)^i (1-p)^{n-i}$$

where $p \in \{p_{\mathcal{H}}, p_{\mathcal{A}}\}$.

Choosing a suitable fraction k**.** The next step of the evaluation is to find a suitable fraction k based on the threshold time T_{con}. Note that both the success probability of the attacker and the legitimate enclave is calculated as the cumulative probability from a binomial distribution (from nk to n). Hence, we require to choose a suitable value of k that maximizes P_{legit} while minimizing P_{adv}.

We calculate two graphs that are depicted in Figure 17 where the x-axis denotes k, and the y-axis denotes attacker's success probability P_{adv} and legitimate success probability P_{legit}, respectively, while using $T_{con} = 461\mu s$. We observe a sharp decrease in the legitimate success probability at $k = 0.3$. Hence, fix $k = 0.3$ to achieve the maximum P_{legit}. Additionally, in the graph of attacker's success probability, the horizontal line is placed at $10^{-48} \approx 2^{-128}$. Hence we propose to choose any round configuration bellow this horizontal line, where $n \geq 30$. With number of rounds set to $n = 50$ and $k = 0.3$, we have $P_{legit} = 0.9999998$ and $P_{adv} = 1.24 \times 10^{-54}$. Similar result could be also observed in Figure 15 where the sucess probability of the legitimate enclave decreases significantly after $k = 0.3$ for $T_{con} = 461\mu s$.

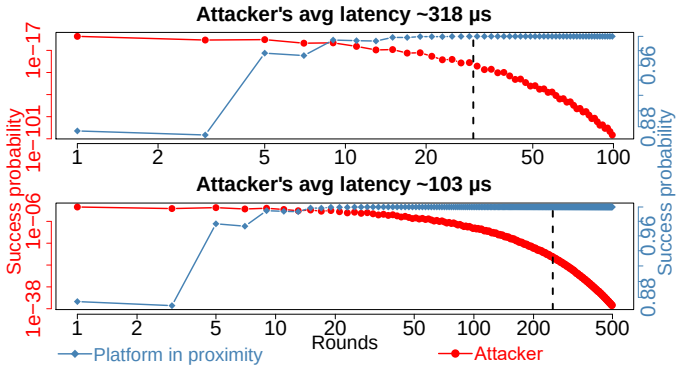

Fig. 18: **Attacker's success and legitimate success.** The plots shows the attacker's success probability P_{adv} and the legitimate success probability P_{legit} in proximity verification for different number of rounds n given a fixed fraction $k = 0.3$ in two scenarios where the attacker's network latency is $318\mu s$ and $103\mu s$ respectively.

Generalizing the number of rounds n**.** Figure 18 extends this analysis to the general number of challenge-response rounds spanning from $n = 2$ to 100 and 500 in two scenarios where the attacker's network latencies are $318\mu s$ and $103\mu s$ (ping flood mode) respectively. Here we compute the probability of attacker returning the reply within $461\mu s$ for at least $k = 0.3$ fraction of challenges. The y-axis denotes the attacker's success probability which diminishes overwhelmingly with the increasing number of challenges (keeping the fraction constant at $k = 0.3$). Notice that, by merely choosing a higher number of rounds, satisfying values for the legitimate and attacker's success probabilities can be achieved even in the case in which the attacker manages to optimize the kernel for minimum network latency. Hence, PROXIMITEE can distinguish legitimate enclave and the attacker despite the lower latency. As long as the network latency is not negligible even smaller attacker's latencies than we could measure can be protected against by employing a higher number of challenge-response rounds.

On the selection of the parameters for different settings. Note that in general, our ad hoc method for selecting the parameters involved in the protocol might not be optimal. An optimal solution is one that minimizes the number of rounds, to make the initial attestation faster, while having some lower bound constraint on P_{legit} and an upper bound constraint on P_{adv}. Nevertheless, for the system that we tested, we were able to find satisfying values for the parameters, for instance, 50 rounds can be performed in $25ms$. It is important to observe that different systems might exhibit slightly different USB and network latencies. Therefore, if the reduction of the initial attestation time is a key requirement for the system in which PROXIMIKEY is being deployed, a platform profiling step[9] might allow reducing further the time required for the initial attestation for a particular system.

E. Parameters for Periodic Proximity Verification

We set the periodic proximity verification parameters based on our experimental results following two main requirements.

[9]essentially collecting the data presented in Figure 14 for the target system

First, the attacker's success probability P'_{adv} must be negligible. Second, the probability of false positives P'_{fp} should be very low. Next, we explain the three-step process to set up the parameters: T_{detach}, w and f for the periodic proximity verification. We refer to figure 6 in Section IV-D where we discussed the sliding window strategy for the continuous proximity verification. We proceed as follows:

1. We find out a suitable latency T_{detach} that define the yellow or red round in Figure 6. Yellow window define the round of challenge response latency between T_{con} and T_{detach}, while the red window define a latency more than T_{detach}. Hence, the probabilities $\Pr[T_{con} \leq \mathcal{L}_{legit} \leq T_{detach}] = \Pr[\text{yellow}]$, and $\Pr[\mathcal{L}_{legit} \geq T_{detach}] = \Pr[\text{red}]$ should be very low. $\mathcal{L}_{legit}$ and $\mathcal{L}_A$ denote the latency of the legitimate enclave running on the platform in proximity and remote attacker platform's latency respectively.

2. Based on the threshold T_{detach}, we select a suitable sliding window size w to minimize the attacker success probability P'_{adv} to a negligible quantity.

3. We fix a suitable frequency f for the periodic challenges. A high f value terminate the communication very fast, leaving very small attacking window.

Finding suitable threshold T_{detach}. We set the threshold T_{detach} to 649 μs. We choose this value as we experience zero sample from the timing distribution (refer to the 'yellow' distribution Figure 14) where no rerouting attack takes place. While in the attacker's distribution, $Pr[530 \leq x \leq 649] = 4 \times 10^{-4}$. We account for the experimental error in our model using the standard error of the mean as $p_e \approx 1.06 \times 10^{-4}$. The value p_e signifies that a legitimate enclave running on the platform in proximity may take more than 649 μs to respond. Using T_{detach}, we can now define the challenge response rounds in Figure 6 for a *single round* as following:

$$\begin{aligned}
\Pr[\mathcal{L}_{legit} \leq T_{con}] &= \Pr[legit \in \text{green}] = 0.6461 \\
\Pr[T_{con} < \mathcal{L}_{legit} < T_{detach}] &= \Pr[legit \in \text{yellow}] = 1.06 \times 10^{-4} \\
\Pr[\mathcal{L}_{legit} \geq T_{detach}] &= \Pr[legit \in \text{red}] = 1.06 \times 10^{-4} \\
\Pr[\mathcal{L}_A \leq T_{con}] &= \Pr[A \in \text{green}] = 1.06 \times 10^{-4} \\
\Pr[T_{con} < \mathcal{L}_A < T_{detach}] &= \Pr[A \in \text{yellow}] = 4 \times 10^{-4} \\
\Pr[\mathcal{L}_A \geq T_{detach}] &= \Pr[A \in \text{red}] = 0.99996
\end{aligned}$$

Finding suitable sliding window size w. Sliding window size is analogous to that of the number of rounds n. We keep the size of the sliding window as $w = n = 50$ as it only requires the PROXIMIKEY to remember the past 50 interactions and achieve high probability for the legitimate enclave and negligible success probability for the attacker. Similar to the previous approach, only if 15 out of 50 ($k = 0.3$) challenge-response round where responses are within 461 μs, PROXIMITEE yields success probabilities as the following:

$$\begin{aligned}
\Pr[A \in \text{success window}] &= P'_{adv} = 1.6 \times 10^{-50} \\
\Pr[A \in \text{failed window}] &= P'_{fp} = 1.16 \times 10^{-8} \\
\Pr[legit \in \text{success window}] &= 0.99999623
\end{aligned}$$

leading to false negative for the legitimate enclave of 3.77×10^{-6}) for the ideal scenario (refer to Figure 17).

The probability that a halt window event occurs for a legitimate application-specific enclave running on the platform in proximity is $\approx 1.06 \times 10^{-4}$. The PROXIMIKEY halts all the data communication to the target platform until the next periodic proximity verification.

If two or more than two latencies $\geq 649\mu s$ (T_{detach}) are received, the PROXIMIKEY terminates the connection and revoke the platform. The downtime that can happen as a result of false positive during a connection of 10 years is $720ms$.

Finding suitable frequency f. The frequency f determines how fast the connection is terminated in case the PROXIMIKEY device is detached. Note that the PROXIMIKEY takes around 12 ms on average to issue a new random challenge in the legitimate case. Hence, by performing a round of the protocol as soon as the previous is over, we achieve the maximum attainable average frequency of ~ 83 rounds per second. We use this frequency as it consumes only 6.48 KB (0.0011% of the channel capacity) and allows the communication channel to be halted on average after $12ms$ of the start of a relay attack and terminated in $24ms$.

F. Performance

Finally, we evaluate the PROXIMITEE prototype performance based on the following metrics.

1. *Start up latency.* The start-up latency of the PROXIMIKEY is less than 1 second for the PROXIMIKEY to establish a `TLS` channel and execute the PROXIMITEE protocol. The initial proximity verification takes 25 ms in the case of Variant I.

2. *Operational latency and data overhead.* The operation latency is defined as the additional latency PROXIMITEE adds to the trusted path, such as sending a keystroke to the target platform from the keyboard. The operational latency is also minimal. It adds around $200\mu s$ for `TLS` and transport over the native `USB` interface of the Arduino. The data overhead is around 80 bytes per packet for the header and the MAC. Execution of the periodic PROXIMITEE protocol with 83 rounds/second only requires around 6.48 KBytes/s of data which is only 0.0011% of the `USB` 2.0 channel capacity. Note that our implementation of the PROXIMIKEY uses an `USB` 2.0 connection, one can implement the PROXIMIKEY on `USB` 3.0 based platform to gain more performance.

VIII. RELATED WORK

Proximity verification of TEEs. Previous literature has proposed to use distance bounding protocols for identification of TMP chip on a local platform [19]. The evaluation is based on a software TPM emulator [1], [4]. Bryan Parno has pointed out that TPM identification using distance bounding would be unreliable, as the attestation operations take half a second or more [29]. Previous literature has also suggested equipping TPM chips with NFC interfaces for secure connection establishment [34], but such solutions are hard to deploy in practice.

Presence attestation [37] provides a solution to bootstrap trust into her own device and ensures that the user is communicating with the genuine dynamic root of trust

(DRTM) [26]. This is achieved by showing an image from DRTM-based trusted execution environment (prover) and capture the image by a camera. Then this image is communicated to a remote verifier (e.g., server) in a small time interval. The attacker model does not assume neither physical attacks on the TEEs (extracting private key from any DRTM) nor emulation of broken TEEs on the victim's platform. Moreover, the approach is not suitable for Intel SGX as the authors assume to have a secure IO form the TEEs peripherals like camera which is not available in SGX.

Trusted path. SGXIO [35] provides a system to enable a trusted path to Intel SGX. This is achieved by using a trusted hypervisor. SGX IO uses seL4 [33] microkernel as hypervisor and requires additional device drivers to communicate with the I/O devices and requires also TPM-based trusted boot. The main problem of formally-verified minimal hypervisors and kernels is their functional restrictions and complicated updates that deployment difficult in practice (see Section III-C).

UTP [17] describes a unidirectional trusted path from the user to a remote server using dynamic root of trust based on Intel's TXT technology [26], [28]. The system suspends the execution of the OS and loads a minimal protected application for execution. This loading is measured and stored to a TPM and proved to a remote verifier using remote attestation. The protected application creates a secure channel, records user input and sends them securely to the server. UTP is limited to VGA-based text UIs to keep the TCB small and it does not apply to TEEs like Intel SGX.

Zhou et al. [38] realize a trusted path for TXT-based TEEs, again relying on a small trusted hypervisor. In this solution, also device drivers are included in the TCB. Wimpy kernel [39] is a small trusted kernel that manages device drivers for secure user input. Our approach requires no trusted hypervisor or kernel and it applies to the latest TEE architectures like SGX.

IX. Conclusion

In this paper, we have presented ProximiTEE, a system where an attached trusted embedded device enables hardened SGX attestation. Our system more secure and easier to deploy than previous solutions and it enables interesting new properties such as secure and automated revocation. We have also shown how the same approach can be extended to a trusted path solution.

References

[1] "Software tpm introduction." [Online]. Available: http://ibmswtpm.sourceforge.net/

[2] "Supported algorithms." [Online]. Available: https://rweather.github.io/arduinolibs/crypto.html

[3] "Tiny core linux, micro core linux, 12mb linux gui desktop, live, frugal, extendable." [Online]. Available: https://distro.ibiblio.org/tinycorelinux/

[4] "Trousers." [Online]. Available: https://sourceforge.net/projects/trousers/

[5] "Universal serial bus specification revision 2.0." [Online]. Available: http://www.usb.org/developers/docs/usb20_docs/usb_20_072418.zip

[6] "Intel sgx homepage," Jun 2017. [Online]. Available: https://software.intel.com/en-us/sgx

[7] "Local attestation," May 2018. [Online]. Available: https://software.intel.com/en-us/sgx-sdk-dev-reference-local-attestation

[8] "Local (intra-platform) attestation," May 2018. [Online]. Available: https://software.intel.com/en-us/node/702983

[9] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell *et al.*, "Scone: Secure linux containers with intel sgx." in *OSDI*, 2016.

[10] S. Brands and D. Chaum, "Distance-bounding protocols," in *EUROCRYPT '93*, T. Helleseth, Ed.

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *USENIX WOOT17*.

[12] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "Securekeeper: confidential zookeeper using intel sgx," in *Middleware 2016*, 2016.

[13] J. V. Bulck, F. Piessens, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *USENIX Security 18*.

[14] Chandler, Matt, and Intel, "Intel enhanced privacy id (epid) security technology," Jul 2017. [Online]. Available: https://software.intel.com/en-us/articles/intel-enhanced-privacy-id-epid-security-technology

[15] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Leaking enclave secrets via speculative execution," *CoRR*, 2018.

[16] V. Costan and S. Devadas, "Intel sgx explained," Cryptology ePrint Archive, Report 2016/086, 2016.

[17] A. Filyanov, J. M. McCuney, A. R. Sadeghiz, and M. Winandy, "Uni-directional trusted path: Transaction confirmation on just one device," in *IEEE/IFIP DSN 2011*.

[18] A. Filyanov, J. M. McCuney, A.-R. Sadeghiz, and M. Winandy, "Uni-directional trusted path: Transaction confirmation on just one device," in *IEEE/IFIP DSN 2011*.

[19] R. A. Fink, A. T. Sherman, A. O. Mitchell, and D. C. Challener, "Catching the cuckoo: Verifying tpm proximity using a quote timing side-channel," in *Trust and Trustworthy Computing*, 2011.

[20] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*, 2017.

[21] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American Statistical Association*.

[22] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, 2018.

[23] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *USENIX Security 2017*.

[24] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, 2018.

[25] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "Rote: Rollback protection for trusted execution." in *USENIX Security 17*.

[26] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *ACM SIGOPS Operating Systems Review*, 2008.

[27] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017.

[28] C. Nie, "Dynamic root of trust in trusted computing," in *TKK T1105290 Seminar on Network Security*. Citeseer, 2007.

[29] B. Parno, "Bootstrapping trust in a" trusted" platform." in *HotSec*, 2008.

[30] A.-R. Sadeghi and C. Stüble, "Property-based attestation for computing platforms: Caring about properties, not mechanisms," in *Proceedings of the 2004 Workshop on New Security Paradigms*.

[31] U. Savagaonkar, N. Porter, N. Taha, B. Serebrin, and N. Mueller, "Titan in depth: Security in plaintext." [Online]. Available: https://cloudplatform.googleblog.com/2017/08/Titan-in-depth-security-in-plaintext.html

[32] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *IEEE S&P 2015*.

[33] seL4, "sel4/sel4." [Online]. Available: https://github.com/seL4/seL4

[34] R. Toegl, "Tagging the turtle: Local attestation for kiosk computing," in *Advances in Information Security and Assurance*, J. H. Park, H.-H. Chen, M. Atiquzzaman, C. Lee, T.-h. Kim, and S.-S. Yeo, Eds., 2009.

[35] S. Weiser and M. Werner, "Sgxio: Generic trusted i/o path for intel sgx," ser. CODASPY '17.

[36] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE S&P 2015*.

[37] Z. Zhang, X. Ding, G. Tsudik, J. Cui, and Z. Li, "Presence attestation: The missing link in dynamic trust bootstrapping," in *CCS '17*.

[38] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *IEEE S&P 2012*.

[39] Z. Zhou, M. Yu, and V. D. Gligor, "Dancing with giants: Wimpy kernels for on-demand isolated i/o," in *IEEE S&P 2014*.

APPENDIX A
FURTHER EXPERIMENTAL RESULTS

Here we provide additional experimental results of PROXIMITEE. We evaluated the consistency of measured latencies across different prototype platforms. Figure 19 shows the frequency distribution of latencies across three SGX platforms and three PROXIMIKEY's. We conclude that measurements are consistent result across devices. The two Intel NUCs are few microseconds faster than the Dell Latitude laptop. Additionally, we evaluated the effect of two different `USB` cable lengths (3m and 1m) and three different Ethernet cables (lengths of 1m, 7m, and 10m). Figure 20 shows that the `USB` cable has very small effect on the latency (around 10 μs average difference). It shows no significant differences between the different cable lengths.

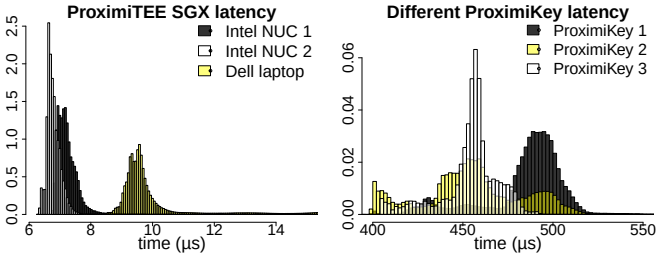

Fig. 19: **Different target platforms/PROXIMIKEY.** We evaluates latencies using three different SGX platforms. The Intel NUCs were few microseconds faster. Additionally, we evaluated latencies using three different Arduino boards. The latencies are consistent.

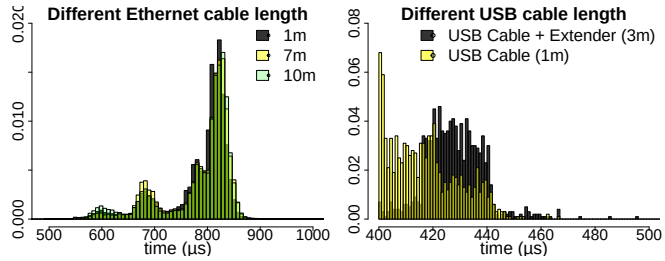

Fig. 20: **Different Ethernet/USB cables.** We evaluated latencies two different USB cables: one with an `USB` cable (1m) and another with an `USB` extender of length 2m attached. Additionally, we evaluated latencies using three different Ethernet cables (1, 7 and 10 m). Latencies are consistent.