ProximiTEE: Hardened SGX Attestation Using an Embedded Device and Proximity Verification

Aritra Dhar, Ivan Puddu,Kari Kostiainen, Srdjan Čapkun
aritra.dhar@inf.ethz.ch
ETH Zurich

ABSTRACT

Intel SGX enables protected enclaves on untrusted computing platforms. An important part of SGX is its remote attestation mechanism that allows a remote verifier to check that an enclave was correctly instantiated before provisioning secrets to it. However, SGX attestation is vulnerable to relay attacks where the attacker, by controlling a malicious OS, redirects the attestation and therefore the provisioning of confidential data to a platform that he physically controls. Such redirection increases the adversary's abilities to compromise the enclave, arming her with physical and digital side-channel attacks that would not be otherwise possible.

In this paper, we propose ProximiTEE, a novel solution to prevent relay attacks. Our solution is based on a simple embedded device, and it is best suited to scenarios where the deployment cost of such a device is minor compared to its security benefit. During attestation, the embedded device that is attached to the target platform verifies the proximity of the attested enclave using distance bounding, thus allowing secure attestation regardless of a compromised OS. The device also performs periodic proximity verification which enables secure enclave revocation by simply detaching the device. Our evaluation shows that proximity verification is secure and reliable for SGX, even using a slow prototype device and assuming very fast adversaries.

Additionally, we consider a stronger adversary that has a leaked, but not yet revoked, SGX attestation key and emulates an enclave on the target platform. To address such emulation attacks, we propose a solution where the target platform is securely initialized by booting it from the attached embedded device. Finally, we show how our hardened attestation can be used to build a trusted path for SGX.

1 INTRODUCTION

Trusted execution environments (TEEs) like Intel's SGX [3] enable secure applications on untrusted computing platforms. SGX isolates enclaves from all the other software running on the same platform, including the privileged OS. The primary security guarantees of SGX are enclave's data confidentiality and code integrity. The remote attestation mechanism allows a remote verifier to check that an enclave was constructed correctly. When an enclave is created, the CPU measures its code and during attestation signs the measurement using its attestation 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 attested enclave.

Relay attacks. While remote attestation guarantees that the attested enclave runs the expected code, it *does not*, however, guarantee that the enclave runs on the expected computing platform. As shown in Figure 1, an adversary that controls the OS on the target platform can relay incoming attestation requests to another platform under his possession. Relay attacks are a long-standing

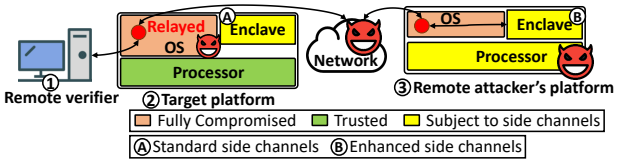

Figure 1: Relay attack. The adversary redirects attestation to his own platform which gives him increased side-channel abilities to attack the attested enclave and its secret data.

open problem in trusted computing, as already a decade ago Parno identified such attacks in the context of TPM attestation and called them "cuckoo attacks" [33].

Upon a first look, it might seem that relay attacks do not pose a problem for TEEs. If the attacker relays to another machine, the same security guarantees should hold since the data will only be available within the remote TEE and the code that is executing is attested. However, this reasoning is not correct, as we demonstrate by performing a careful analysis of the consequences of relay attacks and showing that relaying increases the adversary's capabilities to compromise the attested enclave significantly.

By relaying data from a platform in which the attacker has privileged software control to one where he has physical and full software control, the primary (and intuitive) benefit for the attacker is that she can employ various *physical* side-channel attacks, which have been shown to be both effective and inexpensive [21, 22, 39, 41]. The secondary, and more subtle, benefit is that relaying also enables certain *digital* side-channel attacks that would not be possible otherwise. For example, assume that the enclave is hardened against all known digital side-channels at the time of attestation and secret provisioning. Then, after attestation, the OS compromise in the target platform is detected and disinfected, and later a new side-channel is discovered. If redirection took place during attestation, the adversary could leverage the new side channel to extract the enclave's secrets. Without the relay, the new side channel cannot be exploited.

A typical "solution" to relay attacks is to assume *trust on first use* (TOFU). In a simple TOFU approach, the attestation is performed immediately after a fresh OS installation. However, in many application scenarios re-installation of the OS (for every attestation request) is impractical. In another common TOFU scheme, an enclave creates a key pair during an initialization phase that is assumed trusted. The key is signed by a trusted authority that allows remote clients to establish secure connections to the enclave later. The main problems with this approach are that it is limited to enclaves that are known at the time of initialization and that the OS needs to be trusted, even if only momentarily.

Our solution. In this paper, we propose a solution, called ProximiTEE, that prevents relay attacks by leveraging a simple and auxiliary embedded device that is attached to the attested target platform. Our solution is best suited to scenarios where i) the deployment cost of such an embedded device is minor compared to the benefit of more secure attestation, and ii) TOFU solutions are infeasible as reinstalling OS before every deployment is impractical. Attestation of servers at cloud computing platforms and setup of SGX-based permissioned blockchains are two examples of such deployments.

In our solution, the device performs *proximity verification* to prevent relay attacks. During attestation, the remote verifier establishes a secure connection to a specific device whose public key it knows through standard device certification. The device performs normal SGX attestation and additionally verifies the proximity of the attested enclave using a distance-bounding protocol [11].

After the initial attestation, the device performs *periodic* distance-bounding measurements and the communication channel created during the attestation stays active only as long as the device is connected to the same platform. Thus, the physical act of attaching the device to an SGX platform enables secure attestation (enrollment) while detaching the device will prevent further communication with the attested enclave (revocation). Neither enrollment nor revocation requires interaction with a trusted authority. This property is useful in applications like permissioned blockchains where validator nodes are separate organizations assigned by a trusted authority. The authority can issue one device per organization, and each organization is free to manage their computing resources (e.g., detach the device from one platform and attach it to another) without interaction with the authority. Another interesting application for our solution is implementing flexible access control policies for HSM-protected keys using an attested enclave that is guaranteed to be located in the proximity of the HSM.

Main results. Parno identified distance bounding as a candidate solution to TPM relay attacks already ten years ago [33], but concluded that it could not be realized securely as the slow TPM identification operations (signatures) make a local and relayed attestation indistinguishable. Our evaluation shows that proximity verification is possible for SGX assuming very fast adversaries. The main reason why distance bounding protocols work for SGX, but not with TPMs, is that SGX is a programmable TEE where it is possible to use pre-established security associations and efficient challenge-response protocols based on simple operations such as `XOR`.

To experimentally evaluate proximity verification on SGX, we implemented a prototype of our solution using an Arduino Due microcontroller prototyping board that is connected to the target device over the `USB 2.0` interface. We *simulated* a powerful adversary that can perform the required protocol computation instantly and is connected to the target platform with a fast and short network connection (1m Ethernet cable). In our test setup, the adversary's average latency is $120\mu s$ (less than `ping`). Our experiments and analysis show that against such an adversary the initial proximity verification during attestation is secure and reliable. The adversary's probability of performing a successful relay attack is negligible (2.71×10^{-67}), while legitimate verification succeeds with a very high probability (0.999999965). Importantly, the adversary cannot increase his success probability with repeated attempts, as attestation is triggered by the trusted remote verifier. Additionally, we show that even if the adversary would have infinitely fast network interfaces, secure proximity verification would still be possible.

Our experiments also show that enclave revocation using periodic proximity verification is practical. The probability that the device is disconnected but the enclave is not revoked is negligible (2.71×10^{-67}). The probability that the connection is halted when the device is connected is very small (translating to 2 minutes of downtime in 10 years of operation).

The performance overhead of proximity verification is small: the initial proximity verification adds only a minor delay of 25 ms to the attestation protocol, and the periodic proximity verification consumes 0.0011% of the available `USB 2.0` channel capacity. Our implementation also shows that the complexity of such a device can be small: the software TCB of our unoptimized prototype implementation is 3.6 KLoC.

Addressing emulation attacks. We then consider a stronger adversary that has obtained leaked, but not yet revoked, attestation keys and can *emulate* an SGX-enabled processor.

Proximity verification alone cannot prevent the emulation attacks, as a perfectly emulated enclave would pass any proximity test. Therefore, we propose a second attestation mechanism based on *boot-time initialization*. In this solution, the target platform loads a small, single-purpose kernel from the attached device and launches an enclave that seals a secret key known by the device. Subsequently, when attestation is needed, the enclave can verify the proximity of other enclaves on the same platform using SGX's local attestation. This enables secure attestation regardless of potentially leaked attestation keys. Our second solution can be seen as a novel variant of the well-known TOFU principle. The main benefits over previous variants are easier adoption (e.g., no OS re-installation or pre-defined enclaves) and increased security (e.g., no need to trust the standard OS even temporarily).

Trusted path. Finally, we show how our attestation mechanisms can be used as a building block for *trusted path*. The term trusted path refers to a secure communication channel between a user and enclave, and such property is missing from SGX, since the OS can manipulate any I/O between peripherals and enclaves. In our trusted path solution, the embedded device functions as a *bridge* between the I/O devices and the SGX platform. The device attests the local enclave, shows its identity to the user, and then securely mediates communication between the peripherals and the enclave.

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

1. **Analysis of relay attack implications.** While relay attacks have been known for more than a decade, their consequences regarding the adversary's attack space have not been analyzed in detail, and are often overlooked. In this paper, we provide the first analysis in this direction and show how relaying amplifies the adversary's capabilities for attacking enclaves.

2. **ProximiTEE: Addressing relay attacks.** The main contribution of this paper is that we design and implement a hardened SGX attestation mechanism based on an embedded device and proximity verification to prevent relay attacks. The attestation security of

ProximiTEE does not rely on the common TOFU assumption, and hence requires only a small TCB. Our experimental evaluation is the first to show that proximity verification is secure and reliable for SGX.

3. **Addressing emulation attacks.** We also propose another attestation mechanism based on boot-time initialization to prevent emulation attacks. This mechanism is a novel variant of TOFU with deployment, security and revocation benefits.

4. **Trusted path.** We show how our attestation mechanisms can be used to build a trusted path for SGX.

The rest of this paper is organized as follows. Section 2 describes relay attacks and their implications. Section 3 presents our solution, ProximiTEE. Section 4 explains our implementation and evaluation. Section 5 explains how to address emulation attacks and Section 6 focuses on trusted path. Section 7 provides discussion, Section 8 reviews related work, and Section 9 concludes the paper.

2 PROBLEM STATEMENT

In this section, we explain our system model, analyze the implications of relay attacks, and explain the limitations of previous solutions.

2.1 System Model

We consider a simple system model, shown in Figure 1, that consists of three parties:

① **Remote verifier** is a trusted party, connected to the attested target platform either over network channel or local interface.

② **Target platform** is the attested SGX platforms that is the target of the attack.

③ **Attacker's platform** is similar to the target platform and connected to it over a network such as Internet.

2.2 Relay Attacks

Adversary model. In the first part of this paper, we consider the following adversary model that we call the *relay attacker*. The relay attacker controls the OS and all other privileged software on the target platform *temporarily*, in particular at the time of the remote attestation. The OS compromise on the target platform may be later detected and disinfected. The attacker *does not* have physical access to the target platform and cannot extract attestation or sealing keys from the target platform.

The relay attacker controls the OS and all other privileged software on his platform *permanently* and has physical access to that platform. The attacker also controls the network between the target platform and his platform. At the time of the attestation, the adversary has not been able to extract attestation or sealing keys from his platform or any other SGX processor.

The relay attack. Since the adversary controls the OS on the target platform, he can redirect the attestation requests intended for the target platform to his platform, as shown in Figure 1. SGX's attestation protocol is based on EPID group signatures and anonymous in the sense that it prevents identification of the physical platform from the signed attestation response received from the

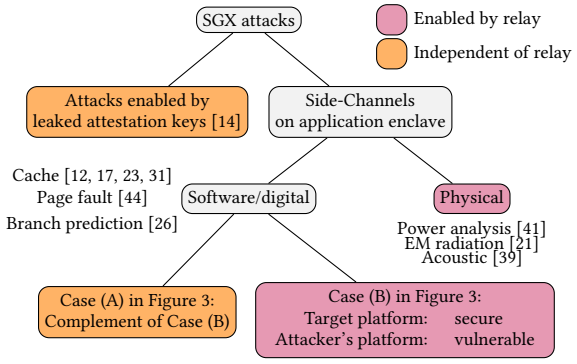

Figure 2: Relay attack implications. The tree shows the types of attacks that are enabled by redirection and ones that are independent of relay.

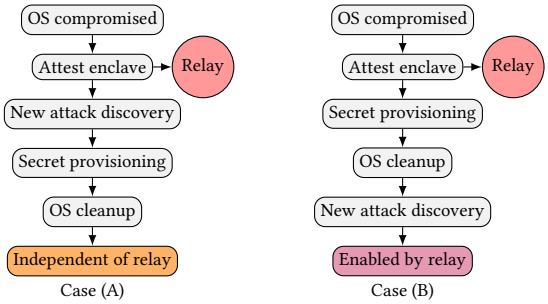

Figure 3: Example sequences of events where attestation redirection either enables digital side-channel attacks (B) or where the attack success is independent of relay (A).

Intel Attestation Service (IAS) server.[1] Appendix A provides more details on SGX's attestation.

Relay attack implications. The main consequence of attestation redirection is that it increases the adversary's ability to attack the attested enclave. In Figure 2 we highlight two major classes of attacks: those that are only possible by first performing a relay attack, which we denote as "*enabled by relay*", and those that can be done whether or not the attacker also does a relay attack, which we call "*independent of relay*."

Our first observation is that attacks based on leaked attestation keys (e.g., ones obtained through the Meltdown platform vulnerability as demonstrated in the Foreshadow attack [14]) are independent of relaying. If the adversary has obtained a valid and non-revoked attestation key, he can emulate an SGX processor on the target platform and obtain any secrets provisioned to it. We revisit such emulation attacks in Section 5.

The main benefit of the relay, from the adversary's point of view, is that it enables *physical* side-channel attacks against third-party application enclaves. Once a secret has been provisioned to the attacker's platform, she has as much time as she likes to perform

[1] Although the EPID signatures do identify the *group* of the attested processor (e.g., manufacturing batch), such grouping alone is not an effective defense against relay attacks, since nothing prevents the adversary from obtaining one or more processors from the same batch. SGX also supports a *linkable* attestation mode that allows the remote verifier to link two attestations of the same platform, but it does not prevent relay attacks during the first attestation.

the attack. Some examples of physical side-channel attacks are acoustic, electric and electromagnetic monitoring, which have been shown to be both effective and inexpensive means to extract secrets from modern PC platforms (see [22] for a good summary of known attacks and their capabilities). Since the adversary does not have physical access to the target platform, such attacks are not possible without relay. Hardening programs like enclaves against physical side channels is difficult and currently an open problem [22]. Therefore, developers cannot easily defend their enclaves against physical side channels enabled by relay.

The second, and more subtle, implication of relay is that it can also enable *digital* side-channel attacks, such as ones that extract enclave secrets by monitoring page faults [44], caches [12, 17, 23, 31] or branch prediction [26]. Whether a digital side-channel attack is enabled by relaying or not, depends on the sequence in which particular events occur. These events include, but are not restricted to: the provisioning of secrets to the enclave, the possible disinfection of the target platform from malicious software, and the discovery of a new side-channel attack. We group the relative ordering of these events into two cases: A and B. Case A covers event sequences that only lead to attacks which are independent of relay and Case B covers event sequences in which redirection gives extra capabilities to the adversary. Below, and in Figure 3, we provide examples of sequences belonging to these two cases:

Case A: independent of relay. A digital side-channel is independent of relay if the adversary could perform it on the target platform as well. An example of such case is shown in Figure 3, where a new attack is discovered after secret provisioning but before the target platform OS is disinfected.

Case B: attack enabled by relay. Case B is reached whenever it occurs that by using a side channel the enclave is exploitable on the attacker's platform, but not on the target platform. An example of such a case is shown in Figure 3, where at the time of attestation and secret provisioning, the enclave is hardened against all known digital side-channel attacks (using tools like Raccoon [34], ZeroTrace [36] or Obfscuro [9]). After secret provisioning, the OS compromise is detected and cleaned. Later, a new side-channel attack vector (that is not prevented by the used tools) is discovered. If the adversary performed redirection and the secret was provisioned to the attacker's machine, the new side channel is exploitable. Without the relay, the attack is not possible.

2.3 Limitations of Known Solutions

To address attestation attacks, a common approach in the literature is to assume *trust on first use* (TOFU) [43]. Typical simple solutions assume that the OS is clean at the time of attestation or perform attestation only immediately after fresh OS installation. Both of these approaches have obvious security and deployment problems.

Recent research papers use slightly better TOFU variants [10, 13, 29, 37]; however their descriptions of the used attestation procedures are not always very accurate, as these papers focus on other problems than attestation. For example, the Rote system [29] requires secure connections to enclaves on specific platforms to establish an enclave group. The proposed solution assumes fresh OS installation at system initialization time and for each used platform it requires a local administrator to input a credential to the enclaves.

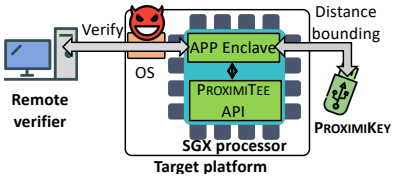

Figure 4: Approach overview. A trusted embedded device ProximiKey is attached to the target platform, verifies the proximity of the attested enclave, and enables a secure connection to it.

As another example, in the VC3 system [37] enclaves generate a public/private key pair at the time of trusted initialization, output the public key and seal the private key. The public key can be sent to a trusted authority for certification, which then enables clients to securely connect to enclaves.

Both of these solutions essentially avoid insecure attestation by pre-authorizing known enclaves during a setup phase that is assumed trusted. Such TOFU solutions have the following limitations:

1. **OS re-installation:** Forcing users or administrators to re-install the OS is not always possible.
2. **Manual configuration:** Manual interaction tasks, such as an administrator that needs to enter credentials to enclaves during initialization, complicates platform enrollment, especially in scenarios like data centers with many enrolled platforms.
3. **Pre-defined enclaves:** Solutions that only work with enclaves that are known at the time of initialization are not applicable to scenarios like cloud computing platforms where users need to install new enclaves after platform installation.
4. **Large temporary TCB:** Modern operating systems have a large TCB and trusting the OS even temporarily is unideal.
5. **Online authorities:** Solutions where a trusted authority needs to either certify or revoke new enclaves typically require that the authorities are online, which increases their attack surface.

3 PROXIMITEE: ADDRESSING RELAY ATTACKS

Our goal in this paper is to design a solution that addresses the above limitations of previous solutions. In short, our solution should be *secure* (small TCB, no online authorities) and *easy to deploy* (no OS re-installation, manual configuration or pre-defined enclaves). In this section, we provide an overview of our approach, outline possible use cases, describe an attestation solution against relay attacks and analyze its security.

3.1 Approach Overview

We propose a hardened SGX attestation scheme, called ProximiTEE, based on a simple and auxiliary embedded device that we call ProximiKey. The embedded device is attached to the target platform over a local communication interface, such as USB, as shown in Figure 4.

Our main idea is to use the combination of such trusted device and *proximity verification* to prevent relay attacks. In our solution,

the ProximiKey device verifies the proximity of the attested enclave and after successful proximity verification it facilitates the creation of a secure channel between the remote verifier and the attested enclave. After the initial attestation, the device periodically checks proximity to the attested enclave. 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 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 API that facilitates communications between the enclave and the device (see Figure 4).

Security assumptions. In our solution, the ProximiKey device is a trusted component. We deem this choice reasonable since it implements only the strictly necessary functions and therefore it a has significantly smaller software TCB, attack surface, and complexity compared to the host OS. We assume that its issuer certifies each embedded device prior to its deployment and such certification can take place fully offline.

Concerning the security of the ProximiKey device we employ the same adversary model introduced in Section 2 for enclaves. While the user's device and its private keys are never exposed to the attacker, another similar device can be in the physical possession of the attacker, which has as much time as she wants to fully compromise it (run arbitrary code and extract keys).

3.2 Example Use Cases

Our solution is targeted to scenarios where the benefits of more secure attestation outweigh the deployment cost of a simple embedded device. Here, we outline four of such cases.

Data center. In our first example, we consider a cloud platform provider that attaches ProximiKey to a server in a specific data center and makes the public key of the connected device known to the users of the service. Our approach is particularly well suited to cloud computing models where customers rent dedicated computing resources like entire servers. In such a setting, our solution ensures that the cloud platform customer outsources data and computation to a server that resides in a specified location. Enforcing location may be desirable to meet increasing data protection regulation that defines how and where data can be stored, even if protected by TEEs such as SGX. Revocation (e.g., when a server is relocated to another data center or function) can be realized by merely detaching ProximiKey.

Permissioned blockchain. Our second case is a setting in which a trusted authority initializes a set of validator nodes for a permissioned and SGX-hardened blockchain. The trusted authority issues one ProximiKey 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 a trusted authority. Furthermore, since ProximiKey can only be active on one platform at the time, such a deployment enables the authority to bound the "voting power" (e.g., identities in Byzantine consensus) of each validator organization in the blockchain consensus.

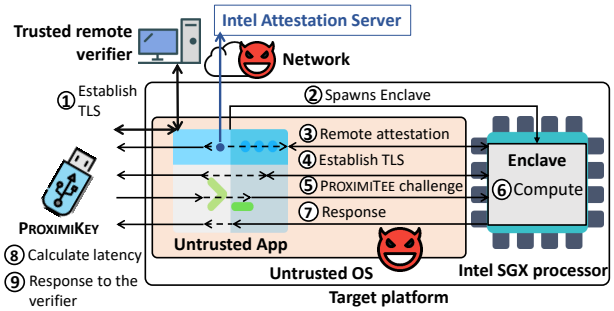

Figure 5: ProximiTEE attestation. The remote verifier establishes a secure channel to the ProximiKey device that performs standard attestation and verifies the proximity of the attested enclave.

Trusted path. Our third example is trusted path – an important property that is lacking from SGX. Our attestation approach provides the means to establish trusted path between I/O peripherals and enclaves. We discuss this solution in more detail in Section 6.

HSM-protected keys. Our last case is the management of HSM-protected keys from an attested enclave. Such deployment enables the secure and flexible realization of various access control policies, implemented as attested enclaves. ProximiTEE guarantees that only an enclave in the proximity of the HSM can control its keys. Such solution provides a high level of protection because, at no point in time, the HSM keys are directly accessible by the enclave (which may be vulnerable to side-channel attacks) or by the untrusted OS.

3.3 Solution Details

Now, we explain the ProximiTEE attestation mechanism in detail.

I. 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 to verify the code configuration of the enclave with the help of the IAS server (see Appendix A). In the attestation protocol, the device learns the public key of the attested 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.

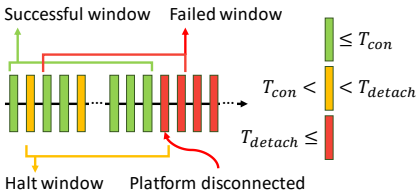

Figure 6: Sliding window for periodic proximity verification with three different types of challenge-response latencies.

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 $k \times n$ 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.

II. Periodic proximity verification. After the initial connection establishment, the ProximiKey device performs *periodic* proximity verification on the attested enclave. ProximiKey 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 elaborated in Section 4 there are 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 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 ProximiKey stops forwarding data 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 ProximiKey terminates the communication and revokes the attested platform.

3.4 Security Analysis

Attestation security. To analyze the security of our hardened attestation mechanism, 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 the ProximiKey device is attached.

The task of establishing a secure channel to the correct enclave can be broken into two subtasks. The first subtask is to establish a secure channel to the correct ProximiKey device. 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 (1m Ethernet cable). Since the attacker's platform might be faster than the target platform, we simulate an adversary that can perform instantly all computation needed to participate in the proximity verification protocol. However, even this adversary cannot break cryptographic hardness assumptions. 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 denote 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 4 our experiments and analysis show that example parameter values $n = 50$, $k = 0.4$ and $T_{con} = 470\mu s$ 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: $P_{adv} = 2.71 \times 10^{-67}$. With the same parameters, proximity verification with the legitimate local enclave succeeds with a very high probability: $P_{legit} = 0.999999965$.

Revocation security. To analyze the security of the periodic proximity verification that we use 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 device 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 2.1), 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 the periodic proximity verification as broken if the adversary can manage to not terminate the previously established connection within a "short delay" after the ProximiKey was detached from the attested target platform. For most practical purposes we consider a delay of 1 ms as sufficiently short. We denote the adversary's success probability in breaking the periodic proximity verification as P'_{adv}. A false positive for periodic attestation is the event where the connection to the legitimate enclave is terminated, and the attested platform is revoked despite the ProximiKey being connected to the target platform. We denote the probability that this happens during a "long period" as P'_{fp}. We consider an example period of 10 years sufficiently long for most practical deployments.

In Section 4 evaluate this experimentally, again by simulating a similar powerful relay attacker. We show that there exists a set of parameters that provide secure, reliable and inexpensive channel termination and platform revocation. The attacker's success probability can be made negligible: $P'_{avd} = 2.71 \times 10^{-67}$, while keeping the false positive probability low: $P'_{fp} = 5 \times 10^{-5}$. 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.

4 EVALUATION OF PROXIMITEE

In this section, we describe our ProximiTEE prototype implementation and experimental evaluation.

4.1 Implementation

We implemented a complete prototype of the ProximiTEE system. Our implementation consists of two components: (i) ProximiKey prototype, and (ii) ProximiTEE enclave API which enables any application-specific enclave to communicate with the ProximiKey device and execute our protocols.

ProximiKey. Our prototype consists of one Arduino Due prototyping board equipped with an 84 MHz ARM Cortex-M3 microcontroller. The board communicated with the target platform over a native USB 2.0 connection that provides 480 Mbps of bandwidth. 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 as the message authentication code, Curve25519 for Diffie-Hellman key exchange and SHA256 as the hash function. Our prototype implementation is approximately 200 lines of code, and the code size of the TLS library is around 3.6 KLoC.

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 the TLS implementation. The prototype is around 200 lines of code.

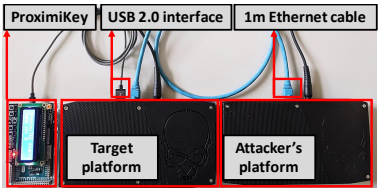

Figure 7: Our experimental setup consists of the ProximiKey device prototype, the target platform, the attacker's platform and the connection interfaces between them.

4.2 Experimental Setup: Simulated Attackers

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., the SGX platform to which our ProximiKey prototype was connected to). Second, we *simulated* a relay-attack adversary whose platform was connected to the target platform via a direct Ethernet cable (see Figure 7). 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.

For the relay attack, 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.

We experimented with the possible *software optimizations* that the adversary could perform, since he controls the OS on the target platform. First, we tested the standard `ping` tool which gave a latency of around 380 μs for one-meter Ethernet connection. After that, we used the `ping` tool in so called flood mode [27] and measured a reduced average network latency of around 153 μs (command `ping -s 300 -af`). Flood mode achieves faster round-trip time as the it forces the OS to fill up the network queue of the kernel. Based on these measurements, we chose to simulate an attacker that fills the kernel's network queues (on both platforms) similar to the flood mode to minimize latency.

Additionally, we consider an adversary that performs *hardware optimizations* on his platform. To consider the worst case, we assume an adversary who has an infinitely fast network interface. Assuming that the transmission time spent on the wire is negligible and most the the round-trip latency is due to processing the in the network interface, such an adversary can be safely approximated by cutting down our measured latencies by half, because in our test setup both platforms have identical network interfaces.

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 27.8 million

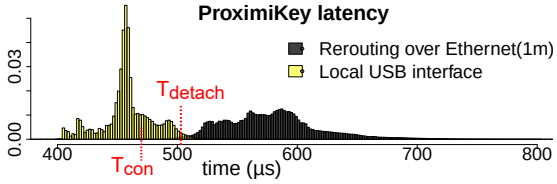

Figure 8: Latency distributions for legitimate local USB connection and simulated 1m Ethernet relay attack. T_{con} and T_{detach} thresholds are placed at 470 μs and 510 μs.

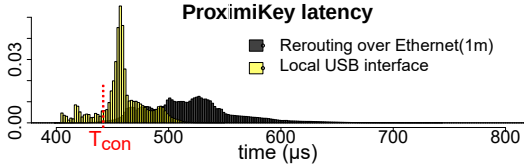

Figure 9: Latency distributions for legitimate local USB connection and relay adversary with infinitely fast network interface (latency is 72μs). T_{con} threshold is placed at 442 μs.

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.

4.3 Latency Distributions

Figure 8 shows our main experimental result for the 1m Ethernet and 1m USB 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 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 451 to 1251 μs (average is 579 μs, 95% samples are in between 490 μs and 655 μs). The difference between the averages of the benign and attack scenarios is 120 μs which is less than the previously tested flood mode `ping` latency (153 μs).

Figure 9 shows the latency distributions for the local enclave and the adversary with an infinitely fast network interface (i.e., latencies reduced to half).

We report additional results for different Ethernet and USB cable lengths, SGX platforms, ProximiKey prototypes, effects of loads on the CPU cores, and effect of application pinning to a specific CPU cores in Appendix D.

4.4 Initial Proximity Verification Parameters

As explained in Section 3.3, 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 our strategy for setting these parameters based on the above 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

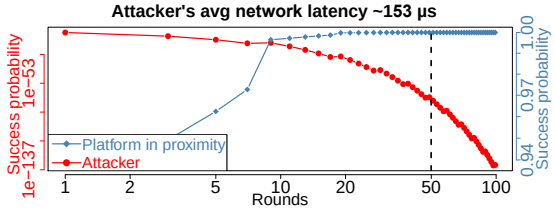

Figure 10: Distinguishing relay attack. The attacker's success probability P_{adv} and the legitimate success probability P_{legit} in proximity verification for different number of rounds (n) given a fixed $k = 0.4$.

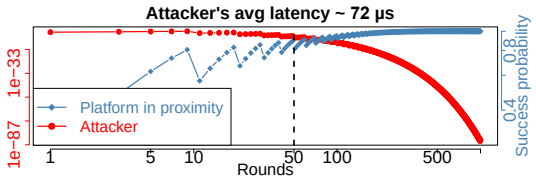

Figure 11: Distinguishing relay attack for adversary with infinitely fast network interface. The attacker's success probability P_{adv} and the legitimate success probability P_{legit} for different number of rounds (n) given a fixed $k = 0.4$.

(v) the legitimate success probability P_{legit} that should be high. We find suitable values for these parameters in the following order:

1. We start with 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. Figure 18 in Appendix C shows effects of different T_{con} values.

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}. Figure 20 in Appendix C illustrates the effect of different values of k on the legitimate enclave and the attacker's success probability.

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.

Main result. Figure 10 shows the legitimate enclave's success probability P_{legit} and the attacker's success probability P_{adv} with different number of rounds. Based on our experiments we set $T_{con} = 470\mu s$, the threshold fraction $k = 0.4$ and the number of rounds $n = 50$ which yields a legitimate success probability $P_{legit} = 0.999999965$ and an attacker's success probability $P_{adv} = 2.71 \times 10^{-67}$. We provide the full details of of this parameter tuning process in Appendix C.1.

Figure 11 shows the success probability of legitimate enclave and the adversary with infinitely fast network interface. With around $n = 1000$ rounds, the attacker's success probability is 0.9999993278 while the attacker's success probability is only 2.3×10^{-85}.

4.5 Periodic Proximity Verification Params

For periodic proximity verification we have two main requirements. First, the attacker's success probability P'_{adv} must be negligible. Recall that P'_{adv} refers to an event where the device is detached but the connection is not terminated sufficiently fast. Second, the probability of false positives P'_{fp} should be very low. Recall that P'_{fp} refers to an event where the connection is terminated when the device is still attached.

Next, we explain the three-step process to set up parameters T_{detach}, w and f for the periodic proximity verification. (Recall Figure 6 for the sliding window strategy.) We proceed as follows:

1. We find out a suitable latency T_{detach} that define the yellow or red round in Figure 6. The yellow window defines the round of challenge response latency between T_{con} and T_{detach}, while the red window defines a latency more than T_{detach}. Hence, the probabilities $\Pr[T_{con} \leq \mathcal{L}_{legit} \leq T_{detach}] = \Pr[legit \in \text{yellow}]$, and $\Pr[\mathcal{L}_{legit} \geq T_{detach}] = \Pr[legit \in \text{red}]$ 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.

Main result. Based on our results and the above strategy, we set the periodic proximity verification parameters to the following values: $\Pr[A \in \text{success window}] = P'_{adv} = P'_{fn} = 2.71 \times 10^{-67}$, $\Pr[legit \in \text{success window}] = 0.999999965$ and $\Pr[legit \in \text{failed window}] = P'_{fp} = \Pr[legit \in \text{red}]^2 = 5 \times 10^{-5}$. If at least two latencies $\geq 510\mu s$ (T_{detach}) are received, the ProximiKey terminates the connection and revokes the platform. The average downtime due to false positives occurring during a connection of 10 years is around 2 minutes. We provide the full details of this parameter tuning process in Appendix C.2.

4.6 Performance

Finally, we evaluated the performance of our solution.

1. **Start-up latency** of our solution is less than 1 second (to establish a TLS channel and execute the distancy-bounding protocol). The initial proximity verification takes approximately 25 ms.

2. **Operational latency and data overhead.** The operation latency is defined as the additional latency our solution adds to normal communication from the remote verifier to the attested enclave. The operational latency is also minimal. Our solution 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 requires around 6.48 KBytes/s of data which is only 0.0011% of the USB 2.0 channel capacity.

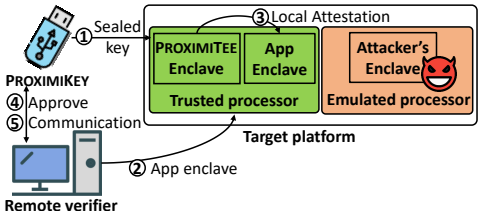

Figure 12: ProximiTEE boot-time attestation. After the boot-time initialization (refer to Figure 13) the ProximiTEE enclave executes a local attestation with the verifier uploaded application-specific enclave.

5 ADDRESSING EMULATION ATTACKS

In this section, we consider a stronger adversary model that we call the *emulation attacker* and present a hardened attestation solution based on boot-time initialization.

5.1 Emulation Attack

Adversary model. The emulation attacker has all the capabilities of the relay attacker (cf. Section 2) and additionally has obtained at least one valid (i.e., not revoked by Intel) attestation key from any SGX platforms but the target platform. The adversary might obtain an attestation key by attacking one of his processors or by purchasing an extracted key from another party.

We consider key extraction from SGX processors difficult and expensive, in contrast to the previously considered relay attacks that require only OS control, but not impossible. The recently demonstrated Foreshadow attack [14] that exploited the Meltdown vulnerability [28] showed how to extract attestation keys from SGX processors. Intel has the possibility to issue microcode patches that address processor vulnerabilities like Meltdown and the processor's microcode version is reflected in the SGX attestation signature (see Appendix A). However, new vulnerabilities may be discovered and before microcode patches are deployed, leaked but not revoked attestation keys may be available.

The emulation attack. In the attack, the adversary uses a leaked attestation key to emulate an SGX-processor on the target platform. Since the IAS successfully attests the emulated enclave, it is impossible for the remote verifier to distinguish between the emulated enclave and the real one.

Emulation attack implications. The emulation attack allows the adversary to fully control the attested execution environment and thus break the two fundamental security guarantees of SGX, enclave's data confidentiality and code integrity, and to access any secrets provisioned to the emulated enclave. Since the OS is also under the control of the attacker, any attempted communication with the real enclave will always be redirected to the emulated enclave.

5.2 Boot-Time Initialization Solution

Proximity verification alone cannot protect against the emulation attacker, as the locally emulated enclave would pass the proximity test. Therefore, we describe a second hardened attestation mechanism that leverages secure boot-time initialization and is designed to prevent emulation attacks. This solution can be seen as a *novel*

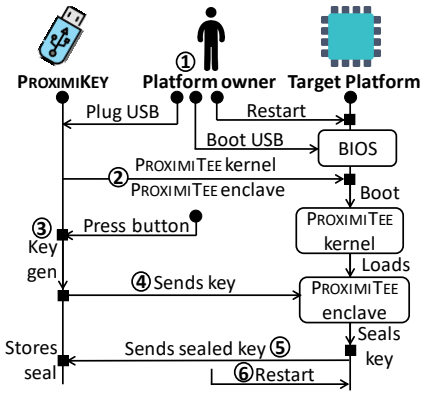

Figure 13: Boot-time initialization. The ProximiKey uses a minimal kernel Linux image to boot and load ProximiTEE enclave on the target platform and seal a platform specific secret to the ProximiKey memory.

variant of the well-known TOFU principle and the main benefits of our solution over previous variants is that it simplifies deployment and increases security. Additionally, when such attestation is used in combination with our previously described periodic proximity verification, our solution enables secure offline revocation.

Security assumptions. Our security assumptions regarding the target platform are as described in Section 2. The only difference is that in this case we assume that the UEFI (or BIOS) on the target platform is trusted.

Solution overview. Figure 12 illustrates an overview of this solution. During initialization, that is depicted in Figure 13, the target platform is booted from the attached device that loads a minimal and single-purpose ProximiTEE kernel on the target device. In particular, this kernel includes no network functionality. The kernel starts the ProximiTEE enclave, which 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 is performed only once. This process is depicted in Figure 13 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.

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 OS 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 12 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. Appendix A) 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.

Following communication. Similar to our previous solution, 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.

5.3 Security Analysis

In this attestation mechanism, the task of establishing a secure communication channel to the correct enclave can be broken into three subtasks. The first subtask is to establish a secure channel to the correct ProximiKey device. In our solution, this is achieved using standard device certification. Recall that the adversary cannot compromise the specific ProximiKey used.

The second subtask is to establish a secure communication channel from ProximiKey to the ProximiTEE enclave. 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.

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. An adversary in possession of leaked sealing attestation keys from

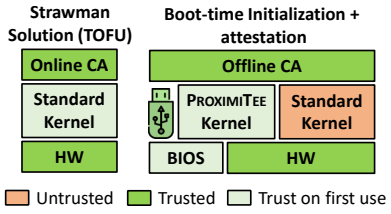

Figure 14: TCB comparison. Trusted components in a common TOFU solution and our boot-time solution.

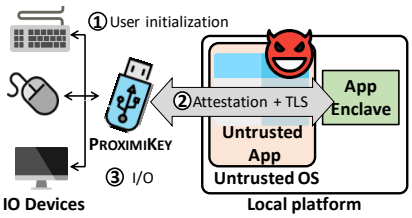

Figure 15: Trusted path to local enclave. The IO devices are connected to ProximiKey that is connected to the local platform. The ProximiKey performs attestation using one of our mechanisms and then mediates all IO communication.

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.

5.4 Comparison to TOFU

Our second attestation mechanism is a novel variant of the well-known "trust on first use" principle. In this section we briefly explain the main benefits of our solution over common TOFU variants.

Smaller TCB size and attack surface. Figure 14 illustrates a comparison of trusted components and attack surface between a common TOFU solution where a trusted authority (CA) certifies enclave keys (cf. Section 2.3) and our boot-time initialization mechanism. In the TOFU solution, the standard and general-purpose OS needs to be trusted on first use and the CA needs to remain online for enrollment of new SGX platforms. In our solution, a significantly smaller and single-purpose kernel needs to be trusted on first use. Additionally, we require trust on the BIOS (or UEFI). In our solution, the CA can remain offline when a new platform is enrolled.

Reboot instead of re-install. Our solution requires that the target platform is rebooted once from ProximiKey. In most TOFU solutions, the target platform requires a clean state which is difficult to achieve without reinstall that makes deployment difficult.

Secure offline revocation. When boot-time initialization is combined with the previously explained periodic proximity verification, our solution provides an additional property of secure offline revocation that requires no interaction with the CA. Such property is missing from previous TOFU solutions.

5.5 Implementation

We implemented a complete prototype of our second attestation mechanism. On top of our previous ProximiTEE implementation (see Section 4.1), the boot-time initialization solution requires the ProximiTEE kernel. We have modified an image of Tiny Core Linux [8] and used it as the boot image for our boot-time initialization. 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. The ProximiTEE enclave is a minimal enclave that uses a simple serial library to communicate with the ProximiKey and local attestation mechanism to attest any application-specific enclave.

6 BUILDING TRUSTED PATH

Another important 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.

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

The commonly suggested solution for the trusted path problem is to leverage a trusted hypervisor to mediate all I/O [42]. 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 [38], 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 it, and 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.

Our approach. In this section, we explain how our attestation mechanisms can be used to build a trusted path between the user and an enclave. Our main idea is to use the ProximiKey device as a *bridge* that attests the local enclave and then securely mediate all user inputs and outputs between I/O devices and enclaves, as shown in Figure 15. Trusted path can be realized using either of our two attestation mechanisms as a building block.

For trusted path, we require that the ProximiKey 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

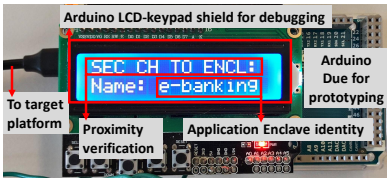

Figure 16: Trusted path implementation. Arduino Due prototype board that is used as an I/O hub with a small LCD display to show the identity of the SGX enclave.

the embedded device is either (i) pre-installed with a list of human-readable names for enclave code measurements, or (ii) it can obtain certified mappings from the platform that it is connected to, similar to property-based attestation [35].

We assume that the user activates the trusted path by selecting the enclave with which she wishes to communicate using a button on the device and the enclave name is shown on the device screen.

Local trusted path. Now, we describe the process of establishing a trusted path to an enclave on a *local* platform. As shown in Figure 15, 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 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 mechanisms. 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.

In Appendix B we explain how such trusted path can be extended to an enclave that resides on a *remote* platform.

Implementation. Figure 16 shows our trusted path implementation that is based on a Arduino Due prototyping board. We implement a minimal prototype that handles keyboard communication using Ardubnio's native `keyboardcontroller` library that intercepts keyboard traffic. All the keystrokes are relayed to the application-specific enclave over the TLS channel. We additionally attach a LCD shield on the Arduino board that shows the enclave identity (name) with which it is currently communicating. This identity is obtained from a certificate that is embedded to the enclave.

7 DISCUSSION

Faster future adversaries. We implemented ProximiTEE on a commercially available and easily programmable Arduino Due hardware using USB 2.0. Such relatively slow prototype shows that proximity verification works securely for SGX enclaves against a fast adversary (1m Ethernet, 120 or 72 μs network latency).

Other networking technologies, such as the InfiniBand [4], could enable even faster adversaries. To address such attacks, the ProximiKey device could be implemented using a USB 3.0 which supports latencies in the order of 5μs or Thunderbolt 3 interfaces which uses direct memory access and communicates over PCI Express. Such ProximiKey device would provide significantly faster responses compared to InfiniBand networks. In general, as long as the local transfer technology is faster than the remote one and the computational overhead of responses is minor compared to communication, distinguishing relay attacks remains possible.

Extension to other TEEs. Our approach could be applied to other TEEs as well. The critical requirements for the TEE is that it must support programmable operations that can be executed sufficiently fast. One TEE that meets these requirements is ARM TrustZone.

8 RELATED WORK

TPM proximity verification. Parno has argued that TPM proximity verification using distance bounding is not secure, as TPM identification operations (signatures) take at least half a second which makes it difficult to reliably distinguish a relayed protocol run from a legitimate one [33]. Despite such compelling reasoning, previous literature has proposed to use distance-bounding protocols for identification of local TPM [20]. However, the provided evaluation is based on a software TPM emulator [1, 7]. Another paper has suggested equipping TPMs with NFC interfaces for secure connection establishment [40], but such solutions are hard to deploy in practice.

DRTM proximity verification. Presence attestation [45] enables proximity verification of a TEE that is created using dynamic root of trust (DRTM) [30]. The DRTM-based TEE shows an image that is captured by a camera and then communicated to a remote verifier in a small time interval. The same approach cannot not be applied to SGX that lacks trusted path for integrity-protected image output. Additionally, the assumed attacker model is weaker than ours, as emulation attacks with leaked keys cannot be prevented.

Trusted path. SGXIO [42] provides trusted path to Intel SGX using a trusted hypervisor. SGXIO uses seL4 [38] 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 make deployment difficult in practice.

UTP [19] describes a unidirectional trusted path from the user to a remote server using dynamic root of trust based on Intel's TXT technology [30, 32]. 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 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. [46] 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 [47] 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.

9 CONCLUSION

In this paper we have analyzed attestation redirection and shown that it increases the adversary's ability to attack an attested SGX enclave. We have proposed ProximiTEE, a novel solution against relay attacks using proximity verification and a trusted embedded device that is attached to the target platform. Our experimental evaluation is the first to show that proximity verification is secure and reliable for SGX and the performance overhead of such verification is minor. As additional contributions, we have also presented a novel boot-time initialization solution for addressing a stronger emulation attacker and explained how our attestation mechanisms can be used for building a trusted path for SGX.

REFERENCES

[1] 2008. TrouSerS. https://sourceforge.net/projects/trousers/.
[2] 2017. Arduino Cryptography Library. https://rweather.github.io/arduinolibs/crypto.html.
[3] 2017. Intel SGX homepage. https://software.intel.com/en-us/sgx.
[4] 2018. Infiniband Trade Association. https://www.infinibandta.org/.
[5] 2018. Local Attestation. https://software.intel.com/en-us/sgx-sdk-dev-reference-local-attestation.
[6] 2018. Local (Intra-Platform) Attestation. https://software.intel.com/en-us/node/702983.
[7] 2018. Software TPM Introduction. http://ibmswtpm.sourceforge.net/.
[8] 2018. Tiny Core Linux, Micro Core Linux, 12MB Linux GUI Desktop, Live, Frugal, Extendable. https://distro.ibiblio.org/tinycorelinux/.
[9] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. OBFSCURO: A Commodity Obfuscation Engine on Intel SGX. NDSS.
[10] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. 2016. SCONE: Secure Linux Containers with Intel SGX.. In *OSDI*.
[11] Stefan Brands and David Chaum. 1993. Distance-Bounding Protocols. In *EUROCRYPT '93*, Tor Helleseth (Ed.).
[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX WOOT17*.
[13] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: confidential ZooKeeper using intel SGX. In *Middleware 2016*.
[14] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security 18*.
[15] Chandler, Matt, and Intel. 2017. Intel Enhanced Privacy ID (EPID) Security Technology. https://software.intel.com/en-us/articles/intel-enhanced-privacy-id-epid-security-technology.
[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086.
[17] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. 2018. Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018, 2 (2018), 171–191.
[18] Atanas Filyanov, Jonathan M McCuney, Ahmad-Reza Sadeghiz, and Marcel Winandy. 2011. Uni-directional trusted path: Transaction confirmation on just one device. In *IEEE/IFIP DSN 2011*.
[19] A. Filyanov, J. M. McCuney, A. R. Sadeghiz, and M. Winandy. 2011. Uni-directional trusted path: Transaction confirmation on just one device. In *IEEE/IFIP DSN 2011*.
[20] Russell A. Fink, Alan T. Sherman, Alexander O. Mitchell, and David C. Challener. 2011. Catching the Cuckoo: Verifying TPM Proximity Using a Quote Timing Side-Channel. In *Trust and Trustworthy Computing*.
[21] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic analysis: Concrete results. In *International workshop on cryptographic hardware and embedded systems*. Springer, 251–261.
[22] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Adi Shamir, and Eran Tromer. 2016. Physical key extraction attacks on PCs. *Commun. ACM* 59, 6 (2016).
[23] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*.
[24] Simon Johnson and Intel. 2017. Intel SGX: EPID Provisioning and Attestation Services. https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services.
[25] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (2018). arXiv:1801.01203
[26] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security 2017*.
[27] Linux. 2017. ping(8) - Linux man page. https://linux.die.net/man/8/ping.
[28] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (2018).
[29] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution.. In *USENIX Security 17*.
[30] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for TCB minimization. In *ACM SIGOPS Operating Systems Review*.
[31] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer.
[32] Cong Nie. 2007. Dynamic root of trust in trusted computing. In *TKK T1105290 Seminar on Network Security*. Citeseer.
[33] Bryan Parno. 2008. Bootstrapping Trust in a" Trusted" Platform.. In *HotSec*.
[34] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution.. In *USENIX Security Symposium*. 431–446.
[35] Ahmad-Reza Sadeghi and Christian Stüble. 2004. Property-based Attestation for Computing Platforms: Caring About Properties, Not Mechanisms. In *Proceedings of the 2004 Workshop on New Security Paradigms*.
[36] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. 2017. ZeroTrace: Oblivious memory primitives from Intel SGX. In *Symposium on Network and Distributed System Security (NDSS)*.
[37] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE S&P 2015*.
[38] seL4. 2017. seL4/seL4. https://github.com/seL4/seL4.
[39] Adi Shamir and Eran Tromer. 2004. Acoustic cryptanalysis. *presentation available from http://www. wisdom. weizmann. ac. il/ tromer* (2004).
[40] Ronald Toegl. 2009. Tagging the Turtle: Local Attestation for Kiosk Computing. In *Advances in Information Security and Assurance*, Jong Hyuk Park, Hsiao-Hwa Chen, Mohammed Atiquzzaman, Changhoon Lee, Tai-hoon Kim, and Sang-Soo Yeo (Eds.).
[41] Zhenghong Wang and Ruby B Lee. 2006. Covert and side channels due to processor architecture. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE.
[42] Samuel Weiser and Mario Werner. 2017. SGXIO: Generic Trusted I/O Path for Intel SGX *(CODASPY '17)*.
[43] Dan Wendlandt, David G. Andersen, and Adrian Perrig. 2008. Perspectives: Improving SSH-style Host Authentication with Multi-path Probing. In *USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association.
[44] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P 2015*.
[45] Zhangkai Zhang, Xuhua Ding, Gene Tsudik, Jinhua Cui, and Zhoujun Li. 2017. Presence Attestation: The Missing Link in Dynamic Trust Bootstrapping. In *CCS '17*.
[46] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. 2012. Building Verifiable Trusted Path on Commodity x86 Computers. In *IEEE S&P 2012*.
[47] Z. Zhou, M. Yu, and V. D. Gligor. 2014. Dancing with Giants: Wimpy Kernels for On-Demand Isolated I/O. In *IEEE S&P 2014*.

A SGX BACKGROUND

In this appedix we provide background on SGX architecture, attestation, leakage and software updates.

SGX basics. Intel's SGX is a TEE architecture that isolates application enclaves from all other software running on the system, including the privileged OS [16]. Enclave's data is encrypted and integrity protected whenever it is moved outside the CPU chip. The untrusted OS is responsible for the enclave creation. Initialization actions taken by the OS to start enclaves are recorded securely inside the CPU, creating a *measurement* that captures the enclave's code. 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 developers must typically 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 [5, 6]. 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. 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 processors microcode version is included to the signed attestation response.

The attestation key is a part of a group signature scheme called EPID (Enhanced Privacy ID) [15] 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 [3].

Limitations and vulnerabilities. Recent research has demonstrated that the SGX architecture can be susceptible to side-channel leakage. Secret-dependent data and code access patterns can be observed by monitoring shared physical resources such as CPU caches [12, 23, 31] or the branch prediction unit [26]. The OS can also infer enclave's execution control flow or data accesses by monitoring page fault events [44]. 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.

The recently discovered vulnerabilities Spectre [25] and Meltdown [28] allow application-level code to read memory content of privileged processes across separation boundaries by exploiting subtle side-effects of speculative execution. The Foreshadow attack [14] demonstrates how to extract SGX attestation keys from processors by leveraging the Meltdown vulnerability.

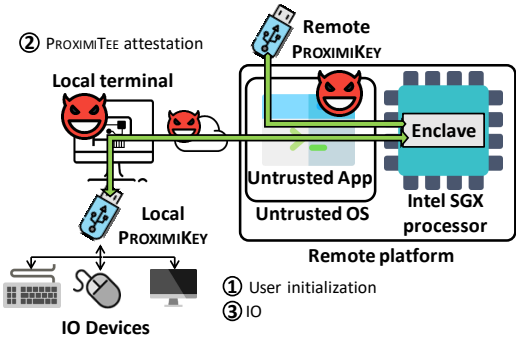

Figure 17: Trusted path to remote enclave. This setup uses two embedded devices. The local ProximiKey is connected to the local platform and the remote ProximiKey is connected with the remote platform.

Microcode updates. During manufacturing each SGX processor is equipped with hardware keys. When SGX software is installed on the CPU for the first time, the platform runs a provisioning protocol with Intel. In this protocol, the platform uses one of the hardware keys to demonstrates that it is a genuine Intel CPU running a specific microcode version and it then then joins a matching EPID group and obtains an attestation key [24].

Microcode patches issued by Intel can be installed to processors that are affected by known vulnerabilities such as the above mentioned Foreshadow attack. When a new microcode version is installed, the processor repeats the provisioning procedure and joins a new EPID group that corresponds to the updated microcode version and obtains a new attestation key which allows IAS to distinguish attestation signatures that originate from patched processors from attestation signatures made by unpatched processors [24].

B TRUSTED PATH TO REMOTE ENCLAVE

In this appendix we describe how a *local* trusted path, described in Section 6, can be extended to an enclave that resides on a *remote* platform, as shown in Figure 17. 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 follows:

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 mechanisms. 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.

C DETAILS ON PARAMETER SETTING

In this appendix we provide further details on how we find suitable parameter values for initial and periodic proximity verification.

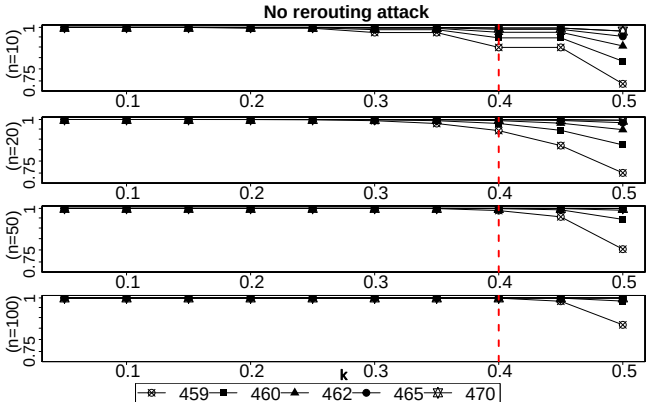

Figure 18: Effect of different threshold latencies (T_{con}). The figure shows the success probability when no relay attack takes place. The threshold latency $T_{con} = 470\ \mu s$ reaches to 0.999999965 success probability for number of trials at least 20 ($k.n,\ k = 0.4$) out of $n = 50$ challenge-response protocol.

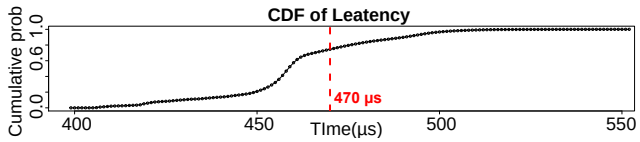

Figure 19: Cumulative distribution function for latencies. We set the threshold T_{con} at 470 μs which has a cumulative probability of 0.75 in the experiment where no rerouting attack takes place with an extremely low probability (9.73×10^{-5}).

C.1 Initial Proximity Verification

In Section 4.4, we outlined a three-step approach to determine suitable parameter values for proximity verification. Here, we provide further details on each of these steps.

1. 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 3) 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 8). 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.

Figure 18 shows the legitimate success probability P_{legit} for different number of rounds ($n \in \{10, 20, 50, 100\}$). We iterate through multiple threshold times ($T_{con} \in \{459\mu s, 460\mu s, 462\mu s, 465\mu s, 470\mu s\}$), and $470\mu s$ provides high success ratio for different values of k ($P_{legit} = 0.9\{7\}65$[2] ($n = 50$) and $P_{legit} = 0.9\{13\}71$ ($n = 100$)). We chose to test T_{con} up until $470\mu s$ because as can be observed in figure 8 for these values we observe extremely small occurrences (9.73×10^{-2}) of latency responses during an attacking scenario. It

[2] $0.9\{n\}x$ denotes $0.n$-times 9 followed by x.

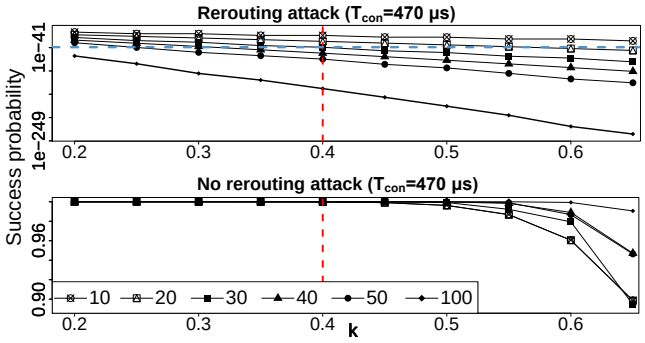

Figure 20: Finding suitable fraction k. The graph shows the legitimate enclave's success probability in an ideal scenario and the attacker's success probability in rerouting attack scenario with varying k.

is possible to increment the latency further to improve the success probability, 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 $p_c = \Pr[396 \leq x \leq 470] = \sum_{i=396}^{470} \Pr[x = i] = 0.75$ (where 396 μs is the smallest latency experienced).

The attacker's success probability for a single round is the cumulative probability sampled from the attacker's distribution (the grey histogram in Figure 8) $p_{\mathcal{A}} = \Pr[x \leq 470] = \sum_{i=451}^{470} \Pr[x = i] = 9.73 \times 10^{-5}$.

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 470 μs out of n. We can write this cumulative probability as a binomial distribution:

$$\Pr[x \geq nk] = \sum_{i=nk}^{n} \binom{n}{i} (p)^i (1-p)^{n-i};\ \text{where } p \in \{p_{\mathcal{H}}, p_{\mathcal{A}}\}$$

2. 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 20 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 $T_{con} = 470\mu s$. We observe a sharp decrease in the legitimate success probability at $k = 0.4$. Hence, fix $k = 0.4$ to achieve the maximum P_{legit}. Additionally, in the graph of attacker's success probability, the blue horizontal line is placed at $10^{-40} \approx 2^{-133}$. Hence we propose to choose any round configuration bellow this horizontal line, where $n \geq 25$. With number of rounds set to $n = 50$ and $k = 0.4$, we have $P_{legit} = 0.999999965$ and $P_{adv} = 2.71 \times 10^{-67}$. Similar result could be also observed in Figure 20 where the success probability of the legitimate enclave decreases significantly after $k = 0.55$ for $T_{con} = 470\mu s$.

3. Generalizing the number of rounds n. Figure 10 extends this analysis to the general number of challenge-response rounds spanning from $n = 2$ to 100 where the attacker uses `ping` in food mode and achieves 153μs average network latency. Here we compute the probability of attacker returning the reply within 470μs for at least $k = 0.4$ 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 $k = 0.4$). 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.

C.2 Periodic Proximity Verification

In Section 4.5 we outlined a three-step approach for finding suitable parameters for the periodic proximity verification that we use for revocation. Here, we provide further details on each of these steps.

1. Finding suitable threshold T_{detach}. We set the threshold T_{detach} to 510 μs. We choose this value as we experience zero sample from the timing distribution (refer to the 'yellow' distribution Figure 8) where no rerouting attack takes place. While in the attacker's distribution, the cumulative probability of the response occurring between T_{con} and T_{detach} is $Pr[T_{con} \leq \mathcal{L}_A \leq T_{detach}] = \sum_{i=451}^{510} \Pr[\mathcal{L}_A = i] = 1.4 \times 10^{-2}$. Using T_{detach}, we can now define the challenge response rounds in Figure 6 for a *single round* as following:

$$\Pr[\mathcal{L}_{legit} \leq T_{con}] = \Pr[legit \in \text{green}] = 0.75$$
$$\Pr[T_{con} < \mathcal{L}_{legit} < T_{detach}] = \Pr[legit \in \text{yellow}] = 0.237$$
$$\Pr[\mathcal{L}_{legit} \geq T_{detach}] = \Pr[legit \in \text{red}] = 7.09 \times 10^{-3}$$
$$\Pr[\mathcal{L}_A \leq T_{con}] = \Pr[A \in \text{green}] = 9.73 \times 10^{-5}$$
$$\Pr[T_{con} < \mathcal{L}_A < T_{detach}] = \Pr[A \in \text{yellow}] = 1.4 \times 10^{-2}$$
$$\Pr[\mathcal{L}_A \geq T_{detach}] = \Pr[A \in \text{red}] = 0.985$$

2. 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 20 out of 50 ($k = 0.4$) challenge-response round where responses are within 470 μs, ProximiTEE yields success probabilities as the following:

$$\Pr[A \in \text{success window}] = P'_{adv} = P'_{fn} = 2.71 \times 10^{-67}$$
$$\Pr[A \in \text{failed window}] = \Pr[A \in \text{red}]^2 = 0.970$$
$$\Pr[legit \in \text{success window}] = 0.999999965$$
$$\Pr[legit \in \text{failed window}] = P'_{fp} = \Pr[legit \in \text{red}]^2 = 5 \times 10^{-5}$$

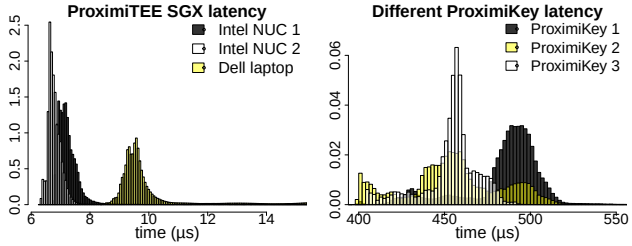

Figure 21: Different target platforms/ProximiKey. We evaluates latencies using three different SGX platforms. The Intel NUCs were few microseconds faster. Additionally, we evaluated latencies using three different Arduino boards. The latencies are consistent.

The probability that a halt window event occurs for a legitimate application-specific enclave running on the platform in proximity is $\Pr[legit \in \text{red}] \approx 7.09 \times 10^{-3}$. The ProximiKey halts all the data communication to the target platform until the next periodic proximity verification.

If two or more than two latencies $\geq 510\mu s$ (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 around 2 minutes.

3. 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 (by reading out the noise of the analog pins of the Arduino board) 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 $\sim$ 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.

D ADDITIONAL EXPERIMENTAL RESULTS

In this appendix we provide results from additional experiments.

We evaluated the consistency of measured latencies across different computing platforms. Figure 21 shows the frequency distribution of latencies across three SGX platforms and three ProximiKey devices. 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 22 shows (on the right) that the USB cable has very small effect on the latency (around 10 μs average difference). It also shows (on the left) no significant differences between the different Ethernet cable lengths.

Figure 24 shows the challenge-response timing trace that we captured from the oscilloscope.

Effects of core pinning. We executes the ProximiTEE enclave application pinning to specific CPU cores (using the command `taskset [COREMASK] [EXECUTABLE]`). Core pinning forces the operating system to use a specific set of CPU core(s) to execute

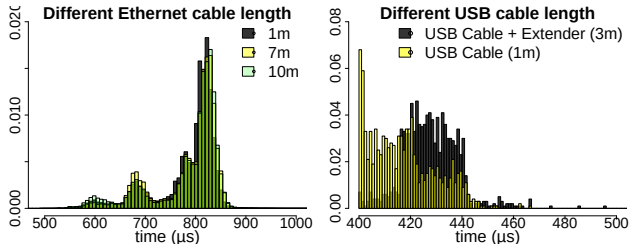

Figure 22: Different Ethernet/USB cables. We evaluated latencies two different USB cables: one with an USB cable (1m) and another with an USB extender of length 2m attached. Additionally, we evaluated latencies using three different Ethernet cables (1, 7 and 10 m). Latencies are consistent. Note, that the latency is sampled in the experiment conducted with non-ping flood mode.

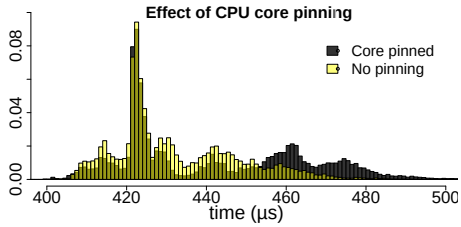

Figure 23: Effect of CPU core pinning on the running enclave application. We evaluated the effect of core pinning. Core pinning restricts the application running on a specific core, eliminating the chance of switching CPU cores by the OS. Switching of the CPU cores has very negligible effect on the latency as shown by the figure.

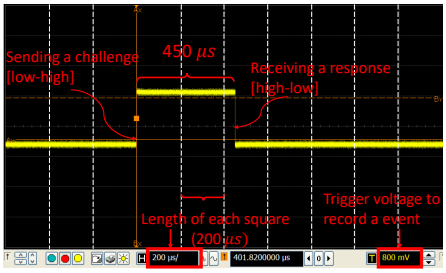

Figure 24: Latency measurement on oscilloscope. We use a high precision oscilloscope to measure the latency of ProximiTEE. The figure annotates the sending and the receiving of the challenge response messages that is activated by the GPIOs on the Arduino board.

a program. CPU pinning may significantly bring down execution time due to the elimination of core switching and ability to reuse L1 and L2 cache. Figure 23 illustrates the effect of CPU core pinning vs. no pinning. We experience negligible effect by core pinning. Hence we conclude that the attacker won't gain any advantage by CPU core pinning.

Effects of CPU load. Figure 25 and 26 shows the challenge response latencies and the enclave execution times respectively with varying degree of CPU stress testing. We used `stress-ng` to stress different number of CPU cores. We experienced a minor slowdown with the increasing number of busy CPU cores. But the slowdown is insignificant. For example, as shown in the Figure 26, we experienced a shift of $12\mu s$ when all the 8 CPU cores are busy executing the benchmark software. Also, note that the load introduced by the benchmark is a sustained load on all the CPU cores which is much more demanding for the CPUs compared to the CPU loads introduced by real-life applications. In that scenarios, the deviation would be even lesser.

We conclude that proximity verification for SGX enclaves is reliable even under high system load. In rare cases of extreme system load, proximity verification might fail, but this is an availability concern, but a security threat.

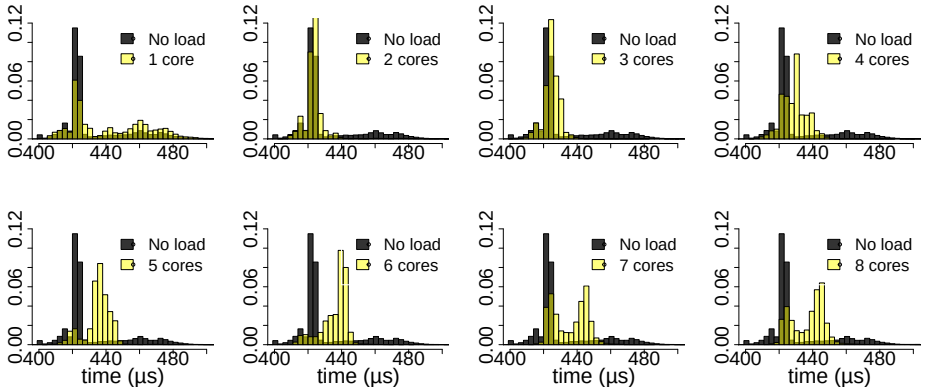

Figure 25: Effect on latency experienced by the ProximiKey with different number of stressed CPU cores. We evaluated latencies while running CPU intensive benchmark on different number of cores. Note that with higher number of busy cores, the means of the distributions start to shift towards right but stayed within $T_{con} = 470\mu s$. We used `stress-ng` Linux stress-testing application.

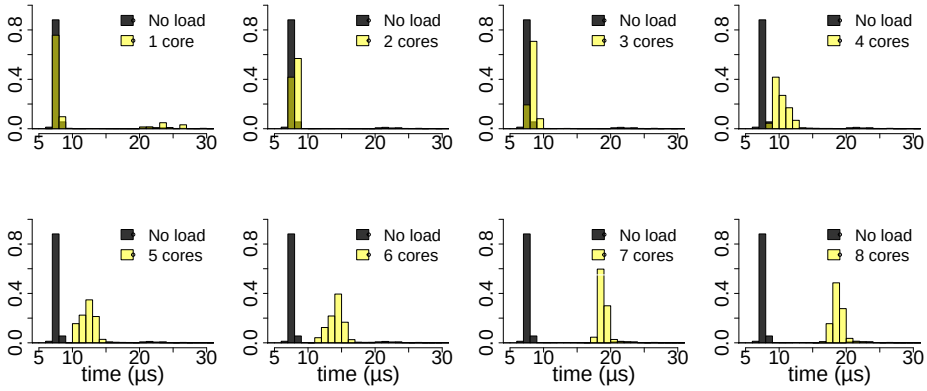

Figure 26: Effect on enclave execution time with different number of stressed CPU cores. We evaluated the enclave running time (computing the challenge-response protocol) while running CPU intensive benchmark on different number of cores.