ARITRA DHAR

BUILDING TRUST IN MODERN COMPUTING PLATFORMS

DISS. ETH NO. 27668

BUILDING TRUST IN MODERN COMPUTING PLATFORMS

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

ARITRA DHAR

Master of Technology in Computer Science & Engineering,
IIIT Delhi, India (2014)

born on 3rd March 1990
citizen of India

accepted on the recommendation of

Prof. Dr. Srdjan Čapkun, examiner
Prof. Dr. Kevin R. B. Butler, co-examiner
Prof. Dr. Ahmad-Reza Sadeghi, co-examiner
Prof. Dr. Adrian Perrig, co-examiner

2021

DOI: 10.3929/ethz-b-000487786

"আমার বৈজ্ঞানিক মনের একটা অংশ আক্ষেপ করছে যে, তাকে ভালো করে স্টাডি করা গেল না, তার বিষয়ে অনেক কিছুই জানা গেল না। সেইসঙ্গে আরেকটা অংশ বলছে যে, মানুষের সব জেনে ফেলার লোভের একটা সীমা থাকা উচিত। এমন কিছু থাকুক, যা মানুষের মনে প্রশ্নের উদ্রেক করতে পারে, বিস্ময় জাগিয়ে তুলতে পারে।"

(A part of my scientific mind is frustrated as 'it' can not be studied more. Many things about it remained unknown. At the very same time, another part of my mind is saying that there should be a limit to the greed of knowing all. There should always be something that provokes curiosity in the human mind, raises wonder.)

– Professor Shonku, The Final Adventures of Professor Shonku
by Satyajit Ray

Dedicated to
The loving memory of Dida, Purnima Dhar,
to whom I owe everything.

ABSTRACT

User interfaces (UI) are essential parts of modern complex computing platforms as it dictates how humans provide inputs to these systems and interpret output from them. Many remote safety and security-critical cyber-physical systems such as industrial PLCs (in manufacturing, power plants, etc.), medical implants are accessible through rich UIs over browsers or dedicated applications that are running on commodity systems or hosts. Similarly, e-banking, e-voting, social networks, and many other remote applications and services are critically dependent on UIs for user authentication and IO. An attacker-controlled host can not only observe user's IO data but also can modify them undetected. Loss of integrity and confidentiality of user inputs can lead to catastrophic failure of critical infrastructures, loss of human lives, leakage of sensitive data. The problem of secure communication between a user and an end-system is known as *trusted path*. Such attacks are not far-fetched as modern software and hardware systems are incredibly complex and span over millions of lines of code. Hence the users are bound to trust a massive trusted computing base or TCB. Exploiting software vulnerabilities of the OS, hypervisors, database systems are very prevalent. Recent technologies such as Trusted execution environments (TEEs) address this problem by reducing the TCB by running isolated environments on the CPU cores, known as enclaves, that are isolated from the OS or hypervisor. However, TEEs do not solve the trusted path problem as TEEs depend on the OS to communicate to the external IO devices. Moreover, the remote attestation mechanism by which a verifier can ensure that she is communicating with the proper enclave is vulnerable to relay attack. In the context of disaggregated computing architecture in modern data centers, the security properties of traditional TEE are insufficient as the trusted path application involves sensitive data not only on the CPU cores but also on the specialized external hardware like accelerators.

In this thesis, we propose mechanisms to build trust in modern computing platforms by addressing the trusted path problem, and we make the following contributions. First, we analyze existing trusted path systems and found several attacks that compromise user IO data integrity and confidentiality. We are the first to analyze the trusted path problem to find a set of essential security properties and implement them in a system named

ProtectION using a trusted embedded device as an intermediary. This trusted device intercepts all IO data and overlays secure UI on the display signal. Next, we look into Intel SGX and investigate how one can integrate a trusted path solution to TEEs. We notice that the relay attack on the SGX remote attestation can be detrimental to the trusted path security properties. We design ProximiTEE, a system that uses distance bounding to verify physical proximity to an SGX processor. We also show how the distance bounding mechanism can be used in a high frequency to allocate or revoke platforms in data centers without relying on an online PKI. Finally, we look into the disaggregated computing model of the modern data centers where the TEEs are insufficient as the computation is no longer limited to the CPU cores but several external devices such as GPUs, accelerators, etc. We propose our system PIE based on RISC-V architecture that combines the enclaves running on the CPU and firmware external hardware to a single attestable domain that we call platform-wide enclaves. Inside these platform-wide enclaves, individual binaries (enclaves and firmware) can be remotely attested.

ZUSAMMENFASSUNG

User interfaces (UI) sind wesentliche Bestandteile komplexer moderner Computerplattformen, da sie vorschreiben, wie Menschen Daten in diese Systeme eingeben und wie sie Ausgaben von den Plattformen interpretieren. Viele sicherheitskritische Cyber-Physical-Remote-Systeme wie industrielle PLC (in der Fertigung, in Kraftwerken usw.) und medizinische Implantate sind über umfangreiche Benutzeroberflächen über Browser oder dedizierte Anwendungen zugänglich, die auf handelsüblichen Systemen oder Hosts ausgeführt werden. In ähnlicher Weise sind E-Banking, E-Voting, soziale Netzwerke und viele andere Remote-Anwendungen und -Dienste entscheidend von UIs für die Benutzerauthentifizierung und IO abhängig. Ein von einem Angreifer kontrollierter Host kann die IO-Daten des Benutzers nicht nur beobachten, sondern auch unbemerkt ändern. Der Verlust der Integrität und Vertraulichkeit von Benutzereingaben kann zu einem katastrophalen Ausfall kritischer Infrastrukturen, zum Verlust von Menschenleben und zum Verlust sensibler Daten führen. Das Problem der sicheren Kommunikation zwischen einem Benutzer und einem Endsystem wird als *Trusted Path* bezeichnet. Solche Angriffe sind nicht weit hergeholt, da moderne Soft- und Hardwaresysteme unglaublich komplex sind und aus Millionen von Codezeilen bestehen. Daher müssen die Benutzer einer riesigen Trusted Computing Base oder TCB vertrauen. Das Ausnutzen von Software-Schwachstellen in Betriebssystemen, Hypervisoren oder Datenbanksystemen ist weit verbreitet. Neuere Technologien wie Trusted Execution Environments (TEEs) lösen dieses Problem durch eine Reduktion der TCB, indem isolierte Umgebungen, die als Enklaven bezeichnet werden, auf CPU-Kernen laufen und vom Betriebssystem oder Hypervisor isoliert sind. TEEs lösen jedoch nicht das Problem des Trusted Path, da TEEs für die Kommunikation mit externen Geräten vom Betriebssystem abhängig sind. Darüber hinaus ist der Mechanismus für Remote Attestation, durch den ein Verifizierer sicherstellen kann, dass er mit der richtigen Enklave kommuniziert, anfällig für Relay-Attacken. Im Kontext einer disaggregierten Rechnerarchitektur in modernen Rechenzentren sind die Sicherheitseigenschaften von traditionellen TEEs unzureichend, da die Trusted-Path-Anwendung sensible Daten nicht nur auf den CPU-Kernen, sondern auch auf spezialisierter externen Hardware, wie z.B. Beschleuniger, beinhaltet.

In dieser Arbeit schlagen wir Mechanismen vor, um Vertrauen in moderne Computerplattformen aufzubauen, indem wir das Trusted-Path-Problem angehen. Dabei leisten wir die folgenden Beiträge: Zuerst analysieren wir bestehende Trusted-Path-Systeme und decken mehrere Angriffe auf, die die Integrität und Vertraulichkeit von Benutzer-IO-Daten gefährden. Wir sind die ersten, die das Trusted-Path-Problem analysieren, um eine Reihe wesentlicher Sicherheitseigenschaften zu finden und diese in einem System namens ProtectIOn unter Verwendung eines vertrauenswürdigen eingebetteten Geräts als Intermediär zu implementieren. Dieses vertrauenswürdige Gerät fängt alle IO-Daten ab und überlagert eine sichere Benutzeroberfläche auf dem Anzeigesignal. Als nächstes schauen wir uns Intel SGX an und untersuchen, wie man eine Trusted-Path-Lösung in TEEs integrieren kann. Wir stellen fest, dass die Relay-Attacke auf die Remote Attestation von SGX die Sicherheitseigenschaften des vertrauenswürdigen Pfads beeinträchtigen kann. Wir entwickeln ProximiTEE, ein System, das Distance Bounding verwendet, um die physische Nähe zu einem SGX-Prozessor zu überprüfen. Wir zeigen auch, wie der Mechanismus des Distance Bounding in hoher Frequenz verwendet werden kann, um Plattformen in Rechenzentren zuzuordnen oder zu widerrufen, ohne auf eine Online-PKI angewiesen zu sein. Schließlich betrachten wir das disaggregierte Rechnermodell moderner Rechenzentren, in denen TEEs nicht ausreichen, da die Berechnung nicht mehr auf CPU-Kerne beschränkt ist, sondern mehrere externe Geräte wie GPUs, Beschleuniger usw. einschliessen. Basierend auf der RISC-V Architektur präsentieren wir unser System PIE, das die Enklaven, die in der CPU und in der externen Firmware-Hardware ausgeführt werden, zu einer einzigen attestierbaren Domäne kombiniert, die wir plattformweite Enklaven nennen. Innerhalb dieser plattformweiten Enklaven kann für einzelne Binärdateien (Enklaven und Firmware) Remote-Attestation durchgeführt werden.

ACKNOWLEDGEMENTS

A few months back, I watched a video where former California governor and former bodybuilder Arnold Schwarzenegger delivered a speech at the graduation ceremony at the University of Huston a few years back. He said that the concept of a self-made man or woman is a myth. No one is truly self-made, as one's journey is supported by countless help and kindness. I could not agree with this statement more, which is why I would like to take some time to thank so many people for my unforgettable Ph.D. journey.

The most significant impact during my Ph.D. was made by my adviser Dr. Srdjan Capkun both professionally and personally. I can not thank him enough for his continuous support and guidance throughout the past years. He gave me immense freedom to explore the topics that I am truly passionate about. He guided me when I did not know what to do with that freedom. Srdjan has a unique mentoring skill, the way he creates bonds with his students always amazes me. I tried to imitate some of his techniques while supervising students. The numerous discussion I had with him, not only about my work but also my personal life, had a profound effect on my perspective on science, and life, in general.

Another person I would like to thank for having an enormous impact on my Ph.D. is Dr. Kari Kostiainen. He was not only a frequent collaborator of almost all my projects, but also he mentored me towards formulating my ideas and giving me feedback on my writing. Kari's structured way of writing papers had a visible impact on all my writings. I remember knocking on his office door on numerous occasions whenever I had some new ideas. Even though most of the ideas did not make it out of his office, some of our papers resulted from these frequent brainstorming sessions. I was always amazed to see no matter how heated the discussions were; he was always calm and composed.

Special thanks to my thesis committee members: Dr. Kevin R. B. Butler, Dr. Ahmad-Reza Sadeghi, and Dr. Adrian Perrig, for their valuable feedback on the thesis.

Dr. Rahul Purandare, my master's thesis adviser at IIIT-Delhi, had a significant influence on me for shaping my career path. Not only he gave me an abundance of freedom to work on my idea for my master's thesis, but he also encouraged me to pursue a research career and supported me enormously. Joining the Ph.D. program at ETH is something that was a

result of his continuous encouragement. Similarly, Dr. Pinaki Sarkar, my bachelor thesis adviser, had an impact on shaping my very early, a naive understanding of research. He took me under his wings to work beside him during his Ph.D. My fond memories of taking long walks with him to brainstorming combinatorics and cryptography are something I cherish. I learned a great deal about critically think about the problems, relentlessly working and shaping ideas, and scientific writing.

In 2016, during the time of moving to Zürich, I was afraid that it would be a stressful affair given how far away I would be from my family and life-long friends. However, I was extremely fortunate to create bonds with so many and had a circle of friends and colleagues that gave me an essential lifeline. Undeniably, these bonds were critical to continue my research and life in Zürich.

First, I would like to thank Mrs. Anne Messinger and her families for their immense kindness and humidity. I spend my first year in Zürich at their home at Wollishofen. They treated me like a family and reminded me that I have a second home here.

During my time in ETH and the System Security group, I was fortunate to meet and become friends with some amazing people. I will start with Mridula Singh, whom I met during my masters at IIIT-Deli and also were my colleagues at Xerox research. Coming to Zürich, she was the only familiar face. Thanks to her and Saurav Bose, we had quite a lot of Indian food dinner sessions. All the weekend trips with them were absolute pleasures. Special thanks to Dr. Siniša Matetić for being such a good friend and tolerating me while I was making fun of him countless times. During my Ph.D. interview, Siniša appeared to be quite a grumpy personality. But getting to know him closely over the last five years revealed that my initial impression could not be more wrong. Thanks to David Sommer for a number of very philosophical discussions. His revolutionary ideas were breaths of fresh air, and our year-long collaboration on CoverUp during the beginning of my Ph.D. was a pleasure. Dr. Esfandiar Mohammadi is another person with whom I collaborated at the beginning of my Ph.D. Special thanks to him for mentoring me in the initial phase of my Ph.D. Special thanks to Dr. Luka Mališa for all the conversions we had. I always found him easy to talk to about my personal struggles, which are always the most difficult conversations. I thank Karl Wüst for such a good friend. Traveling with him and Balz to Iceland was memorable. I was always amused to see how deep knowledge he has about many other things outside computer science. Calling his craft beer knowledge encyclopedic would be an understatement.

Thanks to Daniele Lain for being a good friend and an amazing officemate. Having long conversions over beers and board games at his apartment were some of the most enjoyable times I had. I am also thankful for the authentic Neapolitan pizza recipe that I got from him. Special thanks also to Dr. Thilo Weghorn for being such a good friend. The trip with him to Japan and Taiwan was an amazing experience. Thanks to Dr. Marco Guarnieri for all the conversations and attention to detail while checking the draft of my papers or this thesis. Thanks to Bhargav Bhatt for a good friend and a fantastic office mate for the first two years till I moved my office. Thanks to Daina Romeo for showing me how to cook gnocchi and participating in drawing overly aggressive shouting Pokemons. Big thanks to Dr. Der-Yuan Yu for the collaboration in my earliest idea and for being such a fantastic friend. He was solely responsible for motivating me to come to the super Kondi and weight lifting session at ETH Polyterrasse. Thanks to Moritz and Ivan for being such amazing collaborates and friends. I thank my colleagues Prof. Dr. Aanjhan Ranganathan, Mansoor Ahmed, Dr. Claudio Marforio, Dr. Nikolaos Karapanos, Dr. Hubert Ritzdorf, Mia Felic, Dr. Lara Schmidt, Patrick Leu, Patrick Schaller, Benjamin Rothenberger, Dr. Marc Röschlin, Dr. Enis Ulqinaku, Dr. AbdelRahman M. Abdou, Dr. Arthur Gervais, Dr. Srdjan Krstic, Dr. Carlos Cotrini, and Dr. Ognjen Maric for making my journey memorable. Specifically, I will remember the intense lunch discussion and various science experiments like evaluating the quality of pasta cooked in the microwave oven. Also, thanks to Christopher Signer for all the craft beer and board game sessions and for being such a kind and compassionate person. The Call of Duty gaming sessions with Chris, Daniele, Giovanni, Leonardo Nodari, and Gianluca Lain made the Covid-19 lockdown during mid-2020 tolerable. I express my gratitude to Mrs. Barbara Pfändner and Mrs. Saskia Wolf for all the support and help with administrative matters during the years.

I thank my former flatmate Clara Lovato for being a close friend. Special thanks to Kai Eva Najand for being a good friend and flatmate for the last three and half years, and tolerating my puns, and participating in the intense, heated discussion of science, architecture, and life in general. I owe a great deal to Ria Ghose and Prapti SenGupta for their constant love and support. I am truly grateful to have these two as my friends. Thanks to Dr. Priya Ghose and Dr. Anindya Basu for their friendship and kindness. Thanks to Bruno, Layla, Sofia, Rahel, Lyon, Urs, and Erasmo for making my stay at Zollikerstrasse memorable. The close friends I made during my master's studies at IIIT-Delhi: Sandipan Biswas, Nishan Sharma, Samir

Anwar, Rohit Jain, Rohit Romley, Pankaj Sahu, Gajendra Waghmare, Ganesh Ghongane, Aniya Agarwal, Arpan Jati, Dr. Hemanta Mondal, and Amit Kr. Chauhan and my colleagues from Xerox Research: Abhishek Kumar, Raj Sharma, Kundan, Kuldeep Yadav, and Ranjeet Kumar Kanaily, all played crucial roles in shaping the person I am today.

I cannot be more thankful to my family for a loving home. My Ma, for her boundless love and for being such a huge part in my life. My Baba, for being the biggest influence in my life and being always there for me. Most of my personal traits, love for science, painting, video games, science fiction novels, comics, and countless others, I inherited from him. I am grateful to my parents for raising me, trusting me, and teaching me that value of education is paramount. My paternal grandparents for their love and care. Dida, my late-grandmother, for being such an inspirational woman in my life, and dadu, my late-grandfather, who inspired and shaped my childhood. My late-maternal grandparents for being a crucial part of my childhood. The memories of going to their place during my childhood are still vivid, and so are many holidays in distant places in India. Chod-dida and Chor-dadu for their love and kindness. I fondly remember the weekends I spend at their place on Saturdays after the masters' entrance examination preparation classes. Thanks to my cousin Prithwish Aich for being the person I always look up to.

I cannot finish this section without thanking my girlfriend, Monika Kos, for her unconditional love, kindness, compassion, and support. For standing alongside me in good times and in bad times, giving me hope and strength, especially during those never-ending deadlines. She is my constant source of inspiration and joy.

Finally, *in memoriam* of my grandma, Purnima Dhar to whom I would like to dedicate this doctoral thesis. She always was my biggest champion. Her sacrifices to raise her children in challenging situations, her boundless love towards me teach me to be humble and inspire me to strive to be the best version of myself. Not a single day passes by when I don't miss you. You would have been proud.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

Part I

INTRODUCTION AND BACKGROUND

1

INTRODUCTION

Great things are not done by impulse, but by a series of small things brought together

— Vincent Van Gogh

Interaction between human and complex cyber-physical systems is an essential aspect of modern computing platforms. These interactions enable users to provide inputs to software systems and services and interpret computation output. Over the last half a century [1, 2], researchers in academia and industry spent a significant effort to make this interaction as effortless as possible. Input-output (IO) peripherals and complex user interfaces (UI) are keys to facilitate this interaction. Intuitive UIs were quintessential to the widespread deployment of computing devices around us and the rapid adoption of remote applications and services.

Security and safety-critical remote applications such as e-voting, online banking, industrial control systems (such as remote PLCs [3]), and medical devices [4] rely upon user interaction. This is typically performed through a *host* system that generally is a standard *x*86 platform, which gives the host access to the raw IO data that is exchanged between the user and the remote server. The host consists of large and complex software systems such as the operating system, hypervisor, device drivers, applications such as a browser, and a diverse set of hardware components that expose the host to a large attack surface. Due to cost and convenience, general-purpose PCs are prevalent in many safety-critical application domains such as industrial plants and hospitals. For example, the WannaCry ransomware incident showed that NHS hospitals relied on Windows XP platforms [5, 6] despite the fact that Microsoft ended the mainstream support in 2009 [7]. During the current Covid-19 pandemic, we have seen a steady rise of working from home [8]. In such a scenario, most of the users access these remote systems through their standard desktops and laptops.

1.1 TRUSTED PATH

Trusted path provides a secure channel between the user (specifically through the human interface devices - HIDs) and the end-point, typically a trustwor-

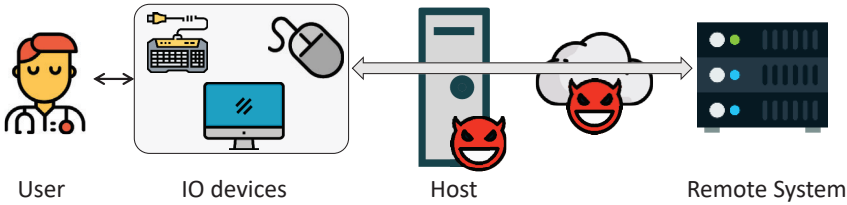


FIGURE 1.1: **Remote trusted path through untrusted host.** The user interacts with a remoter service (that is trusted) with her trusted IO peripherals through the untrusted host and network. A remote trusted path must ensure the integrity (and in some application scenarios, confidentiality) of the data between the user and the remote server.

thy application running on the host. A trusted path ensures that user inputs reach the intended application unmodified, and all the outputs presented to the user are generated by the legitimate application. Hence, a trusted path ensures *IO integrity*. In certain scenarios, the trusted path also provides *confidentiality* to the user data. Trusted path can also be extended to general peripherals such as accelerators, GPUs, and sensors, and even remote systems. Trusted path could be extended to a remote system where the user communicates with a remote end-point using the local host as an untrusted transport. An attacker who exploits the software stack, i.e., the hypervisor or the OS and/or the hardware (refer to Figure 1.1), can control the user's computer. Such an attacker can *observe* and *modify* any user interaction data without being detected by the user or the server, hence undermining the trusted path's properties.

Trusted path to the local host is a well-researched area where many solutions focus on using trusted software components such as a trusted hypervisor [9] or external trusted hardware [10, 11, 12, 13, 14]. However, hypervisors are hard to deploy, have a large TCB, and are impractical in real-world scenarios as most of the existing verified hypervisors offer a minimal set of features, and some of the external device-based approaches such as transaction confirmation devices [10, 11] suffer from poor usability, security issues due to user habituation [15] and are only limited to simple inputs.

1.2 TRUSTED EXECUTION ENVIRONMENTS (TEE)

Trusted execution environments (TEEs) such as Intel SGX, AMD SEV, ARM TrustZone, RISC-V keystone, etc., drastically reduce the trusted computing base (TCB) and provide security to applications, known as enclaves, without having to trust the operating system and hypervisor [16, 17, 18]. Thus, the attack surface is reduced by eliminating two of the largest sources of vulnerabilities for a system [19, 20]. TEEs use *isolation* primitives provided by the CPU to exclude all software but a single target application (commonly known as an enclave) from the software trusted computing base (TCB). Only the CPU is part of the hardware TCB, while the remaining hardware in the system is considered untrusted. Even memory is not included in the TCB and can only be used in conjunction with memory encryption and integrity protection. Such a trust model makes the TEEs ideal candidates for the aforementioned trusted path applications where the software stack is attacker-controlled.

Apart from isolation of enclaves from the untrusted OS, another crucial security primitive of TEEs is *remote attestation*. A remote verifier can ensure that a (legitimate) platform is running an enclave with the correct code. In a simplified form, at the end of a remote attestation process, the platform generally[1] produces three following pieces of information to the verifier:

1. A *measurement* (cryptographic hash of the code and data) of the enclave to ensure that it is running the proper enclave code. The measurement is signed (called *report*) by the platform's attestation key that is physically bound to the platform.

2. A successful verification of the signed measurement[2] ensures that the attestation key originates from a *legitimate platform*.

3. The *public key* of the platform that the remote verifier can use to establish a secure channel to provision secrets to the enclave.

RELAY ATTACK ON REMOTE ATTESTATION. While SGX's 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. An attacker that controls the software stack (OS, hypervisor, etc.)

1 There exist some differences between remote attestation mechanisms in different TEE technologies. We use Intel SGX as an example here, and more details are provided in Section 2.2.1. However, other approaches to remote attestation are comparable.

2 The verification is carried out by Intel Attestation Server.

on the target platform can relay incoming attestation requests to another platform. This way, the user ends up attesting to the attacker's platform rather than her own. Relay attack enables the attacker to see all the IO data to and from the user, e.g., sensitive user input on display, and execute a long-term physical side-channel attack. Hence, relay attacks on Intel SGX remote attestation pose a real threat to trusted paths. Relay attack is not only limited to Intel SGX but also other comparable TEEs such as AMD SEV or RISC-V keystone. Note that the relay attack is a long-standing open problem in trusted computing, as already a decade ago, Parno identified it in the context of TPM attestation [21]. Hence, directly incorporating a trusted path to a TEE like Intel SGX is not a trivial issue.

ISSUES WITH TRUSTED PATH. Solving the relay attack still does not make TEEs ideal for a trusted path. TEEs cannot communicate with any external device without going through the untrusted operating system. This results in a static hardware TCB from the enclave's perspective as the enclaves need to trust the OS/hypervisor, all the device drivers running on it, and all the external devices, even in a scenario where the enclave needs to communicate with only one external device. Due to the lack of isolation, a vulnerability in one device driver, may allow the attacker to gain control over other devices[3]. Hence, constructing trusted path using existing TEEs results in a massive increment in both software and hardware TCB, nullifying the core advantage of using TEEs.

STATIC SOFTWARE/HARDWARE TCB. Static software/hardware TCB in TEEs is also critical in the modern data center where trusted path expands beyond simple IO operation. For example, users can deploy an application that uses sensitive medical data for training machine learning models on a GPU. Existing TEEs protect applications (enclaves) running on the CPU cores; however, they do not protect applications running on the external hardware, which in this case is a GPU. More and more applications rely on application-specific hardware devices (also known as accelerators) that are tuned for specific workloads. Such accelerators provide massive speedups over CPUs. The cloud providers move to a new breed of computing model to cope with such workloads and optimize flexible resource allocation[4]. Such a model is known as the *disaggregated* computing model [23]. Unlike

3 This property is also known as *fault isolation*.

4 Resources such as CPU cores, memory, storage, network interface, etc., are allocated to the VM from a pool based on the demand. One such example is Amazon EC2 [22] where storage and network interface is allocated to a VM dynamically.

the traditional CPU-centric computational model, where the CPU is the platform's sole executioner, in modern data centers, CPUs become mere coordinators of these special-purpose devices. The isolation and attestation primitive of traditional TEEs are geared towards the CPU-centric models as enclaves are running solely on the CPU cores, and only the CPU package is trusted. Hence the current TEE primitives such as the isolation and the attestation need redesigning to support the trusted path applications in the modern disaggregated cloud data centers.

1.3 ADDRESSING THE RESEARCH QUESTION: CONTRIBUTION OF THIS THESIS

Given the problem space discussed above, this thesis addresses the a number of open research questions (RQ) concerning the trusted path and trusted execution in modern platforms.

We start by investigating how trusted external devices can aid users in constructing a remote trusted path. Existing external devices-based techniques such as transaction confirmation devices, suffer from an array of security and usability issues. However, we see the clear benefit of external devices as it eliminates the software (OS/hypervisor) and hardware TCB of the host. From this, we arrive to our first research question:

RQ1

How to build trusted path systems that provide integrity (and possibly confidentiality) guarantees without the users relying solutions with high cognitive load such as transaction confirmation devices while maintaining a small TCB?

1.3.1 *Addressing RQ1*

We propose two trusted path systems: IntegriKey in Chapter 3) and IntegriScreen in Chapter 4 to answer our first research question: **RQ1**. IntegriKey uses *input signing* to provide a second factor for the integrity of the keyboard input. IntegriKey leverages a small-TCB trusted embedded device to sign user input. In IntegriKey, we identify a new form of input manipulation attack that we name *field swapping attack* where the attacker can swap the labels of different input fields that accept overlapping values (e.g., blood pressure and heart rate for a medical implant). IntegriKey

analyzes the type of input fields, and based on the regular expression of these fields; it can compute the overlapping fields that could be vulnerable to swapping attack and recommend the user to append a label with the input value to distinguish it.

In comparison, in IntegriScreen, the user leverages a phone camera to extract the information that the user is typing and the UI on the screen by using text recognition to provide a second factor for user intention's integrity - that we call *visual supervision*. This IO information is then sent to the server over a separate communication channel.

Unlike existing trusted path solutions, these proposals rely neither on transaction confirmation devices that put a heavy cognitive load on the user nor on a hypervisor or trusted drivers that introduce a large TCB. The attacker model in IntegriKey and IntegriScreen differs as the latter requires trust on the smartphone for the host screen analysis. Even though both the systems provide a significant improvement over state-of-the-art trusted path solutions, they suffer from similar security and functionality pitfalls as their counterparts. Both IntegriKey and IntegriScreen focus on solving a single problem (either input or output) and a single input source. This leads to multiple sophisticated attacks on them (e.g., character addition/reduction attack, early form submission attack, etc.).

These attacks are possible not only against IntegriKey or IntegriScreen, but many other existing systems. The lack of security properties in these systems leads to the second research question that looks into the trusted path problem just not as a system building problem but instead addressing the fundamental security properties related to user IO operations. From this, we arrive to our second research question:

RQ2

Why a majority of existing solutions failed to provide trusted paths that provides integrity and confidentiality guarantees to the user interactions? What are the fundamental security properties required for a secure trusted path in the presence of an attacker-controlled (both software and hardware) host?

1.3.2 *Addressing RQ2*

The shortcomings of the existing literature provide the groundwork of our system named ProtectIOn in Chapter 5 that answers the second research

question **RQ2**. Here we assume that the entire host (both software stack and the hardware) is attacker-controlled. We also do not assume that host has a TEE-enabled processor. ProtectION is built on the following observations: i) input integrity is possible only when both input and output integrity are ensured simultaneously, ii) all the input modalities are needed to be protected as they influence each other, and iii) high cognitive load results in user habituation errors. ProtectION uses a trusted low-TCB auxiliary device that we call IOHub, which works as a mediator between all user IO devices and the untrusted host. Instead of implementing a separate network interface, the IOHub uses the host as an untrusted transport - reducing the attack surface. ProtectION ensures output integrity and confidentiality by sending an encoded UI to the host that only the IOHub can overlay on the part of the screen. The overlay is possible as the IOHub intercepts the display signal between the host and the monitor. The overlay generated by the IOHub ensures that the host cannot manipulate any output information on that overlaid part of the screen; hence, it can not trick the user. All the user interaction to this protected overlay is encrypted and signed by the IOHub. Therefore input integrity and confidentiality are preserved.

While addressing RQ2 provides the necessary mechanisms to construct a remote trusted path application, porting the same trusted path application into a trusted execution environment (TEE) such as the Intel SGX is still non-trivial due to the relay attack on the SGX's remote attestation. This problem is due to the trust on first use (ToFU) property of SGX remote attestation. From this, we arrive to our next research question:

RQ3

Relay attack from a local platform to a remote attacker-controlled platform is a real threat to the remote attestation of TEEs like Intel SGX. How to ensure that the attacker cannot relay the attestation to an attacker-controlled platform that exposes all the sensitive IO data?

1.3.3 *Addressing RQ3*

In his paper, Parno [21] identified distance bounding as a candidate solution to TPM relay attacks already ten years ago. But he concluded that it could not be realized securely as the slow TPM identification operations (signatures) make a local and relayed attestation indistinguishable. However, the implication of the relay attack was never well-studied. In this thesis, we

investigate the implication of the relay attack in the context of Intel SGX. Compared to TMP, SGX provide concurrent isolated execution environments, making them suitable for running services. LikeTMP, all flavors of Intel SGX remote attestation (EPID, pseudonymous, and DCAP) rely on trust on first use policy – making them susceptible to relay attacks. We show that relay attacks could have an adverse consequence in the context of the trusted path – complete leakage of user IO data. To answer the third research question, **RQ3**, We propose a new solution in Chapter 6, that we call ProximiTEE, that prevents relay attacks by leveraging a simple embedded device that we call ProximiKey which is attached to the attested target platform. The ProximiKey executes a challenge-response protocol between itself and the platform and based on the latency of the protocol, and the ProximiKey determines if it is connected to the target platform physically (over interfaces such as USB, ThunderBolt, etc.) or not. In short, ProximiKey leverages distance bounding protocol to estimate the physical closeness to the target platform. Our solution is best suited (but not limited to) to scenarios where i) the deployment cost of such an embedded device is minor compared to the benefit of more secure attestation, ii) trust-on-first use (ToFU) solutions are not acceptable, and iii) most importantly, in data center scenario, simply plugging in or out the ProximiKey allocate or revoke a specific platform from a data center fleet without explicitly relying on a public key ledger[5]. Attestation of servers at cloud computing platforms and setup of SGX-based permissioned blockchains are two such examples. Note that, in contrast to the earlier research questions (RQ1 and RQ2), we only assume that the software stack and the network is fully attacker-controlled, and only the CPU is trusted.

ProximiTEE hardens the security properties of the remote attestation and makes TEEs more suitable for a trusted path. Moreover, it provides means to allocate and revoke platforms in a data center setting in a fast and reliable way without relying on an online authority. However, it is not enough when we consider the disaggregated computing architecture of modern data centers where the trusted path endpoint could be an application running on special-purpose hardware such as a GPU or accelerator. The disaggregated computing model brings us to our next and last research question:

5 Note that using ProximiTEE to allocate and remove platforms are efficient even compared to the latest Intel DCAP (datacenter attestation primitive) 2.0 attestation revocation as the former relies on an additional online authority: Intel registration server to register platforms

RQ4

How to extend the trusted path mechanism to a modern platform that is interconnected with a number of heterogeneous peripheral devices? These peripherals can range from IO devices to other complex hardware devices such as GPU or accelerator that can execute programs outside the CPU cores. How to ensure the platform-wide integrity guarantee (configuration and interaction of the TEE enclaves and external peripherals) is preserved without relying on a purpose-built system?

1.3.4 *Addressing RQ4*

Finally, to answer **RQ4**, our fourth and final research question, We propose a TEE with a *configurable* software and hardware TCB including arbitrary external peripheral devices, a concept in Chapter 7 that we call *platform isolation environment* (PIE). PIE executes applications in *platform-wide enclaves*, which are analogous to the enclaves provided by TEEs, except that they span several hardware components. For example, a platform-wide enclave can be composed of an output device such as a display, input devices such as keyboard and mouse, the CPU that is executing the program, the GPU that renders the graphics, and the custom code running on them. Like in traditional enclaves, a platform-wide enclave can be remotely attested. However, the PIE attestation not only reports a measurement of the software TCB but also of the hardware components that are part of the platform-wide enclave.

The shift towards configurable hardware and software TCBs has wide-ranging implications concerning integrity, confidentiality, and attestation of a platform-wide enclave. Attestation, for one, should cover all individual components of a platform-wide enclave atomically to defend against an attacker that changes the configuration in between attestations to separate components. Moreover, the untrusted OS may remap the peripheral devices at runtime with an untrustworthy device, which should not receive access to sensitive data. We carefully design PIE to not be vulnerable to such attacks and present an in-depth security analysis.

We mitigate the above-mentioned attacks with two new properties of platform-wide enclaves: *platform-wide attestation* and *platform awareness*. Platform-wide attestation expands the attestation to cover all components within a platform-wide enclave, and platform awareness enables enclaves to react to changes in their ecosystem, i.e., remapping by the OS. We achieve

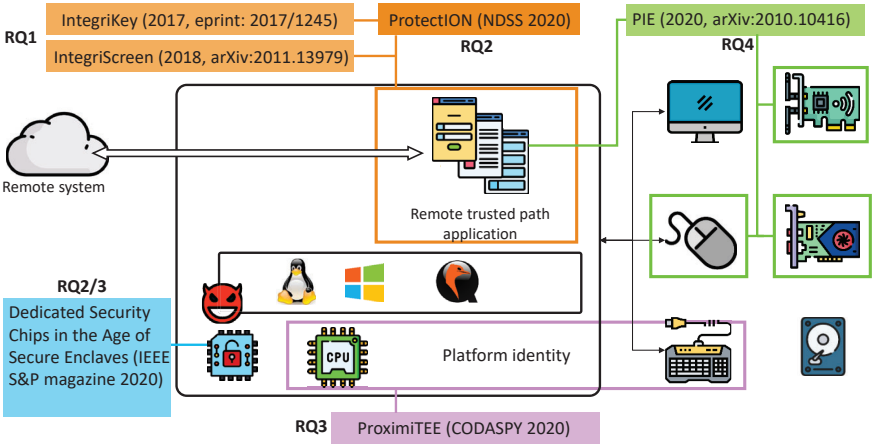


FIGURE 1.2: **Summary of the works in this thesis.** In this figure, we summarize the works that are in the thesis in a context of a modern computing platform.

this by introducing two new events into the enclave lifecycle, *connect* and *disconnect*, which allow to track the liveliness of one enclave from another.

A summary of all above-mentioned research questions are depicted in Figure 1.2. Additionally, the figure provides an overview of all the works in this thesis that address these research question.

1.4 SUMMARY OF THE CONTRIBUTIONS

The contributions of this thesis are divided into four parts, and they are summarized as the following:

Part ii: How (not) to build a trusted path. This part provides negative results on how system-oriented way of proposing trusted path solution can lead to insecure systems.

- Chapter 3: IntegriKey, an input signing approach to provide the second factor for integrity in user input through input signing
  a) *New attack.* In IntegriKey, we identify swapping attacks as a novel form of user input manipulation against simple user input matching strategies.

b) IntegriKey. We design and implement a user input integrity protection system that is tailored for keyboard input, prevents swapping attacks, and is easy to deploy.

c) IntegriKey tool. We develop a user interface analysis and web page annotation tool that helps developers protect their web services and minimize user effort. However, later analysis shows that the input signing is insecure. This is not due to the implementation of the input signing method, rather the fundamental pitfall of the method.

d) *Evaluation.* We verified that our tool could process UIs of existing safety-critical systems and cryptocurrency wallets correctly. Our experiments show that the performance delay of IntegriKey user input integrity protection is low. Our preliminary user study indicates that user input labeling prevents swapping attacks in most cases.

- Chapter 4: IntegriScreen, second factor for user intention integrity through UI analysis captured through a smartphone[6].

a) **Novel approach for transaction confirmation.** We design IntegriScreen based on *screen supervision*, which differs fundamentally from the existing alternatives, integrates well with user devices (e.g., smartphones), and has the potential to offer high integrity guarantees for the user inputs in the presence of compromised hosts.

b) **Prototype implementation and evaluation.** We implement a prototype dubbed IntegriScreen as an Android app and server-side component. Moreover, we perform a variety of experiments, which show the system is practical, effective, and performs well.

c) **Future challenges.** We are the first to explore the use of *screen supervision* for security, and especially in the context of transaction confirmation solutions. This new paradigm opens new possibilities for continuous supervision of user inputs over the limitations of existing solutions while neither risking user habituation nor increasing their efforts.

6 IntegriScreen was a collaboration between multiple researchers. This thesis only includes a part of the paper, where my contributions lies: the development of IntegriScreen's main idea, security analysis, and implementation of the IntegriScreens server-side component.

Part iii: Fundamentals of trusted path. This part provides the fundamental security properties for trusted path, describes attacks on existing systems, and proposes a new system that incorporate these fundamental security properties.

- Chapter 5: ProtectION: Root-of-Trust for IO in Compromised Platforms
  a) **Identification of new attacks:** We identify two new attacks on the existing trusted path systems. The first one is *input addition/reduction* attack on trusted path systems that uses input signing. Existing input signing-based approaches including IntegriKey is susceptible to this attack. Another attack is *early form submission* attack that targets trusted path systems that do not consider all modalities of input. Most of the existing trusted path approaches including IntegriScreen is susceptible to this attack.
  b) **Identification of IO security requirements:** We identify new requirements for trusted path based on the drawbacks of the existing literature: i) unless both output and input integrity are secured simultaneously, it is impossible to achieve any of the two, and ii) without protecting the integrity of all the modalities of inputs, none could be achieved.
  c) **System for IO integrity:** We describe the design of ProtectION, a system that provides a remote trusted path from the server to the user in an attacker-controlled environment. The design of ProtectION leverages a small, low-TCB auxiliary device that acts as a *root-of-trust* for the IO. ProtectION ensures the integrity of the UI, specifically the integrity of mouse pointer and keyboard input. ProtectION is further designed to avoid user habituation.
  d) **System for IO confidentiality:** We also describe an extension of ProtectION that provides IO confidentiality, where the user needs to execute an operation like a secure attention sequence (SAS) to identify the trusted overlay on display.
  e) **Implementation and evaluation:** We also implement a prototype of ProtectION that is based on off-the-shelf components and evaluate its performance.

Part iv: Relay attacks on TEE: effects on trusted path. This part provides an in-depth understanding of the implications relay attack on Intel SGX remote attestation and system to address the attack.

- Chapter 6: ProximiTEE: Understanding Relay attacks on Intel SGX remote attestation and building a system to address the relay attacks in two different attacker models – non-emulating (no leaked attestation key) vs. emulating attacker (at least one leaked attestation key).
  a) **Analysis of relay attacks.** While relay attacks have been known for more than a decade; their implications have not been fully analyzed. We provide the first such analysis and show how relaying amplifies the attacker's capabilities for attacking SGX enclaves.
  b) **ProximiTEE, a system for addressing relay attacks.** We propose a hardened SGX attestation mechanism based on an embedded device and proximity verification to prevent relay attacks. ProximiTEE does not rely on the common trust on first use (ToFU) assumption, and hence, our solution improves the security of previous attestation approaches. Note that the distance bounding approaches are well-known in the literature, but using such a method in the context of SGX is non-trivial.
  c) **Experimental evaluation.** We implement a complete prototype of ProximiTEE and evaluate it against a very strong and fast attacker. Our evaluation is the first to show that proximity verification can be both secure and reliable for TEEs like SGX.
  d) **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.

Part v: Trusted Path in Disaggregated TEEs. This part discusses TEEs distributed over CPU cores and peripherals and generalizes platform security

- Chapter 7: PIE:A Platform-wide TEE
  a) **Security properties for platform-wide integrity** We extend traditional TEEs by introducing a dynamic hardware TCB. We call these new systems platform isolation environment (PIE).

We identify key security properties for PIE, namely *platform-wide attestation* and *platform awareness*.

b) **Programming model** We propose a programming model that provides flexibility to the developers is comparable to the modern operating systems for developing enclaves that communicate with peripherals. The programming model abstracts the underlying hardware layer.

c) **Prototype and experimental evaluation** We demonstrate a prototype of a PIE based on Keystone [24] on an open-source RISC-V processor [25]. The prototype includes a simplified model of the entire platform, including external peripherals emulated by multiple Arduino microcontrollers. In total, our modifications to the software TCB of Keystone only amount to around 350 LoC.

1.5 THESIS ORGANIZATION

The thesis is organized in five parts and six chapters (including this chapter) as the following:

Part i contains Introduction (Chapter 1), and Background (Chapter 2). Chapter 1 introduces the problem statement, research questions that this thesis addresses and a brief overview of the contribution of this thesis. Chapter 2 provides a brief overview of trusted path, Intel SGX and RISC-V keystone that serves as the background of this thesis.

Part ii consists of Chapter 3 and 4 that provide details of the two trusted path systems based on two different second-factor methods: IntegriKey and IntegriScreen. IntegriKey is based on input signing, and IntegriScreen is based on the visual verification of user intent. In Part iii, Chapter 5, we discuss ProtectIOn that explores fundamental security properties of the remote trusted path and shows attacks on existing trusted path solutions.

Next, Part iv contains Chapter 6, where we discuss the challenges with relay attacks when we try to port an existing trusted path application to a TEE like Intel SGX, and we address this by introducing a new system based on distance-bounding mechanism: ProximiTEE. ProximiTEE shows how one can compromise a trusted path application without even compromising any security properties of the trusted path itself, but only by compromising underlying TEE properties - remote attestation.

Chapter 7 in Part v describes PIE, where we address the new challenges that arise from the disaggregated computing model where the trusted path

application is not only bounded to the CPU cores, rather distributed among different hardware components of the platform such as a GPU, TPU, NIC, etc.

Finally, Chapter 8 in Part vi summarizes this thesis and provides concluding remarks.

1.6 PUBLICATIONS

Parts of this thesis are based on the following articles I have co-authored. As is usual for scientific works, all of the publications listed below resulted from the close collaboration with my co-authors, who, alongside me, contributed significantly.

- **Aritra Dhar**, Enis Ulqinaku, Kari Kostiainen, and Srdjan Capkun, ProtectIOn: Root-of-Trust for IO in Compromised Platforms, *In Proceedings of the 27th Annual Network and Distributed System Security Symposium, NDSS 2020*.
- **Aritra Dhar**, Ivan Puddu, Kari Kostiainen, and Srdjan Capkun, ProximiTEE: Hardened SGX Attestation by Proximity Verification, *In Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy (CODASPY '20)*. (Best paper award)
- Moritz Schneider*, **Aritra Dhar***, Ivan Puddu, Kari Kostiainen, and Srdjan Capkun, PIE: A Dynamic TCB for Remote Systems with a Platform Isolation Environment, *arXiv preprint /2010.10416 (2020)* (*shared first authorship). *(in conference submission)*
- Kari Kostiainen, **Aritra Dhar**, and Srdjan Capkun, Dedicated Security Chips in the Age of Secure Enclaves, in *IEEE Security & Privacy, vol. 18, no. 5, pp. 38-46, Sept.-Oct. 2020*.
- Ivo Sluganovic, Enis Ulqinaku, **Aritra Dhar**, Daniele Lain, Srdjan Capkun, and Ivan Martinovic, IntegriScreen: Visually Supervising Remote User Interactions on Compromised Clients, *arXiv preprint 2011.13979 (2020). (in conference submission)*
- **Aritra Dhar**, Der-Yeuan Yu, Kari Kostiainen, and Srdjan Capkun, IntegriKey: End-to-End Integrity Protection of User Input, *IACR Cryptology ePrint Archive 2017: 1245 (2017)*.

In addition, during my Ph.D., I co-authored the following publications that are non included in this thesis.

- Vasilios Mavroudis, Karl Wust, **Aritra Dhar**, and Kari Kostiainen, Srdjan Capkun:, Snappy: Fast On-chain Payments with Practical Collaterals, *In Proceedings of the 27th Annual Network and Distributed System Security Symposium, NDSS 2020*.

- David M. Sommer, **Aritra Dhar**, Luka Malisa, Esfandiar Mohammadi, Daniel Ronzani, and Srdjan Capkun, Deniable Upload and Download via Passive Participation, *In Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019*.

- Garvita Allabadi, **Aritra Dhar**, Ambreen Bashir, Rahul Purandare, METIS: Resource and Context-Aware Monitoring of Finite State Properties, *In Proceedings of International Conference on Runtime Verification, RV 2018*. (Best paper award)

- Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, **Aritra Dhar**, David M. Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun, ROTE: Rollback Protection for Trusted Execution, *In Proceedings of the USENIX Security Symposium 2017*.

2

BACKGROUND

Be clearly aware of the stars and infinity on high. Then life seems almost enchanted after all.

— Vincent Van Gogh

In this chapter, we provide the necessary backgrounds for trusted path and two trusted execution environments (TEEs): Intel SGX and RISC-V keystone that are used frequently in this thesis. Additionally, we also provide some background on disaggregated computing architecture in modern data centers.

2.1 TRUSTED PATH

The term *trusted path* was first used in 1985 in Trusted Computer System Evaluation Criteria[1] [26] which is a computer security standards set by the United States Department of Defense (DoD). There, the trusted path denotes a communication channel between a user and what the user intended to communicate with. The trusted path must ensure that an attacker cannot intercept or modify the information. However, over the last decades, the trusted path concept has been refined heavily. For example, IO trusted path is defined as the problem when a user executes IO operations with a trusted application. The problem of the IO trusted path is an open problem since the wide-spread deployment of GUIs. For example, some of the earliest research papers investigate how much users need to trust the desktop environments [27] as it handles sensitive login information from the user. The problem of trusted user input to a remote server through an untrusted host has been studied in a few different contexts. Here we provide a brief background of some of the existing works. A more detailed analysis of such works could be found in Section 5.2.

1 Frequently refereed to as *Orange Book*.

2.1.1 *Transaction confirmation*

One common approach is transaction confirmation using a separate trusted device. For example, in the ZTIC system [28], a USB device with a small display and limited user input capabilities are used to confirm transactions such as payments. The USB device shows a summary of the transaction performed on the untrusted host, and the user is expected to review the summary from the USB device display before confirming it. This approach is prone to *user habituation*, i.e., the risk that users confirm transactions without carefully examining them to be able to proceed with their main task, such as completing the payment, similar to systems that rely on security indicators [29, 30, 31]. Another limitation of this approach is that it breaks the normal workflow, as the user has to focus his attention on the USB device screen in addition to the user interface of the host. Finally, such trusted devices with displays and input interfaces can be expensive to deploy.

2.1.2 *External Device-based Solution*

There are several existing solutions that use trusted external trusted devices as an intermediary between the IO devices and the untrusted host. Fidelius [14] uses overlay in conjunction with Intel SGX enclaves to project sensitive input data from the user on the screen. Bump in the Ether [12] uses a mobile client as a trusted display that connects to the keyboard and relays encrypted keystrokes to the host. FPGA-based overlays [32] proposal uses an FPGA device between the display and the host to overlay secure information on the screen.

2.1.3 *Trusted hypervisor*

Another common approach is secure user input using a trusted hypervisor. Gyrus [33] and Not-a-Bot (NAB) [34], Aurora [35] are systems where a trusted hypervisor, or a trusted VM, captures user input events and compares them to the application payload that is sent to the server. SGXIO [36] assumes a trusted hypervisor through which the user can provide input securely to a protected application implemented as an Intel SGX enclave [37] which in turn can securely communicate with the server. Bastion-SGX [38] proposes a trusted hypervisor-based solution to enable Bluetooth-based IO communication.

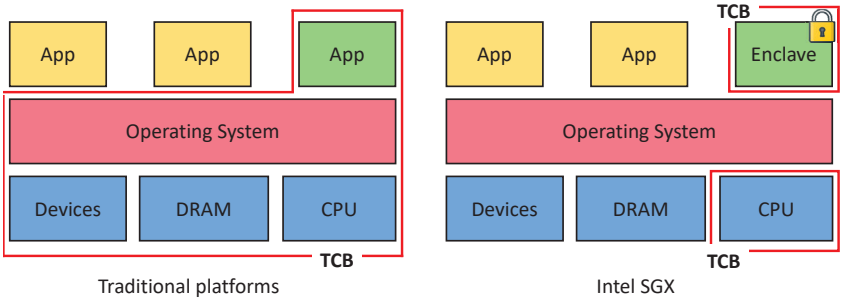


FIGURE 2.1: **Intel SGX trust model compared to traditional platform.** The figure shows the comparison of software had hardware TCB between a traditional platform and an Intel SGX platform. In a traditional platform, the user of an application (e.g., the green app box) needs to trust the hardware (CPU, DRM, other devices, etc.) and the entire software stack (OS, VMM, hypervisor). Compared to the traditional platform, in an Intel SGX platform, the user needs to trust the specific isolated execution environment (enclave) and the Intel CPU.

2.2 INTEL SGX BACKGROUND

Intel SGX is a TEE architecture that isolates application enclaves from all other software running on the system, including the privileged OS, hypervisor, BIOS etc. [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, and its initialization actions are recorded securely inside the CPU, creating a *measurement* that captures the enclave's code. Enclaves can perform local attestation, which allows one enclave to ask the CPU to generate a signed report that includes its measurement. Another enclave on the same platform can verify the validity of the report without interacting with any other external services. Enclaves can *seal* data to disk, which allows them to securely store confidential data such that only the same enclave running in the same CPU will be able to retrieve it later.

2.2.1 *Attestation*

Attestation is an important security primitive for a TEE that ensures that the SGX platform runs a correct piece of software (enclave). Upon instantiation by the trusted hardware, the platform can compute the *measurement* of the

enclave that includes the cryptographic hash of the code, data, stack, heap, etc. The platform's private key then signs the measurement of the enclave. This signed measurement is known as the report. A verifier can then check the signature of the report to verify the following:

1. The enclave is running on a legitimate platform.
2. The enclave code is proper as it matches a known measurement of the software.
3. Corresponding public key of the platform to create a secure channel with the enclave in order to provision secret to that enclave.

Intel SGX also implements an attestation scheme that follows the same principles. Every Intel processor comes with two device root keys that fused into the processor during the production phase [39] - root provisioning key and root sealing key. These keys are the related key generations are shown in Figure 2.2. The attestation key is derived from these initial platform keys that are used to generate a report from the enclave measurement.

MEASUREMENT. When an enclave is built and initialized, Intel SGX will generate a cryptographic log of all the build activities (for more details, refer to Intel's white paper on attestation in sealing [40]), including:

- The content of the enclave that includes code, data, stack, and heap
- Location of each page within the enclave
- Security flags being used

The "Enclave Identity", which is a 256-bit hash digest of the log, is stored as `MRENCLAVE` as the enclave's software TCB. In order to verify the software TCB, one should first securely obtain the enclave's software TCB, then securely obtain the expected enclave's software TCB and compare those two values.

REPORT. `REPORT` contains the following information:

- Measurement of the code and data in the enclave.
- A hash of the public key in the ISV certificate presented at enclave initialization time.

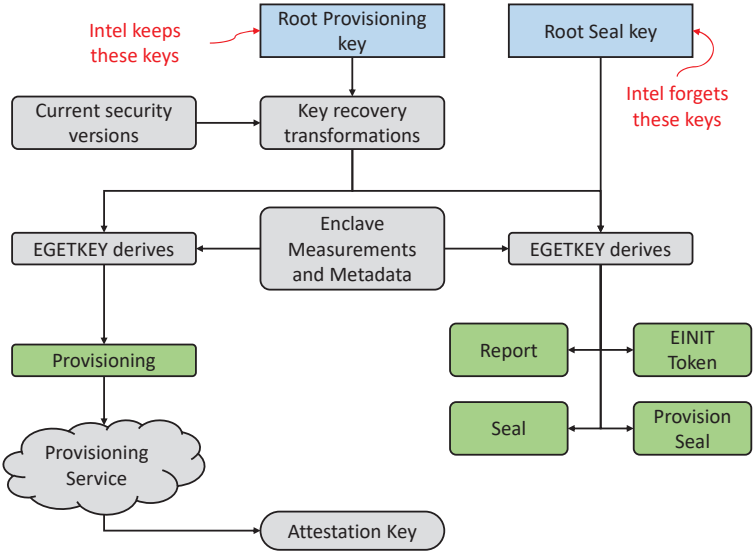


FIGURE 2.2: **Intel SGX keys and key derivations.** The figure shows the two platform keys: root provisioning key and root sealing key and their corresponding derived key for attestation and sealing. The figure is a reproduction of the official Intel documentation found in [39].

- User data.
- Other security-related state information.
- A signature block over the above data, which can be verified by the same platform that produced the report.

Intel SGX provides two forms of attestation: local attestation and remote attestation.

2.2.2 *Local Attestation*

Local attestation (also known as the Intra-Platform Attestation) enables two enclaves to verify that they are running the proper version of the enclave code on the same hardware platform. Figure 2.3 shows one such example of local attestation involving Application A that runs Enclave A and Application B that runs Enclave B. The step of the attestation process

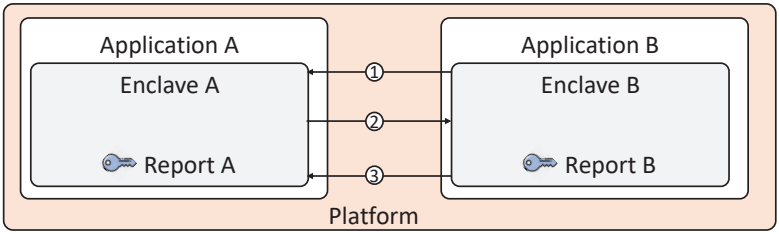


FIGURE 2.3: **Intel SGXlocal attestation example.** Intel SGX local attestation example that involves two pairs of applications and enclaves (A and B). This figure is a reproduction of the figure in Intel's white paper [40].

is as the following, and it follows the description provided in Intel's white paper [40].

① Enclave A obtains Enclave B's `MRENCLAVE` value. We assume that Application A and Application B already have a communication channel over either shared memory or file system. Note that the actual communication channel does not have any effect on the local attestation process. Moreover, the communication channel between Application A and Application B does not need to be secure.

② Enclave A invokes the `EREPORT` instruction together with enclave B's `MRENCLAVE` to create a signed `REPORT` destined for enclave B. Enclave A transmits its `REPORT` to enclave B via the insecure communication channel mentioned in step ①.

③ After receiving the `REPORT` from Enclave A, Enclave B executes the following steps:

a) Enclave B calls `EGETKEY` to retrieve its Report Key, recomputes the `MAC` over the `REPORT` structure, and compares the result with the `MAC` accompanying the `REPORT`. A match in the `MAC` value confirms that A is indeed an enclave running on the same platform as Enclave B, and as such, Enclave A is running in an environment that abides by Intel SGX's security model.

b) Once the firmware and hardware components of the TCB have been verified, and Enclave B can then examine Enclave A's `REPORT` to verify the software components of the TCB. `MRENCLAVE` reflects the contents of the software image running inside the enclave. `MRSIGNER` reflects the sealer's identity.

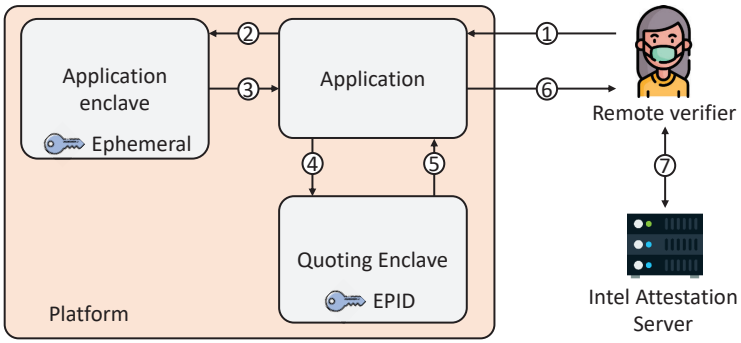


FIGURE 2.4: **Intel SGX EPID remote attestation example.** Intel SGX remote attestation example that involves the application enclave, quoting enclave, challenger and Intel attestation server. This figure is a reproduction of the figure in Intel's white paper [40].

c) Enclave B can then reciprocate by creating a `REPORT` for enclave A by using the `MRENCLAVE` value from the `REPORT` it just received.

d) Enclave B transmits its `REPORT` to Enclave A. Enclave A can then similarly verify the report to enclave B, confirming that Enclave B exists on the same platform as Enclave A.

2.2.3 *Remote Attestation*

Remote attestation enables an external verifier to check whether a specific enclave has been correctly instantiated in an SGX protected environment. In the following, we describe the two main classes of remote attestation supported by Intel: i) "enhanced privacy ID" (EPID) attestation [41], and ii) the recently introduced "data center attestation primitives" (DCAP) [42].

EPID ATTESTATION. The EPID remote attestation is an interactive protocol between three parties: the remote verifier; the attested SGX platform; and the Intel Attestation Service (IAS), an online service operated by Intel. Each SGX platform includes a system service called *Quoting Enclave* (QE) that has exclusive access to an EPID key. EPID is a group signature scheme that allows a platform to sign objects without uniquely identifying the platform or linking different signatures. Instead, each signer belongs to a "group", and verifiers use the group's public key to verify signatures. EPID

supports two modes of signatures. In the fully anonymous mode of EPID, a verifier cannot associate a given signature with a particular member of the group. In Pseudonymous mode, an EPID verifier can determine whether it has verified the platform previously. One example flow of the EPID remote attestation is shown in Figure 2.4. The steps are as the following:

① The remote attestation process starts with the remote verifier who wants to attest to the application enclave running on the remote platform. The remote verifier issues a challenge that is sent to the application. This is because the enclave has no communication capability and depends on the OS or the untrusted application for communication with the outside world. Note that the challenge also includes a nonce for replay protection.

② The application is provided with the Quoting Enclave's Enclave Identity. The application relays the remote verifier's challenge to the application enclave.

③ The enclave generates a manifest that includes a response to the challenge and an ephemerally generated public key to be used by the challenger for communicating secrets back to the enclave. It then generates a hash digest of the manifest and includes it as User Data for the `EREPORT` instruction that will generate a `REPORT` that binds the manifest to the enclave. The enclave then sends the `REPORT` to the application.

④ The application forwards the `REPORT` to the Quoting Enclave for signing.

⑤ The Quoting Enclave retrieves its Report Key using the `EGETKEY` instruction and verifies `REPORT` using the local attestation. It then replaces the MAC over this `REPORT` with a signature created with a device-specific (private) asymmetric key. The output of this process is called a `QUOTE`. The Quoting Enclave returns this `QUOTE` structure to the application.

⑥ The application sends the `QUOTE` structure and any associated manifest of supporting data to the service challenger.

⑦ The remote verifier sends the `QUOTE` to the Intel attestation server (IAS) to validate the signature over the `QUOTE`. IAS then verifies the integrity of the manifest using `USERDATA` and checks the manifest for

the response to the challenge it sent in step ① and send an appropriate response to the remote verifier

The attestation key used by the QE is part of a group signature scheme called EPID that supports two signature modes: random base mode and name base mode, also called "linkable" mode. Both signature modes do not uniquely identify the processor to the IAS, but only a group, like a particular processor manufacturing batch. The difference between them is that the linkable signature mode allows checking whether two attestation requests came from the same CPU.

DCAP ATTESTATION. Whereas the EPID attestation variant requires connectivity to an Intel-operated attestation service and is limited to predefined signature algorithms, the main goal of the DCAP attestation variant is to enable corporations to run their own local attestation services with freely chosen signature types. To achieve this, each SGX platform is, at the time of manufacturing, equipped with a unique *Platform Provisioning ID* (PPID) and *Provisioning Certification Key* (PCK). Intel also provides a trusted *Provisioning Certification Enclave* (PCE) that acts as a local CA and certifies custom Quoting Enclaves that can use freely chosen attestation services and signatures.

DCAP attestation requires a trusted enrollment phase, where the enrolled SGX platform sends its PPID (in encrypted format) to a local corporate key management system that obtains a PCK certificate for the enrolled platform from an Intel-operated DCAP service. After that, the custom Quoting enclave can create a new attestation key that is certified by the PCE enclave on the same platform. The certified attestation key can then be delivered to the corporate key management system that verifies it using the previously obtained PCK certificate. Once such enrollment phase is complete, the custom QE can sign attestation statements that can be verified by a local corporate attestation service without contacting Intel.

2.2.4 *Side-Channel Leakage*

Recent research has demonstrated that the SGX architecture is susceptible to side-channel leakage. Secret-dependent data and code access patterns can be observed by monitoring shared physical resources such as CPU caches [43, 44, 45] or the branch prediction unit [46]. The OS can also infer the enclave's execution control flow or data accesses by monitoring page fault events [47]. 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 system vulnerabilities Spectre [48] and Meltdown [49] allow application-level code to read memory content of privileged processes across separation boundaries by exploiting subtle side-effects of transient execution. The Foreshadow attack [50] demonstrates how to extract SGX attestation keys from processors by leveraging the Meltdown vulnerability.

2.2.5 *Microcode updates*

As described in Section 2.2.1, during manufacturing, each SGX processor is equipped with two 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 demonstrate that it is a genuine Intel CPU running a specific microcode version. It then joins a matching EPID group and obtains an attestation key [41] (or a signing key for the PCE enclave).

Microcode patches issued by Intel can be installed on processors that are affected by known vulnerabilities such as the Foreshadow attack, as mentioned earlier. When a new microcode version is installed, the processor repeats the provisioning procedure and joins a new 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 [41].

2.3 RISC-V KEYSTONE

2.3.1 *RISC-V*

RISC-V is a modern open-source instruction set architecture (ISA). In recent years, RISC-V has sparked immense interest by industry and academia, with many software and hardware proposals. As a result, multiple open-source cores that implement various subsets of the RISC-V standard have emerged [25, 51, 52, 53]. Software-wise, RISC-V is supported by many projects, e.g., GCC, Linux, and QEMU, amongst others. RISC-V has also become a popular prototype platform for security research [18, 24, 54].

2.3.2 *Keystone TEE*

Keystone [24] is a TEE framework based on RISC-V that utilizes *physical memory protection* or PMP to provide isolation. PMP is part of the RISC-V privilege standard [55], and it allows to specify access policies that can individually allow or deny reading, writing, and executing for a certain memory range. E.g., PMP can be used to restrict the operating system (OS) from accessing the memory of the bootloader. Every access request to a prohibited range gets trapped precisely in the core and results in a hardware exception. Keystone relies on a low-level firmware with the highest privilege, called security monitor (SM), to manage the PMP.

The SM maintains its own memory separate from the OS and protected by a PMP entry. It also facilitates all enclave calls, e.g., it creates, runs, and destroys enclaves. The SM configures the PMP so that the OS can no longer access the enclave's private memory. Upon a context switch, the SM reconfigures the PMP to allow or block access to the enclave. E.g., during a context switch from an enclave to the OS, the SM changes the PMP configuration such that access to the enclave memory is prohibited. Conversely, on a context switch back to the enclave, the PMP gets reconfigured to allow access to enclave memory. Because the SM is critical for the security of any enclave and the whole system, it aims to be very minimal and lean. As such, the SM is orders of magnitudes smaller than hypervisors and operating systems (15 KLoC vs. millions LoC [56, 57]). There are also efforts to create formal proofs for such a SM [58]. Keystone also provides extensions for cache side-channel protections using page coloring or dynamic enclave memory.

2.4 DISAGGREGATED COMPUTING ARCHITECTURE

Traditional computing hardware is built around the concept of *motherboard-as-a-unit* (or mainboard), where a motherboard has one or multiple CPU sockets, DRAM slots, PCI express ports to plug expansion cards for storage, accelerators, networks, etc. Many of the legacy data centers employs such integrated server racks (e.g., Dell PowerEdge rack server [60], Lenovo Blade server [61], etc.). However, such systems are not fully flexible, e.g., one needs to allocate a new server rack to expand the DRAM if existing racks are already at their maximum capacity. Such a way to resource allocation/revocation is further complicated because not all VM instances are identical in terms of the workload they handle. Some of them are CPU-

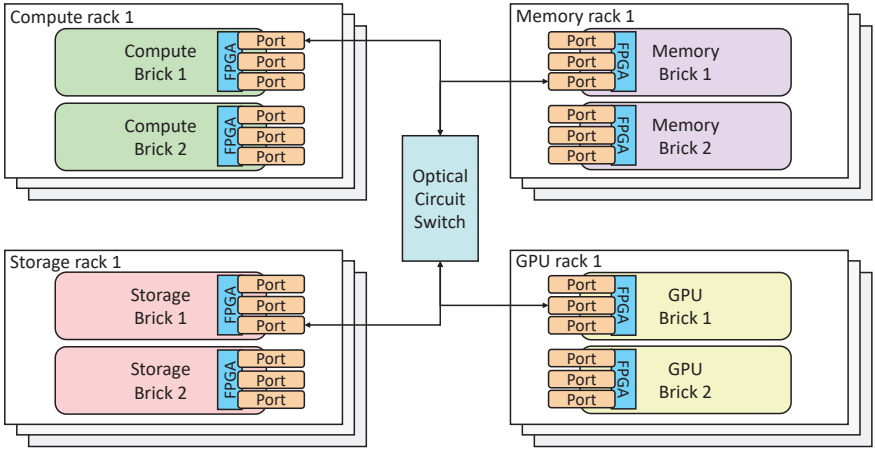


FIGURE 2.5: **Disaggregated architecture similar to dReDBox project [59].** The figure shows an example disaggregated architecture similar to dReDBox project [59] that are common in modern data centers. Rather than fully-fledged computers, the architecture employs different racks - compute, memory, storage etc. Inside this racks, there are multiple special purpose SoC-based microservers that are called bricks. These bricks have communication ports to talk to other bricks over fast optical circuit switches.

intensive, while others could be storage or memory or GPU intensive. In order to have a flexible allocation of resources all-around a data center, many of the modern data centers are moving to the disaggregated architecture where the resources (CPU, memory, storage, GPUs, etc.) are *disaggregated*. One such example architecture is dReDBox project [59, 62] that is also shown in Figure 2.5. Such an architecture is flexible and demand-driven. For example, if a VM requires more memory, Memory bricks can be allocated to the VM dynamically c [63]. After the computation, these bricks can be revoked and utilized in other workloads. Similar could be done for other resources such as GPUs, accelerators, etc. All of these specialized racks are connected over a very low-latency optical network and provides an abstraction of all the data center resources. A summary of different disaggregated computing proposals can be found in [64].

In the commercial space, disaggregated computing model is well-adopted. Some of the notable examples are Amazon Elastic Cloud (EC2) [22], Microsoft Azure high-performance computing (HPC) cloud [65], NetApp

HCI [66] and many more. DPU or data processing units [67] are the key component of such flexible, high-scalable disaggregated architecture. DPUs are SoC-based programmable processors that typically[2] contains the following:

- ARM-based multi-core processor.
- High-performance network interfaces.
- A rich set of flexible and programmable acceleration engines that are capable of executing complex ML or AI workloads.

These DPUs can be integrated into storage, network, or processing (CPU or GPU) nodes to make them self-sufficient. These nodes can talk to each other and combine themselves based on the workloads. Some of the well-deployed DPUs includes Nvidia BlueField 2 [68], Fungible DPU platforms [69], etc.

2 DPUs from different manufacturers differ widely. The following description is how Nvidia defines DPU.

Part II

HOW (*NOT*) TO BUILD A TRUSTED PATH

3

INTEGRIKEY: INTEGRITY PROTECTION OF KEYBOARD INPUT FOR SECURITY-CRITICAL WEB SERVICES

The beginning is perhaps more difficult than anything else, but keep heart, it will turn out all right.
— Vincent Van Gogh

3.1 INTRODUCTION

Many web user interfaces implement security-critical functionality. Examples include web-based configuration of safety-critical devices like Programmable Logic Controllers (PLCs) used in manufacturing plants, medical devices, and home automation systems [70, 71, 72, 73]. Payments in online banking and cryptocurrency transactions from online wallets are additional examples of security-critical user interfaces implemented as web services. Figure 3.1 shows on the left a configuration web form for a commercial PLC system [3] that we use as a running example throughout this chapter, and on the right, a browser-based Bitcoin wallet provided by BitGo [74]. The defining characteristics of such web services are that (i) they accept input from the user, (ii) the inputs are sensitive to minor changes, and (iii) after committing an input, the resulting action is difficult to reverse.

In such remote configuration, the communication between the host and the security-critical device (or its programmer device) is easy to protect through standard means such as a TLS connection [75]. However, if the host platform gets compromised—as standard PC platforms so often do—the attacker can manipulate any user-provided configuration settings. Such *user input manipulation attacks* are difficult to detect (before it is too late!) and can have serious consequences, including safety violations that can put human lives in danger.

More generally, trusted input through an untrusted host platform to a remote server remains an open problem despite various research efforts [9, 28, 33, 36, 76, 77]. Indeed, all known approaches for *trusted input* have their limitations. For example, financial transaction confirmation from the display of a separate trusted device, like a USB dongle, is prone to user habituation and requires expensive additional hardware [28]. Secure input systems based on a trusted hypervisor have a large TCB and do not tolerate

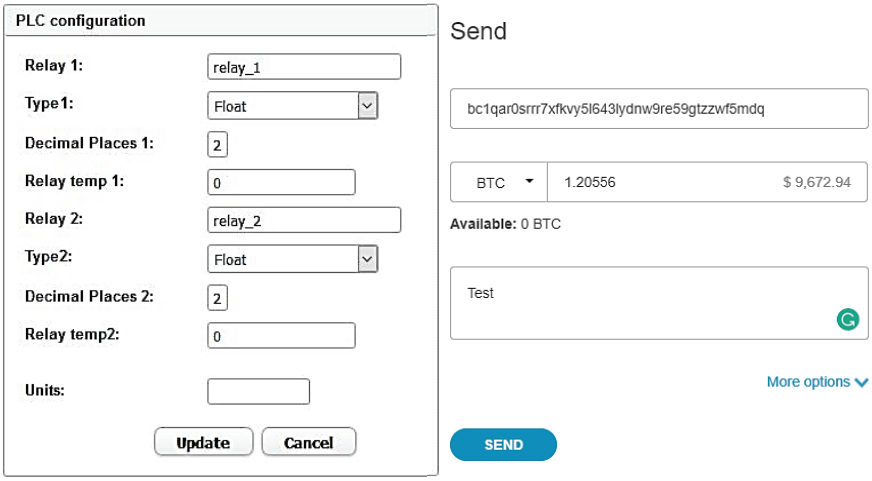


FIGURE 3.1: **Example configuration page of a web-based PLC.** Screenshot from the (a) ControlByWeb x600m [3] I/O server configuration page, (b) web-based bitcoin wallet provide by BitGo [74]. In rest of this chapter, we use the PLC server as the running example for our proposed system.

complete host compromise [36]. We review such prior solutions and their limitations further in Section 3.2.

3.1.1 *Our Solution*

In this chapter, we address the *integrity protection* of user input in security-critical web user interfaces. Our goal is to design a solution that provides strong protection (e.g., no risk of user habituation, small TCB) and easy adoption (e.g., minimal changes to the existing systems and user experience, low deployment cost). We use a remote configuration of a commercial safety-critical device (see Figure 3.1 (a)) as a running example. Still, we emphasize that our solution is not limited to that application domain. In Section 3.9 we discuss how the same approach can be applied to the integrity protection of various other services, including financial transactions, social media posts, and email.

The basic idea behind IntegriKey is straightforward, and we call this approach *input trace matching*. When the user needs to perform a security-critical web operation like configuring a PLC device or performing a cryptocurrency payment, he installs a trusted embedded device between the user input peripheral and the host platform. This device intercepts user's input events, passes them through to the host, and sends a trace of them (over a secure channel) to an authenticated remote server that compares the trace to the user input received from the host to detect input manipulation. Once the user has completed the web transaction, he can remove the embedded device from the host to preserve the user's privacy and prevent any subsequent input events sent to the same server unnecessarily.

This approach can be seen as a second factor for user input integrity protection. If the primary protection mechanism (i.e., the integrity of the host platform itself) fails, the secondary protection provided by input trace matching ensures that the target safety-critical device cannot be misconfigured.

Secure and easy adoption of this idea involves overcoming some technical challenges. The first is related to security, as an attacker that fully controls the host can execute restricted forms of user input manipulation attacks, where he exchanges input values from interchangeable UI elements (e.g., two integers with overlapping ranges). Such *swapping attacks* cannot be detected by the server relying on the input trace alone. Another challenge is related to deployment. Our trusted device needs to communicate with the server, but we want to avoid building an (expensive) separate communication channel into it. We further want to avoid installing additional software on the host that could assist in such communication.

SYSTEM AND TOOL. Based on this idea, we design and implement IntegriKey, a user input integrity protection system, that is tailored for *keyboard* input, as keyboard input is sufficient for controlling many security-critical web interfaces, including the configuration of existing commercial safety-critical devices and execution of financial transactions.

Our system realizes the trusted embedded device as a simple USB bridge (for short Bridge) that is accompanied by a server-side user input matching library. Our solution includes a simple *user labeling* scheme to prevent subtle swapping attacks, where the user is asked to annotate interchangeable input elements. For easy adoption, we leverage the recently introduced WebUSB browser APIs to enable communication between Bridge and the server in a plug-and-play manner.

We also develop IntegriTool, a user-interface analysis tool that helps developers to protect their web services and minimizes the added effort of users. In particular, the IntegriTool detects input fields in web forms that require labeling and annotates the UI accordingly.

We implemented a prototype of Bridge using an Arduino board and evaluated IntegriTool using a range of existing web-based configuration UIs supported by x600m, a commercial PLC server [3], and several web-based Bitcoin wallets such as Bitgo [74], Bitcoin wallet [78], Coinbase [79], Coinspace [80] and Blockchain wallet [81]. Our results show that the tool can correctly process the configuration UIs of many existing security-critical web user interfaces. Our Bridge implementation adds a delay of 5 ms on the processing of keyboard events, and its TCB is 2.5 KLOC.

We also conducted a preliminary user study where we simulated a swapping attack on 15 study participants. Labeling prevented the attack in 14 cases.

3.1.2 *Our Contributions*

To summarize, in this chapter, we make the following contributions:

1. *New attack.* We identify swapping attacks as a novel form of user input manipulation against simple user input matching strategies.
2. IntegriKey. We design and implement a user input integrity protection system that is tailored for keyboard input, prevents swapping attacks, and is easy to deploy.
3. IntegriTool. We develop a user interface analysis and web page annotation tool that helps developers protect their web services and minimize user effort.
4. *Evaluation.* We verified that our tool could process UIs of existing safety-critical systems and cryptocurrency wallets correctly. Our experiments show that the performance delay of IntegriKey user input integrity protection is low. Our preliminary user study indicates that user input labeling prevents swapping attacks in most cases.

3.1.3 *Organization of this Chapter*

The rest of the chapter is organized as the following. We explain our problem in Section 3.2. Section 3.3 introduces our approach, Section 3.4 describes

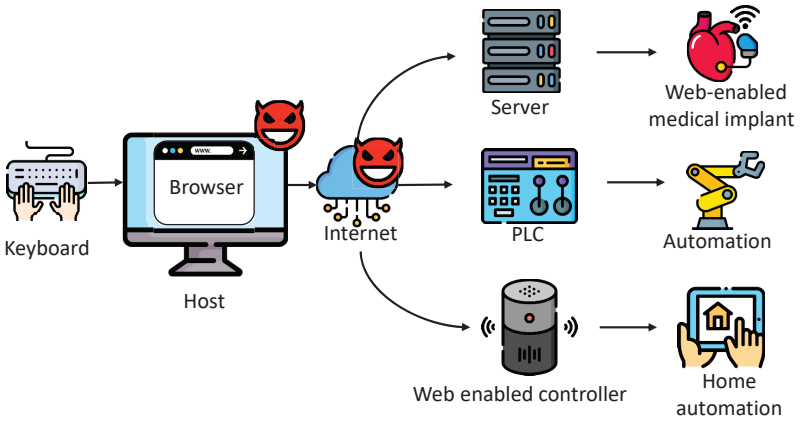


FIGURE 3.2: **System model for remote safety-critical services through an attacker-controlled host.** We consider a setting where the user configures a safety-critical device or executes financial transactions through a web service from untrusted localhost. The user input device (keyboard) and the end-system (target device, bank server, medical implants, etc.) are trusted. The host platform and the network connection can be controlled by the attacker.

our system and Section 3.5 the UI analysis tool. We provide security analysis in Section 3.6. Sections 3.7 and 3.8 explain our implementation and evaluation. In Section 3.9 we discuss other applications for our solution. Section 3.10 reviews related work and Section 3.11 concludes this chapter.

3.2 PROBLEM STATEMENT

In this chapter, we focus on the problem of user input manipulation by a compromised host PC in scenarios such as the web-based remote configuration of safety-critical devices, financial transitions, emails, and social media posts. (Attacks that compromise safety-critical systems directly are discussed in the literature; a survey of such works can be found in [82].)

3.2.1 *System Model*

Our system model is illustrated in Figure 3.2. We consider a common setting, where the user interacts with a security-critical web interface that, for

example, configures a safety-critical device like a medical device, industrial robot, or home automation system or executes financial transactions over the Internet. The user provides input through an input device (in our case keyboard) to a web browser running on the *host* machine that is a standard PC. The browser sends the configuration input to a *server* that configures (or is) the safety-critical device.

We focus on *keyboard* input, as such input is sufficient to use the configuration web interfaces of many existing safety-critical devices [3, 70, 71, 72, 73] and online wallets [74, 78, 79, 80, 81]. Later in this chapter, see Section 7.9, we discuss the challenges of protecting other types of input devices, such as mouse input, with the same approach.

ATTACKER MODEL. We consider the user input device (i.e., the keyboard) trusted and the target safety-critical device, and the server that receives the user input also trusted. We assume that the attacker may have remotely compromised the host platform completely, i.e., the attacker controls the operating system, the browser, and any other software running on the host. We consider that even the host hardware can be exploitable. We assume that the attacker does not have physical access to the host platform.

We consider such strong attacker realistic since OS vulnerabilities in PC platforms are well-known, browser compromise is increasingly common (see, e.g., [83, 84] for recent attack vectors), and hardware exploits are possible, e.g., through fabrication-time attacks [85, 86].

3.2.2 *Limitations of Known Solutions*

The problem of trusted user input to a remote server through an untrusted host has been studied in a few different contexts. Here we review the main limitations of known approaches, while Section 3.10 provides a more extensive review of related work.

TRANSACTION CONFIRMATION. One common approach is transaction confirmation using a separate trusted device. For example, in the ZTIC system [28], a USB device with a small display and limited user input capabilities are used to confirm transactions such as payments. The USB device shows a summary of the transaction performed on the untrusted host, and the user is expected to review the summary from the USB device display before confirming it. This approach is prone to *user habituation*, i.e., the risk that users confirm transactions without carefully examining them

to be able to proceed with their main task, such as completing the payment, similar to systems that rely on security indicators [29, 30, 31]. Another limitation of this approach is that it breaks the normal workflow, as the user has to focus his attention on the USB device screen in addition to the user interface of the host. Finally, such trusted devices with displays and input interfaces can be expensive to deploy.

TRUSTED HYPERVISOR. Another common approach is secure user input using a trusted hypervisor. Gyrus [33] and Not-a-Bot (NAB) [34] are systems where a trusted hypervisor, or a trusted VM, captures user input events and compares them to the application payload that is sent to the server. SGXIO [36] assumes a trusted hypervisor through which the user can provide input securely to a protected application implemented as an Intel SGX enclave [37] which in turn can securely communicate with the server. The main limitation of such solutions is that even minimal hypervisors have large TCBs, and vulnerabilities are often found in them [87, 88].

DYNAMIC ROOT OF TRUST. The third common approach is trusted user input using *dynamic root of trust* [89]. In the UTP system [10], the normal execution of the OS is suspended, and a small *protected application* is loaded for execution. The protected application includes a minimal display and keyboard drivers and is, therefore, able to receive input from the user and send it to the server together with a remote attestation that proves the integrity of the application handling the user input. The main drawback of this approach is that it changes the user experience of the web-based configuration application significantly, as small protected applications cannot implement complete web UIs. For example, the UTP system implements only a minimal VGA driver for text-based user interfaces.

3.2.3 *Design Goals*

Given these limitations of previous approaches, our solution has the following main design goals:

1. *Strong integrity protection.* Our solution should provide strong user input integrity protection even if the input host and the network are compromised. In particular, the solution should have a small TCB and not rely on tasks like transaction confirmations which are prone to user habituation.

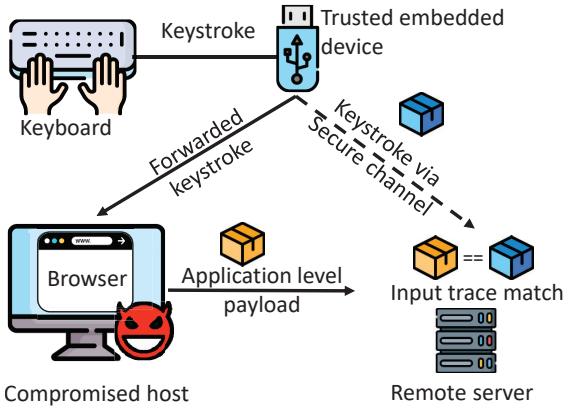


FIGURE 3.3: **IntegriKey approach overview.** The user connects a trusted embedded device between the host and the keyboard. This device relays the received keystroke events to the host and sends a trace of them over a secure channel to the remote server. The server compares the trace to the application payload received from the host to detect user input manipulation.

2. *Easy deployment.* Our solution should be easy to deploy. In particular, we want to avoid significant changes to existing safety-critical systems, input devices, host platforms, or the web-based remote configuration user experience. We also want to avoid the deployment of expensive hardware.

3.3 OUR APPROACH

In this section, we introduce our approach and explain the technical challenges involved in realizing it in a manner that is both secure and easy to deploy.

3.3.1 *Input Trace Matching*

We propose a conceptually simple approach for the protection of user input integrity that we call *input trace matching*. Our approach is tailored for keyboard input that is delivered to a remote server through a web application running on an untrusted host, as illustrated in Figure 3.3.

The main component of the solution is a trusted embedded device. When the user needs to perform a security-critical web transaction like configuring a PLC or performing a cryptocurrency transaction, the user connects the embedded device *between* the keyboard and the host. The connection from the keyboard to the embedded device and from the embedded device to the host can be wired (e.g., USB) or wireless (e.g., Bluetooth). We consider the embedded device trusted because it performs only very limited functionality, and therefore it has significantly smaller software TCB, attack surface, and hardware complexity compared to the host.

The trusted embedded device performs two types of functionalities. The first functionality is that it forwards received keystroke events from the keyboard to the host. The application running on the host (e.g., a web browser) receives the user input events and constructs an application-level payload (e.g., an HTTP response) that it sends to the server. Our approach imposes no changes to the host platform or the application software running on it. The second functionality of the embedded device is that it sends a trace of the intercepted keystrokes to an authenticated and authorized remote server over a secure channel when the user either changes the text field (by pressing the tab key) or submits the form (by pressing an enter key).

The server parses the application payload received from the host and extracts the user input values from it. Then it compares input values to the received traces to detect any possible discrepancies. If the input values and keystroke events in the traces match, the user input can be safely accepted.

Once the user has completed the web transaction, he can remove the embedded device from between the host and the keyboard. This action prevents further input events from being forwarded to the server. Therefore, our solution does not violate the user's privacy by exposing the user's input outside the security-critical task to the authorized server. We discuss user privacy in more detail in Section 3.6.3.

3.3.2 *Challenges*

Realizing the above idea involves both security and deployment challenges that we discuss next.

SWAPPING ATTACKS. Input trace matching, as outlined above, prevents *most* user input manipulations by the untrusted host. For example, if the

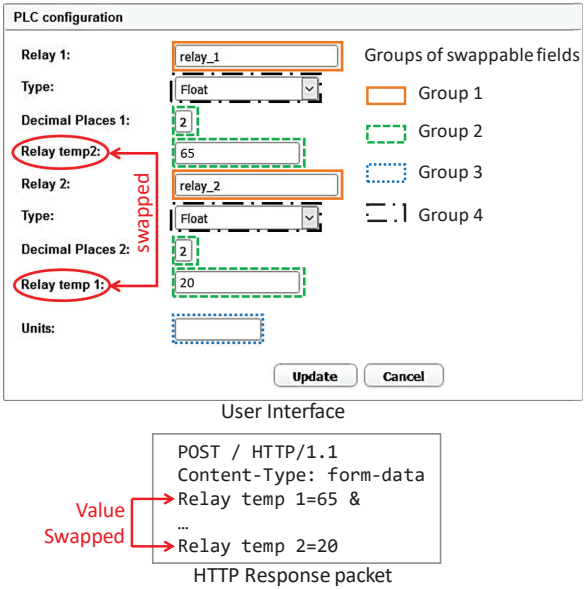


FIGURE 3.4: **Swapping attack and interchangeable inputs.** This screenshot shows our running example UI (PLC configuration web form), where the '`Relay temp 1`' and '`Relay temp 2`' user input field descriptions in the UI are swapped by the attacker. The corresponding HTTP response packet shows the swapped value of these two fields. Additionally, the figures shows groups of input fields that are swappable.

user types in one value, but the application payload contains another, the server can detect the mismatch and abort the operation.

However, the attacker may still perform more subtle and *restricted* forms of user input manipulation. The problem is exemplified by our running example UI, shown in Figure 3.4. Input trace matching allows the server to verify that all values received from the host were indeed typed in by the user, but since some values may *interchangeable* (i.e., they can have the same format and overlapping acceptable ranges), the untrusted host can perform a user input manipulation that we identify and call as *swapping attack*.

In a swapping attack, the malicious host manipulates the web form that is shown to the user and the application payload that is sent to the server. Figure 3.4 illustrates one such example, where the malicious host swaps the field names '`Relay temp1`' and '`Relay temp2`' in the UI. The user is

likely to enter the values based on the swapped field names, but the server will interpret the user input differently, based on the manipulated HTTP response constructed by the host, as shown in Figure 3.4. Because the order of the user input in the received trace matches the HTTP response, the server cannot detect such manipulation from the input order. Assuming that the entered values are in the overlapping region of acceptable values for the respective input fields, the server cannot detect such manipulation based on the received values either.

Similarly, the attacker can swap any interchangeable user input fields (overlapping format and range) in the UI. Figure 3.4 shows a grouping for all interchangeable values in our example web form. One example is in the home automation system, where the user can set the temperature of a specific room by providing the input to the web application. The attacker can swap the labels of the field with the label of the temperature of another room. More interesting examples are the fields that are semantically disjoint but share specifications. E.g., the parameters for the medical devices where the doctor can set 'blood pressure' and 'heart rate limit'. As these two fields have overlapping rangers; the attacker can swap the labels of two such fields even though the fields 'blood pressure' and 'heart rate limit' are semantically different.

ADOPTION CHALLENGES. Input trace matching requires a secure communication channel from the trusted embedded device to the server. Our goal is to keep the device simple (small TCB) and inexpensive, and thus we avoid designs where the embedded device has its own communication capabilities (e.g., dedicated cellular radio). Host-assisted communication requires the installation of new software on the host, which can complicate adoption and, in some cases, may not even be possible for the user. Ideally, connecting the trusted embedded device to the host should be all the user has to do.

Another adoption challenge is that a single device should be able to provide user input integrity protection for multiple web services. The device can be configured with the keys and addresses of all supported servers, but during deployment, we want to avoid additional user tasks, such as manually indicating which of the pre-configured servers should be used.

3.4 INTEGRIKEY: INPUT PROTECTION SYSTEM

In this section, we present IntegriKey, our system for user input integrity protection for remotely configurable safety-critical systems. Our system includes two main components: (1) the embedded trusted device realized as a simple USB bridge that we call for short Bridge and (2) a server-side user input matching library. To enable the easy deployment, we use WebUSB [90], a recently introduced browser API standard supported by the Chrome browser. This API allows JavaScript code, served from an HTTPS website, to communicate with a USB device, such as our Bridge. To prevent swapping attacks, we propose a simple *user labeling* scheme where the user is instructed to annotate swappable input values.

3.4.1 *Pre-configuration*

Our system requires a secure (authenticated, encrypted, and replay-protected) channel from the embedded device (Bridge) to the remote server. In our system, we leverage standard TLS and existing PKIs for this. To enable server authentication, the public key of the user root CA is pre-configured to Bridge. To enable client authentication, we use TLS client certificates. Each Bridge device is pre-configured with a client certificate before its deployment to the user, and the server is configured to accept such client certificates.

Besides input integrity protection, user authentication to the remote server without revealing the user's credential to the compromised host is also important. Our current implementation does not implement such user authentication, but in Section 7.9 we discuss how this can be enabled.

3.4.2 *System Operation*

Next, we describe the operation of the IntegriKey system that is illustrated in Figure 3.5.

① The user attaches the Bridge device between the host and the keyboard. The user starts the browser on the host and opens the web page for remote configuration of the target safety-critical device.

② The browser establishes a server-authenticated TLS connection (TLS_1) to the server.

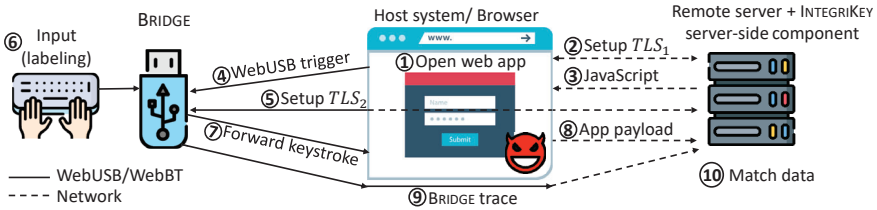


FIGURE 3.5: **IntegriKey operation**. The browser on the host opens a standard TLS connection (TLS_1) to the server which replies with a web page and JavaScript code. Using the WebUSB API, the JavaScript code invokes the BRIDGE that will establish another TLS connection (TLS_2) to the server. The BRIDGE forwards received keystroke events to the host and periodically sends a trace of them to the server that performs matching between the traces and the received application payload.

③ The server sends the web configuration form to the browser together with JavaScript code. The webform includes instructions for user labeling, as described below in Section 3.4.3.

④ The browser shows the web form to the user and runs the received JavaScript code that invokes the WebUSB API to communicate with BRIDGE. The browser passes the server URL to BRIDGE.

⑤ Based on the received URL and the pre-configured trust root and client certificate, BRIDGE establishes a mutually-authenticated TLS connection (TLS_2) to the server through the host using the WebUSB API. Mutually-authenticated TLS

⑥ The user completes the web form, as explained in Section 3.4.3.

⑦ BRIDGE captures keystrokes and forwards them to the browser.

⑧ Once the user has completed the web form, the browser constructs a payload (HTTP response) and sends this to the server over the TLS_1 connection.

⑨ BRIDGE collects intercepted keystrokes and periodically (e.g. when receiving a tab or return key press, or on every keyboard event) sends a trace of them to the server through the TLS_2 connection.

⑩ The server compares the received application payload and traces (input trace matching), as explained in Section 3.4.4. If no mismatch

is detected, the server accepts the received user input. The user can remove BRIDGE from the host.

HANDLING MULTIPLE SESSIONS/TABS. The WebUSB driver only allows one browser window to communicate with a USB device at a time. This restricts the BRIDGE from operating on multiple browser tabs or sessions at the same time. As INTEGRIKEY is targeted towards the protection of specific security-critical web-based transactions (PLC configuration, payment) as opposed to generic input protection for all web browsing, we do not consider this a problem in practice.

HANDING OF KEY STROKES. The BRIDGE acts as a USB host and handles all keyboard events from the user that includes modifiers such as `shift`, `ctrl`, navigation keys, character removal keys such as `backspace` and `del` etc. The BRIDGE registers itself as a generic USB plug-and-play device and emulates a keyboard. Hence, the BRIDGE replicates all the user keyboard actions and sends them to the browser along with the signed actions (traces) to the server. As one concrete example, assume that the user types `shift` + a, b, c and `backspace`, which appears as `Ab` in the browser. The BRIDGE records the trace as `[shift + a]+b+c+backspace` and translate to `A+b` which is received by the remote server.

3.4.3 *User Labeling*

To prevent swapping attacks explained in Section 3.3.2, we introduce a simple user labeling scheme. In this scheme, the user is instructed to annotate each interchangeable input with a textual label that adds *semantics* to the input event traces and thus allows the server to detect user input manipulation like swapping attacks.

An example of the user labeling process is illustrated in Figure 3.6. When the server constructs the web form, it adds labeling instructions to it. These instructions indicate the textual label, such as `'rel1:'` for input field named `'Relay 1'`, that the user should type in. The server adds such labeling instructions to each input field that needs protection against swapping attacks. In Section 3.5 we explain an automated UI analysis tool that helps the developer to securely find all such input fields and update the UI accordingly.

For each such field, the user types in the label, followed by the actual input value. For example, to enter value `'r1'` to the input field with name

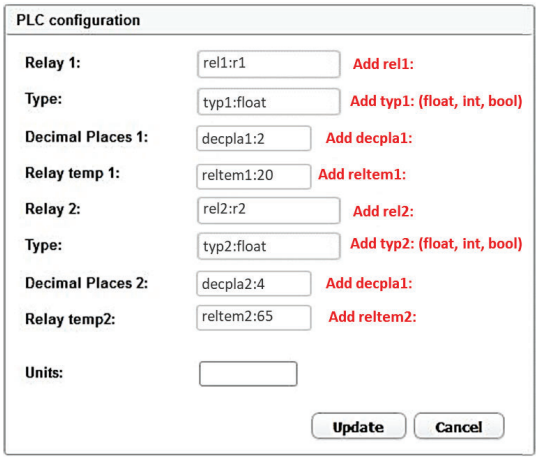


FIGURE 3.6: **User labeling example.** All input fields that need protection against swapping attacks are marked with labeling instructions. For example, to enter a value 20 to the input field `'Relay temp 1'` the user should type in `'reltem1:20'` as indicated in the web form next to the input field.

`'Relay 1'`, the user types in `'rel1:r1'`. Some input fields may not require labeling. For example, the `'Units'` field in our example user interface (Figure 3.6) does not have to be labeled by the user as it is not swappable with any other field.

We consider trained professionals that configure industrial control systems, medical devices, etc., as the primary users of our solution. Such users can receive prior or periodic training for the above-described labeling process. The secondary target group is people such as home automation system owners. In this case, no prior training can be assumed, but the UI can provide labeling instructions.

3.4.4 *Server Verification*

To verify the integrity of the received user input, the server performs a matching operation shown in Figure 3.7.

LABELED INPUTS. First, the server parses through the application payload, and for every user input field that requires labeling makes the fol-

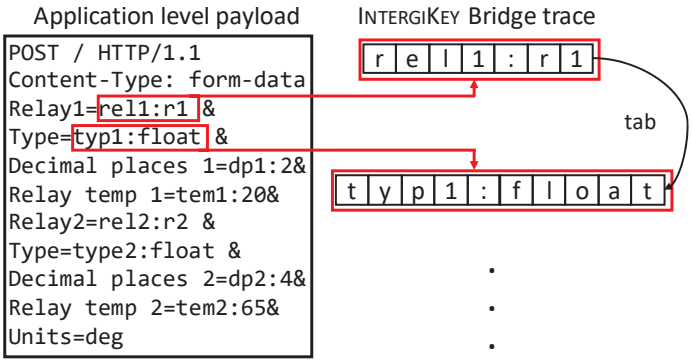


FIGURE 3.7: **Input trace matching.** The server compares user input values (and their labels) in the application payload (e.g., HTTP POST data) against the user input in the received traces.

lowing checks: the server verifies that (i) the input appears in the expected position in the application payload, (ii) the input has the expected label, and (iii) one of the received input traces contains a matching labeled value. The order in which the correctly labeled value appears in the traces is not a reason for input rejection. For example, in Figure 3.7 the input labeled as `'rel1'` appears before the input labeled as `'typ1'`, but also the opposite order in the trace would be acceptable. Such a case might happen if the user would fill in the web form values in an order that differs from the default top-to-bottom form filling.

UNLABELED INPUTS. Next, the server parses through the application payload again, and for every user input field that does not require labeling, it performs the following checks: the server verifies that (i) the input appears in the expected position in the payload and (ii) one of the input traces contains the matching value. Also, unlabeled input values can appear in any order in the in traces.

3.5 INTEGRITOOL: UI ANALYSIS TOOL

Our labeling scheme helps web service developers prevent swapping attacks. An obvious approach for developers is to require that users label all inputs. However, as labeling increases user effort, a better approach is to ask the user to label only those input fields that are interchangeable and thus

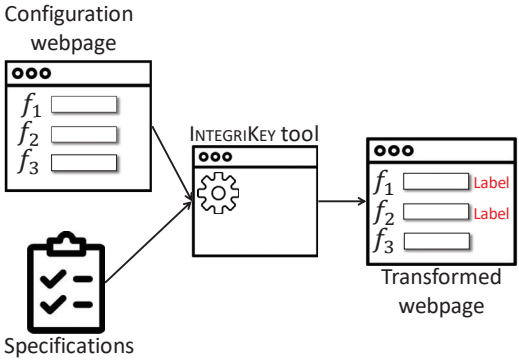


FIGURE 3.8: **IntegriTool overview**. IntegriTool takes a web page (HTML file) and a web page specification with input fields f_1, f_2, f_3. In this example f_1 and f_2 are swappable. The final output is a transformed webpage with labelling information (f_1 and f_2 requires labelling) for the users and converted mouse based UIs (drop-down menus, radio buttons, sliders etc.) to text fields.

susceptible to swapping. In this section, we describe a UI analysis tool, called IntegriTool, that helps developers to identify input fields that should be protected. When developers request labeling for those fields only, our tool also reduces user effort.

Figure 3.8 illustrates an overview of the tool that takes two inputs. The first input is the HTML code of the web form. The second input is a user interface specification that contains definitions for all input fields on the page. IntegriTool processes the provided inputs and outputs a generated webpage which is annotated with labeling instructions for the user. For example, for a user interface with input fields f_1, f_2, f_3 where f_1 and f_2 are interchangeable, the tool outputs a webpage with the labelling instruction for f_1 and f_2. An example screenshot of a webpage generated using our tool is shown in Figure 3.6.

3.5.1 *UI Specification*

The UI specification needs to be manually written by the developer. The specification captures the fact whether two user input fields in the UI are interchangeable. One such example is a home automation system, where

```
<InputSchema>
 <Input>
  <ID>Relay 1</ID>
  <Format>s[a-zA-Z0-9]+[min=1,max>min]</Format></Input>
 <Input>
  <ID>Decimal places 1</ID>
  <Format>i[0-9]*[min=0,max=5]</Format></Input>
 <Input>
  <ID>Type 1</ID>
  <Format>m[{int, float, bool}]</Format></Input>
 <Input>
  <ID>Relay temp 1</ID>
  <Format>i[0-9]*[min=-20,max=150]</Format></Input>
 <Input>
  <ID>Relay 2</ID>
  <Format>s[a-zA-Z0-9]+[min=1,max>min]</Format></Input>
 <Input>
  <ID>Decimal places 2</ID>
  <Format>i[0-9]*[min=0,max=5]</Format></Input>
  <Input>
  <ID>Type 2</ID>
  <Format>m[{int, float, bool}]</Format></Input>
 <Input>
  <ID>Relay temp 2</ID>
  <Format>i[0-9]*[min=-10,max=100]</Format></Input>
 <Input>
  <ID>Unit</ID>
  <Format>s[unit][min=1, max=5]</Format></Input>
</InputSchema>
```

SPECIFICATION 3.1: **A webpage specification example.** This web page specification corresponds to our running user interface example that is illustrated in Figure 3.4.

the user can set the temperature of a specific room by providing the input to the web application. The attacker can swap input fields for temperatures of two rooms. Another and more interesting example is a UI where two fields are semantically different but share a similar format. Consider, for example, the configuration of a medical device, where the doctor can set blood pressure and heart rate limit. As the range of these two fields is overlapping, the attacker can swap the two fields even though they are semantically very different.

The user interface specification is an XML document that defines each user input field. Specification 3.1 shows an example for our running example

UI. For each user input field, the specification provides the identifier of the webform element and its format. The format defines the type of the input (e.g., string (`s`) or integer (`i`)) and constraints for the acceptable value (e.g., a regular expression for a string or the minimum and maximum values for an integer). More precisely, we define the input format as:

$$type[regx][min = x, max = y][\{elements\}]^*$$

where *type* denotes the input field data type such as `string` (*s*), `integer` (*i*), `float` (*f*), `date` (*d*), `time` (*t*), `menu` (*m*) and `radio button` (*r*). $[regx]$ defines the regular expression for acceptable values. *min* and *max* define possible minimum and maximum `string` length or minimum and maximum values if the type is `integer`, `float`, `date` or `time`. The optional $[\{elements\}]$ is only applicable to UI objects such as `menu` (such as drop down menus) and `radio button`. $\{elements\}$ represents all the objects in the given UI element that can be chosen by the user.

We note that IntegriTool requires a tight specification to provide a precise output. If the developer provides a coarse-grained specification, that leads to an over-approximation of swappable fields by the tool that increases user effort but will not impose a security risk.

3.5.2 *Tool Processing*

IntegriTool processes all input fields from the specification by evaluating them based on their specification. For numeric input fields (`integer`, `float`, `time`, `date`) the test checks for overlapping acceptable values, i.e., a boundary condition test. For `string` fields, our tool tests if the format constraints of two input fields can be met at the same time. For example, consider the following expressions:

$$\begin{aligned} RE_1 &= s[a-zA-Z]^+[min = x, max = y] \\ RE_2 &= s[a-zA-Z0-9]^+[min = x, max = y] \\ &\implies RE_1 \subsetneq RE_2 \end{aligned}$$

where RE_1 represents a `string` containing uppercase or lowercase alphabetic characters and RE_2 represents a `string` containing uppercase or lowercase alphabetic or numerical characters. In this case, RE_1 is a subset of RE_2 as all strings from RE_1 are also members of RE_2 but there are strings in RE_2 that are not in RE_1. This can be verified by checking if $RE_1 \cap (RE_2)^c = \phi \implies RE_1 \subset RE_2$, where ϕ denotes empty set. In general,

two fields f_i (corresponding regular expression RE_i) and f_j (corresponding regular expression RE_j) can be swapped if and only if $RE_i \cap RE_j \neq \phi$ and, f_i & f_j shares at least two elements. A short proof for this is as the following:

Proof. Let F_x and F_y be two input fields and their corresponding regular expressions are RE_x and RE_y. If F_x and F_y are swappable fields, then RE_x and RE_y have at least two overlapping accepted input.
If F_x and F_y are swappable, then

$$\exists x_i \in RE_x : x_i \in RE_y \text{ and } \exists y_j \in RE_y : y_j \in RE_x$$

This was the input values x_i and y_j can be swapped. Hence, $\{x_i, y_j\} \in RE_x \cap RE_y \Rightarrow |RE_x \cap RE_y| \geq 2$ □

Based on such tests, we design Algorithm 1 that generates a group of overlapping input fields. The algorithm works by *comparing every user input field to all the other fields* in the specification.

If one of the two compared fields is `string` and another is or number (`integer`, `float`, `date` and `time`) type, we check if their regular expression is overlapping (line 6). If one of the fields is `string` and another is either `menu` or `radio button`, then we check if an element of the `menu` (or `radio button`) is a member of the `string` regular expression (line 9). If both of the compared fields are of the numeric type, then we check for the boundary condition (line 15). The boundary check is also done for the elements of `menu` and `radio button` as the members could be number type (line 12). If both fields are `menu` or `radio button` type, then we check if the intersection of two fields is empty (line 17).

Evaluating if a regular expression is a subset of another requires conversion of the regular expression to a deterministic finite automaton (DFA). The algorithm requires computing pairwise swappable tests over all the fields in the specification and returns groups of swappable fields. We analyze the complexity and performance of this algorithm in Section 3.8.

UI CONVERSION. For drop-down menu and radio button inputs, our tool simply checks for overlapping `menu` and `radio button` elements. Our tool converts such elements into the corresponding textual representation to enable form completion with the keyboard. This is illustrated in Figure 3.9, where an example `radio button` with two options (`on` or `off`) is replaced with a text field where the user is asked to type in either value `on` or `off`, correspondingly. Similarly, drop-down menus and slider elements are converted to text input fields.

Algorithm 1: This algorithm finds swappable user input fields based on user interface specification.

Input: Specification S with input fields F.
Output: Set of subset of fields $G = \{g_1, \ldots, g_n\}$ where all the fields in a $g_i \in G$ are swappable.

$G \leftarrow$ Initialize empty group
for $\forall f \in F$ **do**
  for $\forall f_{in} \in F$ **do**
    $f.regEx, f_{in}.regEx \leftarrow$ read from S
    if $f.type =$ `string` **then**
      if $f.regEx \subset f_{in}.regEx$ **then**
        $addField \leftarrow true$
      if $f_{in}.type = (menu \vee radio\ button)$ **then**
        if $f_{in}.elements \in f.regEx$ **then**
          $addField \leftarrow true$
    if $f.type = (integer \vee float \vee time \vee date)$ **then**
      if $f_{in}.type = (menu \vee radio\ button)$ **then**
        $f_{in}^{min} \leftarrow min(f_{in}.elements)$
        $f_{in}^{max} \leftarrow max(f_{in}.elements)$
      if $\neg(f^{max} < f_{in}^{min} \vee f^{min} > f_{in}^{max})$ **then**
        $addField \leftarrow true$
    if $f.type = (menu \vee radio\ button) \wedge f_{in}.type = (menu \vee radio\ button)$ **then**
      if $f.elements \cap f_{in}.elements \neq \phi$ **then**
        $addField \leftarrow true$
    if $addField = true$ **then**
      $g \leftarrow$ empty set of fields
      $g.add(f, f_{in})$
      $G.add(g)$
      $addField \leftarrow false$
return G

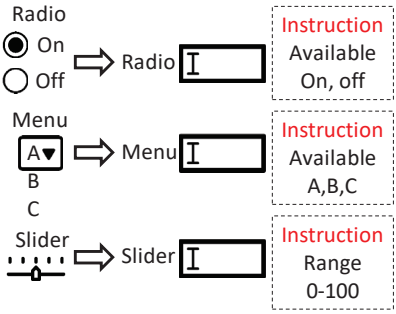


FIGURE 3.9: **UI conversion.** Conversion of radio button, drop-down menu and slider to an equivalent text field with added instructions.

3.5.3 *Web Page Annotation*

The second output of INTEGRITOOL is an annotated user interface. Our tool generates labeling instructions for users and embeds them into the web form, i.e., our tool instruments the HTML code. The instruction includes what label the user should add before each input value.

For choosing label names, we implement a simple approach, where INTEGRITOOL takes the first three characters from each of the words. For example `'Relay temp 1'` converts to `'reltem1'`. Other label generation approaches are, of course, possible as well. In case of collision of generated labels, INTEGRITOOL appends an incremented counter at the end of the label. Additionally, if there are multiple configuration pages (web forms) on the remote server that are identical, INTEGRITOOL also appends an incremented counter. This ensures that no two text fields have identical labels.

An example of the tool's output is shown in Figure 3.6 which was produced using our running example UI and the specification listed in Specification 3.1 as inputs.

3.6 SECURITY ANALYSIS

In this section, we provide an informal security analysis of our system. The goal of the attacker is to violate the *integrity* of web input. Examples include a misconfiguration of a safety-critical device or a false payment against the intention of the user.

3.6.1 *Arbitrary Modifications*

The simplest adversarial strategy is to manipulate only the application payload that is sent to the server. The attacker can, e.g., change one input value provided by the user to another arbitrary value in the HTTP response. Such attacks are detected by the server because the configuration data received over TLS_1 does not match the traces received over TLS_2.

3.6.2 *Swapping Attacks*

A more sophisticated adversarial strategy is to manipulate both the application payload and the user interface at the same time. More specifically, the attacker can change the names and the order of the user input fields and modify any instructions that are part of the user interface, such as the labeling instructions. Figure 3.4 shows one such example of the attack where the field names `'Relay temp 1'` and `'Relay temp 2'` are swapped.

The goal of a swapping attack is that the server interprets the received input values with different semantics than the user intended. Assuming that the user interface contains interchangeable fields, the attacker can construct an HTTP response where all input values are listed in the correct order, and their values match the input events. Two variants of such attacks are possible.

MANIPULATED FIELD NAMES OR INSTRUCTIONS. In the first variant, the attacker manipulates *either* the field names *or* the instructions. In such a case, the user interface that is shown to the user has an *inconsistency* because the input field names and the labeling instructions do not correspond to each other. The user may react in different ways that we enumerate below:

- *Case 1: Abort.* The user may notice the inconsistency in the UI and abort the process.
- *Case 2: Correct labeling.* The user may perform the labeling correctly. That is, he may prefix each entered input value with the matching label. The target device is configured correctly, despite the user interface manipulation.
- *Case 3: Incomplete labeling.* The user may fail to complete the required labels. The server will abort the process.

- *Case 4: Incorrect labeling.* Finally, the user may perform the labeling incorrectly. That is, he may associate one of the asked input values with an incorrect (and swappable) label prefix. The server cannot detect this case, and the target device will be misconfigured.

In Section 3.8.3 we report results of a small-scale user study that provides preliminary evidence on how common these cases are, and especially how many people would fall for the attack (*Case 4*).

MANIPULATED FIELD NAMES AND INSTRUCTIONS. In the second variant, the attacker manipulates *both* the field names and the labeling instructions. In this variant, there are two possible cases. First, the labeling instructions do not correspond to the UI field order, in which case the effect is the same as above. Second, the modified labeling instructions correspond to the modified UI, i.e., the matching field names and labeling instructions are both modified the same way. In this case, the labeling instruction reordering essentially nullifies the effect of UI field name reordering, and the UI is consistent again (no risk of misconfiguration).

We conclude that any manipulation of (*i*) UI field names, (*ii*) values in the application payload, (*iii*) labeling instructions in the UI, or (*iv*) the combination of thereof cannot violate input integrity unless there is a visible indication of it (i.e., inconsistency) in the user interface. This is the *Case 4* above, which we evaluate with a small user study.

3.6.3 *Privacy Considerations*

In our solution, the trusted BRIDGE device intercepts the user's keyboard input events and sends a trace of them to the server for matching. As *any* interception-based solution has obvious privacy concerns, in this section, we explain why the typical and recommended usage of our solution does not violate user's privacy.

1. *Device removal.* The primary usage model for our solution is one where the user attaches the BRIDGE device before a security-critical web operation and removes it after it. Thus, in such typical use, user input events outside the security-critical operation are not shared with the server. We do not recommend using our solution as a generic input protection mechanism for all web browsing but rather as a hardening mechanism for only specific security-critical operations.

2. *Server white-listing.* The BRIDGE sends the user input only to one (or more) pre-authorized (white-listed) servers. Therefore, even if the user would forget to remove the BRIDGE device from the host after the security-critical operation, the user input would be shared only with known and trusted servers. Such servers can implement additional privacy-preserving mechanisms like send a signal to the device to stop input sharing once the operation is completed.

3. *Safe handling of tabs.* As the BRIDGE uses WebUSB as the communication channel through the host's browser, the implementation of the WebUSB restricts only one website (known as the *landing page*) to bind with the USB device [91]. This ensures that if the user switches the browser tab *during the security-critical operation*, the keystrokes from that application will not be forwarded to the BRIDGE unless the user reinitializes the device manually.

Finally, we emphasize that even in the worst case where the user forgets to remove the device and one of the white-listed servers turns out to be malicious. Our solution does not *reduce* the user's privacy when compared to use without our solution. Since we assume a compromised host, the OS can trivially share all user input with any server regardless of whether our solution is used or not.

3.6.4 *Other Security Considerations*

DEFAULT VALUES. INTEGRIKEY eliminates any default values on the webpage. However, the host can always show default values to the user. If the user does not type the value by herself, the server rejects the input as the data from the browser, and the trace does not match.

TRACE DROPPING. Since all communication from BRIDGE to the server is mediated by the untrusted host, the attacker may also attempt to manipulate the traces by selectively dropping packets (e.g., remove certain user input). However, such attacks are prevented by the use of a standard TLS connection.

CROSS-DEVICE ATTACKS. An additional attack strategy is to trick the user into providing input for the configuration of one safety-critical device but using this user input for the configuration of another device. In such cross-device attacks, the host presents to the user the configuration user

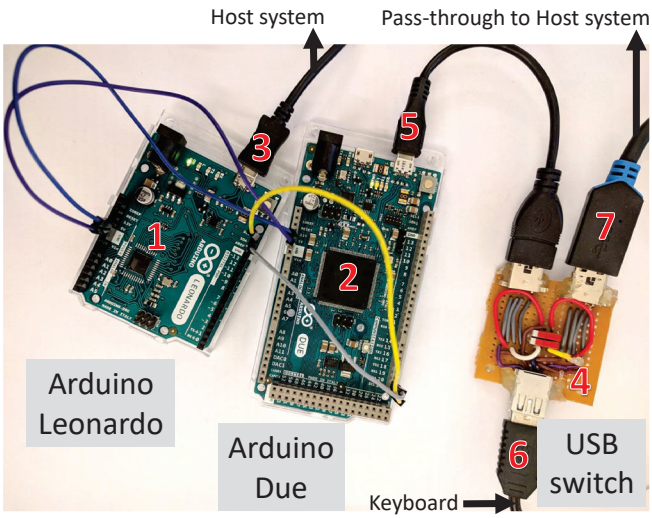


FIGURE 3.10: **Bridge prototype.** BRIDGE prototype consists of the following: 1) Arduino Due board is connected with the keyboard and executes cryptographic operations in TLS, 2) Arduino Leonardo board communicates with the browser using WebUSB, 3) USB connection from the BRIDGE to the host system, 4) USB switch to switch between the secure and insecure mode (pass-through), 5) the connection between the BRIDGE and the USB switch, 6) the keyboard connection, 7) the host pass-through connection for the insecure mode.

interface from server A but tricks BRIDGE to establish a connection with server B.

Cross-device attacks are only possible if (*i*) the same BRIDGE is preconfigured for both servers A and B, (*ii*) every user input field in the configuration web pages of servers A and B is interchangeable, and (*iii*) both configuration pages have exactly the same labels. We consider such cases rare. To protect against cross-device attacks, both configuration user interfaces A and B can be processed with the same instance of INTEGRITOOL, which can annotate the pages with unique labels.

3.7 IMPLEMENTATION

We implemented a complete IntegriKey system. Our implementation consists of three parts: (1) Bridge prototype, (2) Server input trace matching library, and (3) IntegriTool UI analysis tool.

3.7.1 Bridge *Prototype*

Our Bridge prototype consists of two Arduino boards and one USB switch, as shown in Figure 3.10. We used two separate boards, and an additional switch, because of the limited USB interfaces and computational power in the used Arduino boards, but we emphasize that a production device could be realized as a single embedded device In more detail, our Bridge prototype consists of an Arduino Leonardo board, a 16 MHz AVR microcontroller, which communicates with the host using WebUSB, and an Arduino Due, an 84 MHz ARM Cortex-M3 microcontroller, to execute computationally more expensive cryptographic operations needed for TLS. The two boards are connected using the I^2C protocol [92] in the master-slave configuration. The prototype can be connected to the keyboard via a custom-made USB switch (see 4 in Figure 3.10). We use two boards as the WebUSB library only supported AVR boards such as Arduino Leonardo, which is not powerful enough to execute cryptographic operations required by the TLS that we implement on the Arduino Due board. We also develop a limited version of the same prototype on a single Arduino Zero board using the newer driver to evaluate the viability of a single board.

As the currently available version of the WebUSB library [90] allows only one USB interface, our prototype cannot emulate a keyboard (interrupt transfer) and a persistent data (bulk transfer) device required for the TLS channel at the same time. Therefore, our prototype sends keyboard signals to the JavaScript code running in the browser. The JavaScript code interprets these signals and translates them to keyboard input on the web page. We use the Arduino cryptographic library for the TLS. The limited set of cipher suites in our TLS implementation uses 128-bit AES (CTR mode), Ed25519 & Curve25519 for signatures, Diffie-Hellman for key exchange, and SHA256 for the hash function. Our prototype implementation is approximately 2.5K lines of code.

3.7.2 SERVER *Implementation*

Our server implementation for input trace matching is a JAVA EE Servlet hosted on an Apache Tomcat web server. We tested this implementation on a standard server platform, but the same code could be installed on a PLC server, such as [3, 71, 72, 73], as well. If a legacy PLC server does not allow the installation of new code, our system could be deployed via a proxy, as discussed in Section 3.9. The server implementation consists of JavaScript that is served to the host's browser. We develop this JavaScript code that uses Google Chrome's WebUSB API to communicate with the BRIDGE. We use `XMLHttpRequest` to communicate with the remote server. SERVER users JAVA cryptographic library to implement TLS. The input trace matching is computed on the server after it decrypts the trace data from the TLS channel. This implementation is approximately 500 lines of code.

3.7.3 INTEGRITOOL *Implementation*

We implemented the UI analysis tool in JAVA based on the JAVA AWT graphics library. The tool is around 1.5K lines of code and uses the JAVA native XML interpreter library to read the specification, DK.BRICS.AUTOMATON [93] for regular expression and *Jsoup* HTML parser to parse web pages.

3.8 EVALUATION

In this section, we provide an evaluation of the INTEGRIKEY system and the INTEGRITOOL UI analysis tool. We also report results from a small-scale user study, where we simulated a swapping attack on 15 study participants.

EXPERIMENT SETUP. All experiments were performed on a laptop with a 3.2 GHz quad-core Intel i5 CPU and 16 GB memory running Ubuntu 16.10 64-bit. We used Google Chrome version 61 and JDK v1.8.

3.8.1 INTEGRIKEY *Performance*

We evaluated the performance of our BRIDGE prototype using the following three metrics:

1. *Page loading latency:* The elapsed time between the web page load and when the BRIDGE is ready to take input from the user. The JavaScript

code served by the remote server communicates with the BRIDGE and establishes a TLS using the WebUSB API. The additional TLS messages and the BRIDGE processing introduce this delay only at the initial loading of the page. We measure the difference between the time when the JavaScript code gets loaded into the browser and the time when the final TLS handshake message is sent.

2. *Keystroke latency:* The added processing delay when the user presses a key. This time is due to the internal processing of the BRIDGE. We place the measurement at the program points where the `USBHost` library starts capturing the keyboard event, and the device sends the data via the WebUSB interface to the browser.

3. *Communication overhead.* The communication overhead between the BRIDGE and the remote server. As the BRIDGE establishes a separate TLS channel with the server, this adds data overhead to the standard HTTP communication between the browser and the remote server.

We measured the page loading latency as 500 ms (includes loading of the WebUSB JavaScript and establishment of the TLS channel between the BRIDGE and the remote server) and the keystroke latency as 5 ms (both averaged over 500K iterations). These latencies are specific to the implementation architecture and the used boards, and that they can be reduced significantly using newer prototyping boards.[1]

Here, we emphasize that our solution is intended for user input integrity protection for specific security-critical operations. While a page loading latency of 500 ms may be a significant performance penalty for many web applications like online commerce in general, such a delay is insignificant for the types of operations like configuration of safety-critical devices that we focus on.

The communication overhead between the host system and the BRIDGE is very small. At the time of initialization, the BRIDGE and the remote server exchange TLS handshake messages to establish the session. The handshake messages are 60 bytes each. Each TLS message adds an extra overhead of 80 bytes (signature and announcement of next key).

We also tested the performance overhead of the server-side processing. The server has to maintain an additional TLS connection (TLS_2) which has a small cost and matches the parsed HTTP response with the received user

1 The master-slave I^2C channel is limited to 1 kHz. A Standalone implementation based on Arduino Genuino Zero, supported by the new version of the WebUSB driver, eliminates the I^2C channel and reduces the latencies significantly.

input events, which takes less than a microsecond. From a bandwidth point of view, the overhead of the second TLS channel is also small (this channel is only used to send the characters typed in by the user).

3.8.2 IntegriTool *Evaluation*

We evaluated our UI analysis tool implementation using two existing systems: PLC and home automation controller. The PLC system we used was ControlByWeb *x600m* [3] I/O server, and we tested six separate configuration web pages for it. The home automation system we used is called home-assistant [94], and we tested two different configuration pages for it. We wrote UI specifications these pages and fed the specifications to our tool implementation. The tool produced groups of interchangeable user input fields that we manually verified to be correct. Table 3.2 and 3.3 provide the details of this evaluation, including specifications for tested UIs and reported swappable elements. Based on this evaluation, we make two conclusions. The first is that our tool is able to process configuration UIs of existing, commercially available safety-critical systems. The second is that many such UIs have swappable user input fields that need the protection provided by our labeling scheme.

Additionally, we tested our UI analysis tool on user interfaces of other PLC controllers, home automation systems, medical device control, personal data management, and online banking. Again, we wrote specifications for these user interfaces and processed them through our tool that founds swappable input elements in many of the tested user interfaces. We mined around 35 different text fields from 4 different application types. We list our findings in Table 3.4. We enumerate the user input fields on each page, their specifications that we manually created (including types and constraints), and the output of the tool that is a grouping of swappable fields. We verified each output of the tool manually. We observe that in some cases, the tool outputs as "swappable" input fields that can, in fact, be easily detected at the server. For example, 'start date' and 'end date' are not swappable as the former has to be less than the latter. This is an example of a case where both fields are specified correctly, but their relationship imposes additional constraints that can be checked by the server. For such types of fields, the developers can exclude them from the input specification.

EXAMPLE INPUT FIELDS. Table 3.4 provides a listing of additional web UIs that we analyzed using the tool. The list includes web pages for online

Web page	#Fields	Processing time (ms.)	SD
x600m Web PLC			
Register configuration	6	1.654	0.0131
Counter configuration	7	0.771	0.0089
Event configuration	8	0.622	0.0085
Action configuration	5	1.241	0.0111
Supply voltage	4	0.673	0.0099
Calender configuration	11	0.713	0.0105
Home automation			
Home configuration	6	0.016	0.0018
Room configuration	5	0.012	0.0015
Bitcoin wallet			
Send Bitcoin	4	0.9	0.024

TABLE 3.1: **IntegriTool user interface processing time.** We tested the processing time of our UI analysis tool on the web pages from the x600m PLC server, the home automation system and bitcoin wallets. All the measurements where conducted over 500K iterations.

banking pages, medical programmer devices, PLC server, and home automation system. The main purpose of this table is to provide examples (or templates) for the developer for the fields which they are likely to encounter while analyzing with IntegriTool. The table provides the names of the input fields along with their specifications, such as the regular expression and the length/value constraints, and whether some of the fields are mutually swappable or not.

We notice that some user input fields are strictly swappable only with another identical field. Such as an arbitrary field is not swappable with bank account number such as *IBAN* number due to the specific format (e.g., $[ISO3166-1 \text{ IBAN code}][0-9A-Z]^{+}$ with minimum and the maximum length of 20 and 30 respectively).

PROCESSING TIME. We measured the processing time of our tool. Table 3.1 shows our results: the processing time of one web page varies from 0.01 ms to 1.65 ms. The processing time depends on the number of states in

Web pages	Fields	Type	Length/value constraint	Swappable fields
		Web PLC configuration forms		
Register configuration	Name	`string`	$[min = 1, max = 20]$	Name
	Description	`string`	$[min = 0, max = 60]$	Description
	Type	`integer`	$[min = 1, max = 5]$	Units
	Units	`string`	$[min = 1, max = 5]$	Decimal place
	Decimal places	`integer`	$[min = 0, max = 5]$	Initial
	Initial Value	`integer`	$[min = 0, max = 999999]$	Type
Counter configuration	Device	`radio button`	{`on`, `off`}	Name
	Device counter number	`integer`	$[min = 0, max = 50]$	Description
	Name	`string`	$[min = 1, max = 20]$	Device counter number
	Description	`string`	$[min = 0, max = 60]$	Decimal places
	Decimal places	`integer`	$[min = 0, max = 5]$	Debounce
	Debounce	`integer`	$[min = 0, max = 9999]$	Edge
	Edge	`integer`	$[min = 0, max = 6]$	
Event configuration	Name	`string`	$[min = 1, max = 20]$	Name
	Description	`string`	$[min = 0, max = 60]$	Description
	Type	`menu`	{`int`, `float`, `boolean`, `constant`}	
	I/O	`menu`	{available IO}	
	Event group	`menu`	{available Groups}	
	Condition	`menu`	{`On`, `Off`, `Equals`, `Change state`}	
	Eval on powerup	`radio button`	{`yes`, `no`}	
	Duration	`integer`	$[min = 0, max = 9999]$	
Action configuration	Name	`string`	$[min = 1, max = 20]$	Name
	Description	`string`	$[min = 0, max = 60]$	Description
	Event source	`menu`	{available events}	
	Type	`menu`	{`On`, `Off`, `Toggle`,...}	
	Relay	`menu`	{Available relays}	
Supply voltage	Name	`string`	$[min = 1, max = 20]$	Name
	Description	`string`	$[min = 0, max = 60]$	Description
	Decimal places	`integer`	$[min = 0, max = 5]$	
	Device	`menu`	{Available devices}	
Calender configuration	Name	`string`	$[min = 1, max = 20]$	Name
	Description	`string`	$[min = 0, max = 60]$	Description
	Event group	`integer`	$[min = 0, max = 5]$	Start date
	Start date	`date`	$[min = 01/01/2007, max = 31/12/2029]$	Stop date
	Stop date	`date`	$[min = 01/01/2007, max = 31/12/2029]$	Start time
	Start time	`time`	$[min = 00:00, max = 23:59]$	Stop time
	Stop time	`time`	$[min = 00:00, max = 23:59]$	Occurrence
	All day	`radio button`	{`on`, `off`}	Repeat val
	Repeat type	`menu`	{None, Secondly, Minutely, ...}	
	Repeat val	`integer`	$[min = 10, max = 9999]$	
	Occurrences	`integer`	$[min = 0, max = 999999]$	

TABLE 3.2: **IntegriTool evaluation 1.** We tested our implementation of the UI analysis tool on 'ControlByWeb x600m' industrial I/O server Web pages column shows the configuration pages that we tested. We list types and formats for each user input field in the tested pages, and also list those input fields that are swappable.

Web pages	Fields	Type	Length/value constraint	Swappable fields
Web Home automation configuration forms				
Home configuration	Room door lock	`radio button`	{on, off}	Room door lock
	Alarm	`radio button`		Alarm
	Water lawn	`radio button`		Water lawn
	Alarm time	`time`	$[min = 00:00, max = 23:59]$	
	Nest (thermostat)	`integer`	$[min = 16, max = 25]$	
	Sound selection	`menu`	{Available sounds}	
Room configuration	Table lamp	`radio button`	{on, off}	Table lamp
	TV back light	`radio button`		TV back light
	Celling lights	`radio button`		Celling lights
	AC	`integer`	$[min = 16, max = 25]$	
	Window shutter level	`integer`	$[min = 0, max = 10]$	

TABLE 3.3: **IntegriTool evaluation 2.** Similar to Table 3.2, we tested our implementation of the UI analysis tool on 'home-assistant' home automation systems.

the DFA constructed from the regular expression of the specification and the number of input fields.

The time complexity of our UI analysis algorithm is exponential [95] ($\mathcal{O}(2^S)$) with respect to the number of states S in the non-deterministic finite automaton (NFA) that is derived from the regular expression that is quadratic $\mathcal{O}(|F|^2)$ with respect to the number of input fields $|F|$. In practice, the analysis of tested UIs was very fast as i) the number of input fields is usually 6 or less and ii) the DFAs from the specifications contain 2-3 states for most of the input fields.

3.8.3 *Preliminary User Study*

We also conducted a small-scale user study to understand whether the users can perform the proposed labeling correctly.

RECRUITMENT. We recruited 15 study participants, aged 26-34, and all having a master's degree in computer science or related field.

PROCEDURE. We prepared a web page extracted from the ControlByWeb x600m I/O server. We passed this page through our INTEGRITOOL that annotated the page with the labeling instruction and converted drop-down

Name	Type	Regular expression	Length/value constraints	Swappable
Personal information				
Email	string	$(*)^+(@)[a-zA-Z0-9]^+(.)[a-z]^+$	$[min = 5, max = *]$	
Name	string	$[a-zA-Z.]^+$	$[min = 1, max = *]$	
Address	string	$[a-zA-Z0-9]^+$	$[min = 5, max = *]$	
Medical parameters				
Heartbeat	integer	$[0-9]^+$	$[min = 55, max = 210]$	✓
Blood pressure	integer	$[0-9]^+$	$[min = 80, max = 150]$	✓
Blood sugar (Fasting)	integer	$[0-9]^+$	$[min = 108, max = 126]$	✓
Body temperature	float	$[0-9]^+((.)[0-9])^*$	$[min = 94, max = 108]$	
Web-based PLC form [3]				
Analog Input(voltage)	float	$[0-9]^+((.)[0-9])^*$	$[min = 0, max = 12]$	
Current	float	$[0-9]^+((.)[0-9])^*$	$[min = 300(mA), max = 2(A)]$	
Thermocouple	integer	$[0-9]^+$	$[min = -15, max = 150]$	
Frequency	integer	$[0-9]^+$	$[min = 0, max = 500(Hz)]$	✓
Logic repetition	integer	$[0-9]^+$	$[min = 0, max = 9999]$	
Event duration	integer	$[0-9]^+$	$[min = 0, max = 9999999999]$	
Decimal places	integer	$[0-9]^+$	$[min = 0, max = 5]$	
Initial value	integer	$[0-9]^+$	$[min = 0, max = 999999]$	
Relay status	radio button	{On, Off}	$[min = 0, max = 1]$	
Thermocouple status	radio button	{On, Off}	$[min = 0, max = 1]$	
Thermocouple status	radio button	{On, Off}	$[min = 0, max = 1]$	✓
Energy slave status	radio button	{On, Off}	$[min = 0, max = 1]$	
Input module status	radio button	{On, Off}	$[min = 0, max = 1]$	
Thermostat status	radio button	{On, Off}	$[min = 0, max = 1]$	
Logic start/end date	date	$[0-9]^+(/)[0-9]^+(/)[0-9]^+$	$[min = 1/1/2007, max = 12/12/2029]$	
Logic start/end time	time	$[0-9]^+(:)[0-9]^+$	$[min = 00:00:00, max = 23:59:59]$	
Logic Script	string	$(*)^+$	valid controller script	
Module name	string	$[a-zA-Z0-9]^+$	$[min = 1, max = 20]$	✓
Description	string	$[a-zA-Z0-9]^+$	$[min = 0, max = 60]$	✓
Web-based home automation				
Room light toggle	radio button	{On, Off}	$[min = 0, max = 1]$	✓
Door lock toggle	radio button	{On, Off}	$[min = 0, max = 1]$	✓
Alarm	radio button	{On, Off}	$[min = 0, max = 1]$	✓
A/C	radio button	{On, Off}	$[min = 0, max = 1]$	✓
Room temperature	integer	$[0-9]^+$	$[min = 6, max = 25]$	
Window shutter level	integer	$[0-9]^+$	$[min = 0, max = 8]$	
Alarm time	time	$[0-9]^+(:)[0-9]^+$	$[min = 00:00, max = 23:59]$	
Web-based bitcoin wallet [74, 78, 79, 80, 81]				
Address	string	$[1][P2PKH]^+$	$[min = 34, max = 42]$	
	string	$[1][P2SH]^+$	$[min = 34, max = 42]$	
	string	$[bc1][Bech32]^+$	$[min = 34, max = 42]$	
BTC	string	$[0-9]^+[.][0.9]^+$	$[min = 0.0, max = 9999999999.0]$	
Reference	string	$[0-9a-zA-Z.-]^*$	$[min = 0, max = *]$	
Financial transaction, online banking				
IBAN account no.	string	(ISO3166 − 1 IBAN code)$[0-9A-Z]^+$	$[min = 20, max = 30]$	
Transaction amount.	float	(ISO4217 currency code)$[0-9]^+((.)[0-9])^*$	$[min = 0, max = *]$	

TABLE 3.4: **Input field specifications.** This table lists specifications (type, regular expression, length/value constraints) that we mined by analyzing various web pages (banking, medical, PLC, home automation). The swappable column denotes that the group of input fields with ✓ mark can be swapped with each other. Such as the current can be swapped with the frequency field.

menus to equivalent text fields. To simulate a swapping attack, we modified the page such that the description for the `'Relay temp 1'` and `'Relay temp 2'` fields were exchanged. The labeling instructions were unmodified.

We provided each study participant with an information sheet that provided brief background information on labeling and explained that the task is to configure a PLC device based on the provided instructions. We observed the study participants while they performed this task. Figure 3.11 shows the study UI and the information sheet.

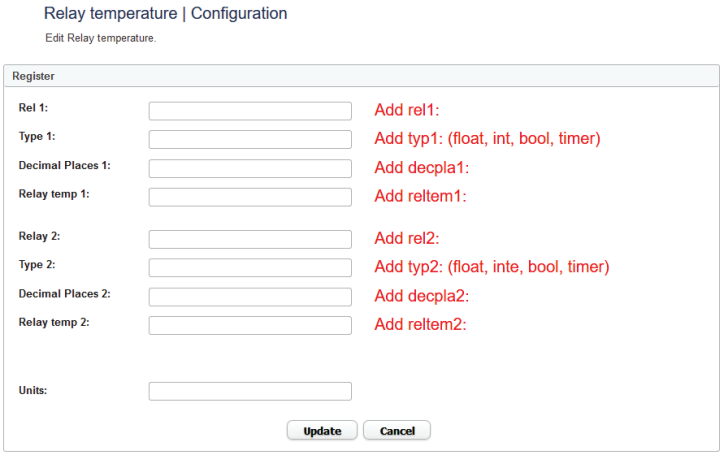
RESULTS. Out of 15 participants, 7 noticed the inconsistency between the fields and labeling instructions in the UI, stopped the task, and report it to the study supervisor (*Case 1* in Section 3.6.2). Another 7 participants did not detect the UI inconsistency but filled the input with correctly associated labels. The result is a correctly configured device (*Case 2*). One study participants completed the labeling incorrectly and fell for the attack (*Case 4*). The users reported that the additional labeling does not pose any significant effort. Additionally, none of the users made any mistakes while writing the labels in the text field along with the data.

ETHICAL CONSIDERATIONS. Our user study did not collect any private information, such as email addresses or passwords. In such cases, our institution does not issue IRB approvals.

STUDY DISCUSSION. In our user study, we provided brief instructions to the participants as shown in Figure 3.11. This is in line with the primary usage of our system, where IntegriKey is used by trained professionals, who configure medical devices, industrial PLC systems, and similar safety-critical devices. The secondary user group of our system is people like home automation system owners or cryptocurrency wallet users who have not received training for the task. Our study was not tailored for this scenario.

3.9 DISCUSSION

OTHER APPLICATION DOMAINS. In this chapter, we have focused on the web-based configuration of safety-critical devices and payments through online banking and cryptocurrency wallets. However, our approach is not limited to those application domains. Additionally, one could integrate IntegriKey with browser-based email clients and social media to certify that the legitimate user is the one who composes the mail/post. This can be



You have to fill a form that configures a remote safety-critical PLC. Misconfiguration may cause serious damage.
General instruction: Some of the input text fields require you to provide a label in front of the data. This prevent the host to manipulate the input parameters in case it is compromised. E.g., the abbreviated label for Relay temp 2 is reltem2. So, when you fill up the form, you write "reltem2:65"
You have to configure the following:

Relay 1= criticalRelay_1	Relay 2= criticalRelay_2
Type 1 = float	Type 2 = float
Decimal Places 1 = 2	Decimal Places 2 = 5
Relay Temp 1 = 20	Relay Temp 2 = 65

FIGURE 3.11: **User study instructions.** This figure shows the instruction sheet that was given to our user study participants.

achieved by the BRIDGE to sign the mail/post and communicated directly to the server. On the receiving side, there can be two distinct scenarios: i) if the receiver has a BRIDGE installed on her host, the BRIDGE can check for the signature of the mail/post and validate it, ii) if the receiving side does not have a BRIDGE, the browser can use an extension to provide a secure indicator indicating that a legitimate BRIDGE indeed signed the mail/post.

DEPLOYMENT. Assuming a browser that supports the WebUSB standard, our solution can be deployed without any changes to the host. The server-side component of our solution introduces small changes to the server. In the case of legacy systems that are difficult to modify, the required server-side functionality could be implemented by a proxy server. BRIDGE could be configured to send the user input events to the proxy that could perform

the input trace matching before passing the response to the unmodified legacy server.

BLUETOOTH. WebBluetooth [96] is another recent web API standard by Google Chrome that allows a JavaScript code to communicate with devices that are connected to the host. Our approach could be realized using WebBluetooth as well.

WEBUSB SECURITY. The recent WebUSB and WebBluetooth APIs have received some criticism regarding possible security vulnerabilities. We emphasize that in IntegriKey, usage of WebUSB is not security-critical, but it only enables communication from the Bridge to the remote server via the unchanged host. If the use of WebUSB should be avoided, IntegriKey can still be used with a browser plugin or an additional application on the host (that enables communication to the Bridge).

OTHER USER INPUT. Our current implementation is limited to keyboard input. To enable usage with various UIs with keyboard only, our tool converts elements, such as drop-down menus, sliders, and radio buttons, to text inputs. To extend our approach to pointer devices, such as the mouse, several aspects, such as mouse sensitivity and acceleration, and behavior of the mouse at the screen border would have to be considered. The fact that such mouse settings are controlled by the host OS would complicate the implementation. We investigated several commercially available PLCs, medical devices, and online wallets and learned that most of them could be configured through the keyboard alone.

USER AUTHENTICATION. An attacker that controls the host is able to eavesdrop on any user authentication credentials, such as passwords, entered into the host. To prevent such credential stealing, the trusted embedded device could be configured to act as an authentication token in addition to its main purpose of input integrity protection. For example, an administrator could configure the device with client certificates that could be used to authenticate the user during the establishment of TLS_1 connection to the server without revealing the authentication credentials to the untrusted host.

SECURE AUTOFILL MECHANISM. Besides integrity protection of user-provided input, Bridge can also be used as a hardware-assisted autofill

mechanism, eliminating the need for storing sensitive data such as user credentials, credit card number, etc. on the browser storage on the untrusted host system. The BRIDGE can use its internal flash storage to keep a key-value pairing of the identifier of the input fields and the actual data. The autofill operation is performed by the remote server to send the specific identifier of the input fields to the BRIDGE over the TLS channel between the BRIDGE and the remote server.

INTEGRIKEY autofill has two phases: *i) Initialization* where the user provides input data to the web application for the first time. The flow of operation is identical to the standard INTEGRIKEY operation described previously. The remote server sends the identifiers of the input fields to the BRIDGE. The BRIDGE stores the user data on its internal flash storage corresponding to the identifier. *ii) Autofill* phase allows the BRIDGE to populate the web forms in the browser from the data that are stored on its flash storage. The remote server achieves this by sending the identifier to the BRIDGE over the dedicated TLS channel.

AUTOMATED SPECIFICATIONS. Our current implementation of INTEGRITOOL requires that the developers specify the web page specification manually. An interesting direction for future work would be the development of a tool that parses the web page HTML and JavaScript code to generate the specification automatically.

OTHER CHANNELS. In our design, the connection from the trusted embedded device to the server shares the same physical channel as the browser, i.e., the Internet connectivity of the host. However, INTEGRITOOL can be configured in such a way that this channel remains separated physically from the host. This can be achieved, for example, by using a smartphone application in the role of the trusted device. The drawback is increased TCB size.

OTHER TRUST MODELS. We designed our system considering an attacker that can fully compromise the host. An alternative trust model, similar to [33, 34], would be one where the host OS trusted, but the browser, or one of its extensions, is compromised. Under such a trust model, the OS could take the role of the trusted embedded device.

LEGACY PLCS. Although many modern PLCs use web-based configuration interfaces (e.g., Siemens [71, 72], Schneider [73]), this is not the case for

all legacy PLSc. In principle, IntegriKey could be used on non-web PLCs as well. In this case, our solution requires a proxy that runs the webserver that connects to the Bridge.

3.10 RELATED WORK

The problem of protecting the integrity of user input that is delivered to a remote server via an untrusted host has been studied previously in a few different contexts. Here we review the most related prior works.

3.10.1 *Transaction Confirmation Devices*

The third set of known solutions use a separate trusted device to confirm user input for transactions like online payments. ZTIC [28] is a small USB device with a display and user input capabilities. This device shows a summary of the transaction performed on the untrusted host, and the user is expected to review the summary from the USB device display before confirming it. Kiljan et al. [97] propose a similar transaction confirmation device.

Such solutions have three main drawbacks. First, they are prone to user habituation, i.e., the user will not always carefully review the transaction. Second, they break the normal workflow, as the user has to focus his attention on the USB device screen in addition to the normal UI on the host. Third, such devices can be expensive to deploy. Our solution is cheap to deploy, and the user experience remains mostly unchanged.

3.10.2 *User Intention Monitoring*

The first set of related solutions focus on user *intention*. These systems attempt to ensure that the data received by the remote server is constructed as the user intended.

Gyrus [33] records user intentions in the form of text input typed by the user and later tallies it with the application payload that is sent to the server. On the host, Gyrus assumes an untrusted guest VM (dom-U) that can manipulate user input and a trusted VM (dom-0) that draws a secure overlay and captures the user input. The overlay is application-specific and covers critical input fields such as the website address bar, mail compose window, etc. When the application sends a message to the server, dom-0 matches the captured user input data with the application payload.

Not-A-Bot (NAB) [34] attempts to ensure that data received from the host was generated by the user and not by malicious software. Also, NAB relies on a trusted hypervisor that loads a simple *attester* application whose software configuration can be verified through remote attestation. The attester records user input events and provides a signed statement of them to the server. Binder [98] is another similar system where a trusted OS correlates outbound network connections with the recorded user inputs events. The main difference between these solutions and our work is that we assume a fully compromised host.

3.10.3 *TEE-based Solutions*

User input integrity has also been studied in the context of hardware-based trusted execution environments (TEEs). UTP [10] describes a unidirectional trusted path from the user to a remote server using the dynamic root of trust based on Intel's TXT technology [89]. The system suspends the execution of the OS and loads a minimal protected application for execution. This loading is measured and stored to a TPM and proved to a remote verifier using remote attestation. The protected application creates a secure channel, records user input, and sends them securely to the server. The main drawback of this approach is that such minimal protected applications cannot implement complex (web) user interfaces. For example, UTP is limited to VGA-based text UIs to keep the TCB small.

SGXIO [36] assumes a trusted hypervisor and trusted device drivers and uses them to create a secure channel from the user to an SGX enclave. Intel's Software Guard Extensions (SGX) [37] is a trusted execution environment (TEE) implemented as a specific execution mode in the processor. SGX allows isolated execution of small protected applications (enclaves) and protects their secrets and execution integrity from any untrusted software running on the same platform. The main difference to our work is the need for a trusted hypervisor.

Zhou et al. [9] 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 [77] is a small trusted kernel that manages device drivers for secure user input. We, in contrast, assume a completely compromised host.

3.11 CONCLUSION

The remote configuration of safety-critical systems is prone to attacks where a compromised host modifies user input. Such attacks can have severe consequences that can put human lives in danger. In this chapter, we have proposed a new solution, called IntegriKey, to prevent user input manipulation by the untrusted host. In our scheme, the user installs a simple embedded device between the user input peripheral and the host. This device sends a trace of user input events to the server that can detect input integrity violations by comparing it to the received application payload. Our evaluation shows that IntegriKey is cheap to build, easy to deploy.

However, despite identifying a new form of UI manipulation attack and addressing it, IntegriKey is still not a fully secure system. As the Bridge is oblivious to what is shown to the user, it can always be tricked by the attacker by manipulating some instruction or the input shown on the screen. We describe these attacks in ProtectIOn in Chapter 5.

4

INTEGRISCREEN: VISUAL SUPERVISION OF USER INTERACTIONS

A good picture is equivalent to a good deed.
— Vincent Van Gogh

4.1 INTRODUCTION

In the previous chapter (Chapter 3), we describe IntegriKey that provides a second factor for the integrity of the user input coming from a keyboard. However, this integrity guarantee is only limited to the keyboard input and does not take into account what the user sees on display. In other words, IntegriKey is oblivious to the user's *intent*. We use the term intent to establish the connection between what the user observes on display and the user response based on that. A large class of web-based services and applications (e.g., online banking or remote database access) uses modern user interfaces (UIs) displayed on the browser to interact with the user. In all of these UIs, the user's intended input is displayed on the host's screen. Users communicate their intention by entering and modifying the values shown on the screen until they are satisfied with what they see or abort if they are prevented from doing so. Similar to IntegriKey, we assume an attacker that controls the software stack of the host. Hence, the attacker controls the OS/hypervisor, device drivers, and all applications.

4.1.1 *Our Contribution*

In this chapter, we build on the observation that *users' intentions are what they see on the screen*, regardless of what a compromised host might do in the background: users enter and modify the values shown on the screen until they are satisfied with what they see, or abort if they are prevented from doing so.

Motivated by the advancements in computer vision capabilities of various camera-enabled devices (e.g., augmented reality headsets [99, 100], smart home camera assistants [101, 102] and smartphones [103, 104]), we propose the concept of *visual supervision of user's intent*: by extracting the contents of

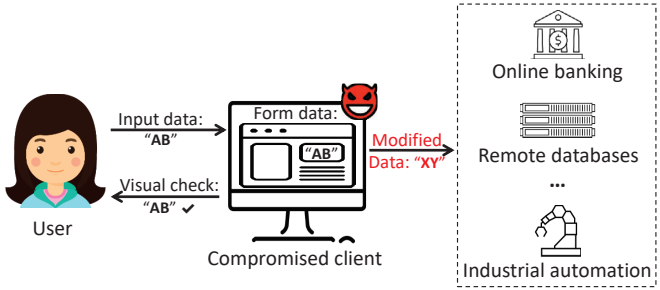


FIGURE 4.1: **Attack scenario in a compromised host.** The user communicates with remote services via a compromised local host. The attacker correctly displays user's input on the screen ("AB"), but submits "XY" instead.

the host's screen during normal user input, a camera-equipped device can confirm to the remote service the values that the legitimate user intends to submit, or notify the user of any mismatch.

In summary, this chapter makes the following contributions:

1. **Novel approach for transaction confirmation.** We design IntegriScreen based on *screen supervision*, which differs fundamentally from the existing alternatives, integrates well with user devices (e.g., smartphones), and has the potential to offer high integrity guarantees for the user inputs in the presence of compromised hosts.

2. **Prototype implementation and evaluation.** We implement a prototype dubbed IntegriScreen as an Android app and server-side component. Moreover, we perform a variety of experiments, which show the system is practical, effective, and performs well.

3. **Future challenges.** We are the first to explore the use of *screen supervision* for security, and especially in the context of transaction confirmation solutions. This new paradigm opens new possibilities for continuous supervision of user inputs over the limitations of existing solutions while neither risking user habituation nor increasing their efforts.

4.1.2 *Organization of this Chapter*

The rest of this chapter is organized as the following. In Section 4.2, we describe the problem statement, including the system and the attacker

model. Section 4.3 provides an overview of our approach: visually supervising IO on the screen and discuss the challenges of implementing such a trusted path system. Section 4.4 and 4.5 provide detailed technical description of our system IntegriScreen and its security analysis respectively. In Section 4.6, we describe the experimental prototype and evaluation result. Section 4.7 and 4.8 provide additional discussion and related works respectively. Finally, Section 4.9 conclude this chapter.

4.2 PROBLEM STATEMENT

Many sensitive applications such as e-banking or e-voting, safety-critical systems such as remote industrial PLCs, and medical devices provide web-based interfaces where users input parameters. Typically, devices such as computers and smartphones run complex software, e.g., operating systems and browsers, among other applications, but still, they are used to access sensitive remote services. Hence, these platforms expose a large attack surface from OS level to the JavaScript and browser extensions [105, 106, 107, 108, 109, 110, 111, 112, 113]. An attacker-controlled host can modify the user inputs to profit or cause intentional damage to the remote services.

Trusted path [9] provides a secure channel between the user and a trusted service (usually mediated by a trusted app on the local system). Existing trusted path solutions typically are based on two approaches: i) *trusted execution environments* (TEEs) [36] which rely on the OS and its drivers to communicate with the IO devices; therefore, they should be combined with a trusted hypervisor, which increases the TCB significantly; and ii) *transaction confirmation devices* [10], which introduce high cognitive load to the users, risk user habituation and are limited to specific use cases.

The goal of this work is to design a system that protects the interaction between the user (who uses a traditional local host) and the remote server without relying on trusted hypervisors, specialized hardware like TEEs, or solutions with high cognitive load to users and that risk habituation such as transaction confirmation devices.

4.2.1 *System and attacker Model*

Figure 4.2 shows an overview of a visual supervision system: the user owns a local host (i.e., a desktop or laptop computer) and provides sensitive inputs via a web form to a remote server over the network. We assume that

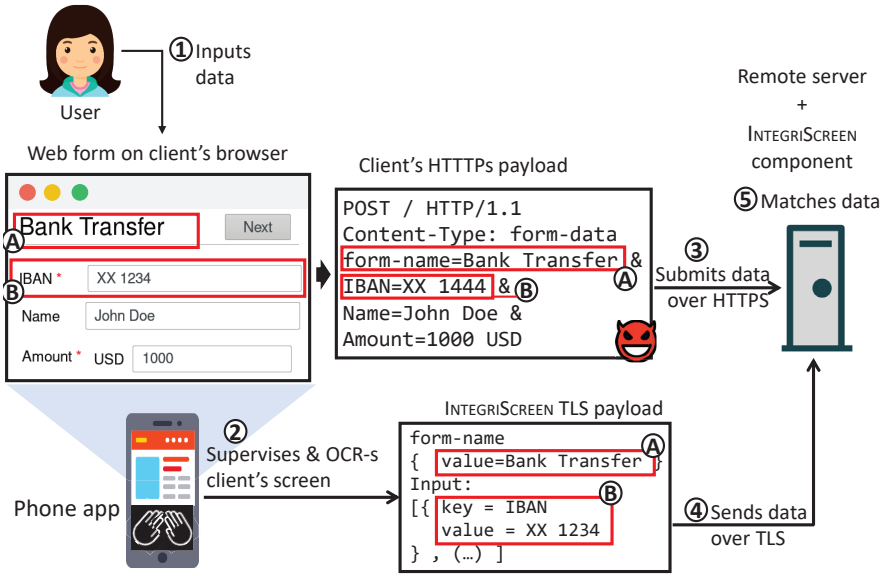


FIGURE 4.2: **System model of a visual supervision system.** The figure shows the overall approach of the visual supervision system. The phone application continuously captures the user's screen. The phone application verifies the UI, records user input, signs the input, and sends them to the remoter server. The server receives payloads both from the browser and the phone application and checks if they match.

both the host and the network are fully compromised, while the remote server is a trusted entity.

The user inputs data via a keyboard or a mouse that is connected to the host system. The user's interaction with the local host is visually supervised by a trusted application running on a device equipped with a camera, e.g., a smartphone, placed such that it can continuously capture the contents of the host screen during input. For simplicity, we refer to this device as a smartphone in the rest of this chapter – however, the proposed technique generalizes to any device with network connectivity and a front-facing camera, such as an augmented reality (AR) headsets or a smart home assistant device.

ATTACKER MODEL. We assume that the host and the network are fully compromised: the attacker can arbitrarily modify the UI elements on the

screen, execute keystrokes, or simulate mouse events in the device it controls. As the attacker controls the network, he can observe, modify, or drop any network packet between the remote server and the host. However, we assume that the smartphone is not compromised; only the legitimate user can unlock the device and run applications. The remote server is considered to be trusted. We assume that it is infeasible for an attacker who performs opportunistic attacks to compromise all the user devices at the same time. This is further reinforced by the fact that these second-factor devices have different hardware architecture and OS compared to the traditional host PC that the user owns. Moreover, such an attacker model is in line with the existing second-factor proposals.

The attacker's goal is to trick the remote server into accepting a request that does not correspond with the legitimate user's intended input. Finally, we consider the privacy of user inputs and denial of service attacks to be out of the scope of this work.

4.3 APPROACH OVERVIEW

We start by providing the general idea of visual supervision of user input, depicted in figure 4.2. A visual supervision system consists of three main components: (i) the web form and its code, running on untrusted localhost; (ii) the trusted remote SERVER with a dedicated component; and (iii) an app running on the user's secondary camera-equipped device, previously enrolled to the remote service. The general flow of the user's interaction with a visual supervision system is the following:

① The user enters the data through the form running in the untrusted host's browser.

② The visual supervision app extracts the data that is input by the user by performing optical character recognition (OCR) of the host screen in order to generate a visual *proof-of-intent*.

③ The browser transfers the user's input over a HTTPS channel to the remote server.

④ The visual supervision app transfers the generated proof-of-intent to the remote server over a dedicated TLS channel between the secondary device and the remote server.

⑤ Upon receiving data from both the browser and the secondary device, the server matches the data from two inputs. If the two inputs match exactly, the webserver accepts the input; otherwise, it rejects it.

Even after compromising the host and obtaining the victim's authentication credentials, the attacker can neither generate new requests nor covertly modify the data before submitting. This is prevented by the server, which would reject the requests due to not receiving a matching proof-of-intent.

4.3.1 *Challenges*

Following are the challenges and possible attacks to be addressed to ensure that all received requests truly correspond with the user's intended input:

1. *UI manipulation attacks* The attacker could lunch a more sophisticated attack targeting the semantics of the UI. For example, the attacker can swap labels of two text fields or change the unit of input in the label. This would result in a drastic change in input as described in existing work such as IntegriKey [114]. Hence, preserving the integrity of the UI is a crucial challenge. Note that these attacks would be successful even if the user re-types his inputs into a special device that generates a TAN (transaction authorization number) code.

2. *On-screen data modification attacks* If the data shown on the host's screen are correctly sent to the remote server, the integrity of user input is not necessarily guaranteed if data extraction happens only when the form is submitted. Despite the user entering the intended data, an attacker can subsequently modify the content of the screen in such a way that the user does not notice the change, the smartphone app records it, and the server thus receives the same maliciously modified data from both channels. For example, in the web form shown in Figure 4.2, the attacker can modify the IBAN number while the user sets the transaction amount or the execution date.

3. *Feasibility and easy integration* Given that detecting computer screens in images is an open research question [115], the recognition of form elements on screen is a challenge also. Hence, building a system that requires *minimal changes to the user interface* while at the same time maintaining the performance and achieving security guarantees against the adversarial host system poses a significant challenge. For instance, doing OCR on the whole screen degrades the performance

(< 0.5 fps) and has low classification accuracy. Camera position is an important factor to steadily track and supervise the form completion; however, another line of research in AR/VR focuses on addressing this issue [116, 117, 118].

4.3.2 *Design Goals*

DESIGN GOALS. Given the challenges mentioned above, a successful user input supervision to extract user intent should:

1. Authenticate inputs that users make through compromised hosts, i.e., ensure that the attacker cannot manipulate existing remote requests successfully.
2. Ensure that users are not manipulated into entering and submitting data that they would not submit in the absence of the attacker.
3. Require minimal added interaction in the absence of attacks: do not require users to input or explicitly verify any data except on the host device.

4.4 INTEGRISCREEN ARCHITECTURE

Now we describe how IntegriScreen addresses the challenges (refer to. Section 4.3.1) to ensure that all remote requests truly correspond to a legitimate user's intended input. IntegriScreen has two major components: IntegriScreen server-side component and IntegriScreen phone app. IntegriScreen server-side component instruments the webpage with visual cues that help the IntegriScreen application for the image processing of the web page. The server-side component also sends a signed specification of the given form to the phone app. This specification serves as the ground truth. IntegriScreen phone app processes the rendered form on the host's screen and detects any changes as per the form specification. Additionally, it also records all the user inputs. The IntegriScreen server-side component receives input data from both the browser and the phone app and checks for discrepancies. In the following, we discuss these two components in detail and show how we address the challenges. We assume a typical PKI scenario where the phone app knows the public key of the server. The phone app also generates a keypair and signs the data submitted to the server. We also assume there is an offline registration phase by which the server learns the IntegriScreen application's public key.

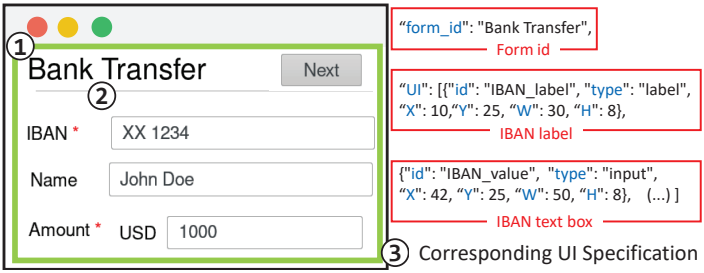


FIGURE 4.3: **Example web form and specification** generated by the INTEGRISCREEN server-side component.

4.4.1 *Server-side Component*

INTEGRISCREEN server-side component is responsible for: i) instrumentation of the web form, and ii) input trace matching. We use the webform depicted in Figure 4.3 as a running example.

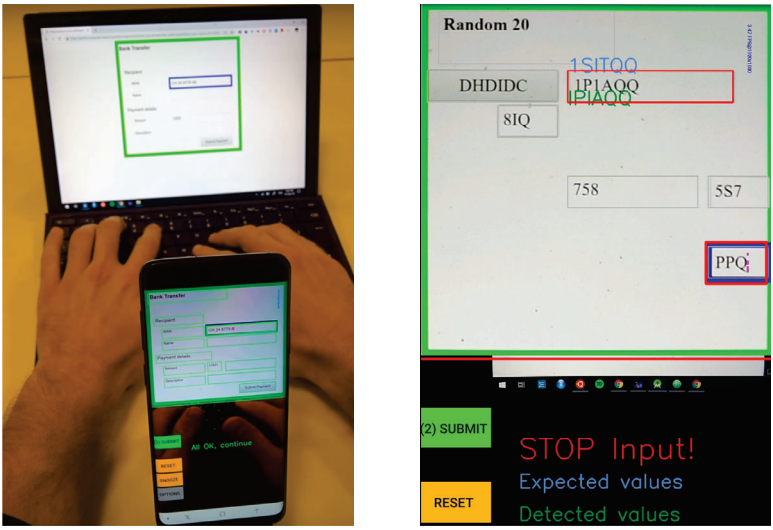
4.4.1.1 *Instrumentation of the web form*

INTEGRISCREEN server-side component carries out the following instrumentation in the web form as depicted in Figure 4.3:

① *Form Border* INTEGRISCREEN server-side component puts a visible boundary around the UI that aids the INTEGRISCREEN phone app to detect the form and align it properly. For simplicity, the current prototype uses a solid green color to indicate the form border. However, this can be fully customized as long as the four corners that indicate the protected area can be detected and tracked by the mobile app [119].

② *Form Title* The title serves as a unique identifier corresponding to a UI form and its corresponding specification.

③ *Form Specification* The INTEGRISCREEN phone app receives a form specification from the server-side component signed by the server. This specification dictates the type and location of UI elements relative to the form boundary (refer to the example specification in Figure 4.3). This specification serves as ground truth for the phone app to verify the UI integrity on the host's screen. The specification can either be provided by developers of the remote service or automatically generated by the INTEGRISCREEN server component.



(a) App verifies all UI elements, instructs users to input data.

(b) Server comparison mismatch shown on the smartphone.

Figure 4.4: User experience of the smartphone application.

4.4.1.2 *Input trace matching*

IntegriScreen requires a server-side component that carries out the *input trace matching* between i) the user input from the HTTP payload sent by the browser, and ii) signed input from the IntegriScreen TLS payload. One such example is illustrated in Figure 4.2. If these two traces mismatch, the IntegriScreen server-side component notifies the IntegriScreen phone app.

4.4.2 IntegriScreen *Application*

The app provides a second-factor confirmation of the user's intended input. This is achieved by verifying that the user interface matches its specification during user interaction, supervising against any on-screen modification attacks, and capturing the data shown on the screen to generate and send a respective proof-of-intent. After the user starts IntegriScreen app, it performs the following steps:

1. *Locates* the border of the web form on the host's display, realigns the captured video feed to a flat perspective (as shown in Figure 4.4a), and extracts the form's unique title.

2. *Loads* the corresponding UI specification file sent by the server. The app verifies the signature.

3. *Continuously verifies that the UI* of the web form shown on the host device matches its specification, i.e., that all UI elements are present and that none have been modified or added.

4. *Continuously supervises user input*, allowing only the element in focus to change, only when the user is present and active.

5. *Submits* the generated proof-of-intent to the server. Note that this data is signed and replay protected with a nonce.

6. *Notifies* the user about the result of the server's data comparison: either success (if the host and smartphone-submitted data match) or failure (in case of data mismatch). In the latter case, the user can choose one of two versions of submitted data (or a combination thereof). For additional security guarantees, the user confirms her choice in a hardware-protected user interface which is available since Android 9 [120].

We will now expand on steps *(3)* and *(4)*: the core of the IntegriScreen mobile app, as they are continuously executed for each frame that the app captures to protect IO integrity.

4.4.2.1 *Continuous UI Verification*

Initially, the app downloads the UI specification from the server and validates its signature. Afterward, it checks if the UI elements captured by the camera *comply* with the specification. The form name is a unique identifier (refer to Section 4.4.1.1) that ensures that the phone app uses the appropriate specification for the form shown on the host's screen. In the case of a mismatch, the application warns the user of a potential attack, both visually and by ringing or vibrating. The application clearly augments its preview of the screen (refer to Figure 4.4b) to show which elements are problematic (showing them in red) and prevents any data input until the mismatch is corrected.

4.4.2.2 *Continuous Input Supervision*

Besides the layout of the UI, the app supervises UI changes continuously on the form to make sure it is a result of intended user interaction:

Only the element in focus can change. IntegriScreen app can detect which UI element is in focus. The app mandates that except for the element in focus, all other elements must remain the same. This ensures that the user needs to only pay attention to the value of the currently active element (which they are editing), while all other elements are *protected*.

Activity detection. If the value of the active element changes, the app also ensures that the user is present by detecting the user's hands from the camera feed.

Focus cool-down time. If the value of some active element changes, the focus should not change to another element in less than x ms after the last edit and in less than y ms since this element first came into focus. The server can optionally set the values of x and y in the specification file. In our prototype, we set $x = 300$ ms and $y = 2$ s. This mechanism serves two goals: (i) ensures that any change can be correctly detected, given frame rate limitations of the mobile app; and (ii) ensures that the attacker can not quickly move the focus to another element and change its value without the user noticing. We experimentally evaluate these assumptions in Section 4.6 and show an example of input supervision detecting malicious modification in Figure 4.4b.

Supervised form submission. Furthermore, to prevent the attacker from prematurely submitting the form, the proof-of-intent is submitted to the server only after the user explicitly presses the *Submit* button on the mobile device.

Occlusion and multi-page forms. Our system design supports user interactions in which the form is temporarily occluded (e.g., changing browser tab, minimizing the browser window). In such cases, UI verification temporarily fails but will be resumed as soon as the form is displayed again on the host's screen: if the values of all elements are unchanged, user input is allowed again. Such a design also supports multi-page interfaces, as the application will simply store the values of all input elements on each page

as the user edits them in arbitrary order. Changes to hidden elements are rejected, and the value is verified every time the element is displayed again.

4.5 SECURITY ANALYSIS

We now informally analyze how IntegriScreen provides authenticated user input under our strong attacker model, from the general setting to more specific attacks. We assume that the user is not a victim of a targeted attack rather than a widespread vulnerability in her OS/applications – hence it is safe to assume that not all of the user devices are compromised at the same time. Such trust assumption is valid in any system that involves two factors, such as OTP. Note that all the data communication between the IntegriScreen phone app and the server-side component is authenticated and replay protected (Cf. Section 4.4.2). Hence, any modification or relay attempt by the attacker can be detected by both the phone app and the server-side component.

4.5.1 *UI manipulation attacks*

An attacker-controlled host can edit the web form shown to the user on-screen and change its context, to manipulate the user to input the malicious data himself. IntegriScreen prevents such attacks by ensuring that the web form shown to the user complies with the specification: all labels must show the correct text, all default values of input elements must be present (as their modification could also misguide the user), and no unexpected text is allowed in the rendered web form. If any of those requirements are not met, the smartphone app clearly shows the offending UI element to the user and does not accept any new input. The attacker can also change some of the form's visual instrumentation (Cf. Section 4.4.1.1), which leads to the IntegriScreen app not being able to detect the form. However, this attack causes denial of service (DoS) and it is outside the scope of this chapter. We experimentally measure the performance of this UI verification in Section 4.6.1 and discuss potential extensions to non-textual elements in Section 4.7.

MODIFYING THE FORM HEADER. Since the smartphone app relies on optical recognition of the form title to detect which form specification to load, the attacker can cause the smartphone app to load a form specification that he fully controls by changing the title name. Note that this only results

in a DoS attack: the application uses the same endpoint to load the specification and to submit the proof-of-intent; thus, the original server endpoint never receives a matching proof-of-intent and the attack is not successful. Similarly, attack vectors exploiting typosquatted domains or phishing cannot trick the smartphone app into submitting valid proofs-of-intent to legit endpoints for transactions not performed by users.

4.5.2 *On-screen data modification*

Another strategy for the attacker to evade the defense mechanisms is to modify the data in form elements while the user fills another field. Below we discuss different attack scenarios and explain how IntegriScreen prevents them.

CONCURRENT DATA MODIFICATION. The attacker may try to modify the data of a text field when the user in the process of filling up another. As described in Section 4.4.2.2, IntegriScreen phone app ensures that it only records the input data from the screen that is in focus and filled up by the user. In case of concurrent modification of the active input element by both the attacker and the user, we assume that the user detects such changes (which are similar to autocorrect not behaving according to the user's expectation) and will not move the focus to the next input element until they are satisfied with its content.

RAPID CHANGE OF FOCUS. A potential attack is to change the focused element from one element to another, modify the data there and go back to the first one. This entire process could be extremely fast without the user noticing it in order to evade the concurrent form modification protection. However, the cool-down time for focus change (Cf.Section 4.4.2.2) prevents such attacks. The users are thus likely to detect such sudden changes in focus during data input (for at least 2 seconds).

MODIFICATION DURING USER ABSENCE. The attacker may trigger some modification in the form when the user is absent. However, the IntegriScreen app also detects the user's hand from the camera stream to make sure that the user is physically present. We discuss other ways to implement this step in Section 4.7.

4.6 EXPERIMENTAL EVALUATION

We now evaluate the effectiveness of IntegriScreen, by running a series of experimental tests against potential attacks.

PROTOTYPE IMPLEMENTATION. We developed a smartphone app in Android with the Text Recognition API [121] for optical character detection and recognition, and OpenCV [122] for the rest of the image processing functionality (e.g., detection of form boundaries and perspective realignment). Users press a button to start the form supervision; the app then augments the camera feed indicating whether verification is having success or there are mismatches or failures (Figure 4.4); finally, users press another button to submit the generated proof-of-intent to the server. The prototype server is implemented using the Apache Tomcat 9.0 framework [123]. The lower part of the camera feed detects hand activity by filtering out the background of the keyboard and detecting significant changes between consecutive frames.

4.6.1 *Verification of Loaded Web Forms*

We first evaluate the baseline accuracy of UI verification (i.e., the integrity of elements of the web form) without any attack.

SETUP. We use 100 randomly generated web forms (similar to the one in Figure 4.4b), varying in the complexity of visual elements (between 4 and 9) and types of data as labels and default input values (English words, numerical, and random alphabetical strings). Each form in the dataset is displayed for exactly 5 seconds. During this time, the smartphone app automatically detects the form based on its title, loads its specification from the server, and performs UI verification. Mismatching forms are stored for later analysis.

We use three Android devices: Samsung Galaxy S9+, Google Pixel 2XL, and Samsung Galaxy S6. Each device is evaluated in two positions: i) in front of the screen (straight setup) and ii) to the right of the keyboard, observing the host's screen with an angle (inclined setup).

RESULTS. Table 4.1 shows the results of verifying the UI for all randomly generated forms. The highest recognition rate was achieved by Samsung Galaxy S9+, which was successful in detecting, loading the specification,

	Mobile Device	Forms	Elements
Straight Setup (Fig. 4.4a)	Samsung Galaxy S9+	98%	99.75%
	Google Pixel 2XL	93%	97.86%
	Samsung Galaxy S6	82%	95.15%
Inclined Setup	Samsung G. S9+	93%	99.12%
	Samsung G. S9+ [3 seconds]	93%	98.97%

TABLE 4.1: **Success rates of UI Verification** on 100 randomly generated forms displayed for 5 seconds, and overall percentage of correctly detected UI elements.

and *verifying the text values of 98%* of the forms in less than 5 seconds. This translates to 99.75% of the UI elements being correctly detected and recognized on the screen.

All three devices achieved stable performance, with average processing rates of 2.6 (Samsung S6), 3.3 (for Google Pixel 2XL), and 4.7 (for Samsung S9+) frames per second.

Note that the errors in this experiment on randomly generated forms that often featured atypical designs (e.g., many elements close to each other) do not immediately translate to false positives in real-world forms, with more regular designs and better spacing within elements.

POSITIONING AND VERIFICATION TIME. Table 4.1 also shows the performance of the UI verification procedure when the mobile device is on the right side of the host's keyboard (inclined setup), resulting in a significant angle between the camera and the host's screen. Our evaluation shows that realigning and then detecting elements with such a large angle is successful at *correctly verifying 93%* of the forms from the dataset, resulting in an overall per-element detection rate of 99.12%.

Finally, when the total time allowed for the application to verify a single form is reduced to 3 seconds, the application maintains a high detection rate: the form verification rate for this short duration remains at a high 93%, with a per-element detection rate of 98.97%.

4.6.2 *Preventing Edits To Displayed Data*

We now evaluate the ability of IntegriScreen to detect modification of an element that is not a result of user activity, i.e., one that happens either during page load or at the time of user input but outside of the focused element.

SETUP. To achieve testing consistency and allow running a large number of controlled experiments, we simulate user input with Selenium WebDriver [124], a UI testing framework. We evaluate potential UI manipulation and on-screen data modification attacks by loading randomly-generated forms and simulating user input by an average *touch* typist (120-200 ms per character) [125]. During user input, the attacker replaces three subsequent random characters, that can be either:

1. *Concurrent modification.* Changing an input element not in focus, concurrently with the simulated user input.
2. *Modification before input.* Changing the value of an element while the form is being loaded.

4.7 DISCUSSION

DETECTING USER ATTENTION AND NON-REPUDIATION. In this chapter, the system uses hand movement to detect the user's presence and activity. However, when the mobile device is placed between the host and the user, its front-facing camera is well-positioned to capture the user's face. Given the face tracking capabilities available in recent iOS and Android mobile phones, as well as recent advances in mobile camera-based eye tracking [126], the system could be extended to precisely track user's attention on the screen and require that gaze is present for certain data modification. Furthermore, if the mobile device used face recognition to continuously authenticate the user, the system could also provide non-repudiation guarantees.

NON-TEXTUAL UI ELEMENTS. As the first of the proposed *visual supervision* paradigm, in this chapter, we focused on textual input. However, our approach can be extended to support non-textual UI elements – as long as their final state is shown on the screen – such as checkboxes, sliders, or calendar widgets. Furthermore, while we focused on text extraction, this

step could be implemented by a more literal comparison of the host's screen with a screenshot of the web form, as rendered by the server with the same aspect ratio and element position.

PRIVACY AND SECURITY OF VISUAL SUPERVISION. While continuously recording one's interaction with another electronic device seems intrusive, we note that all processing in IntegriScreen happens on the mobile device. Therefore, the server only receives a duplicate of the data from the host. If sending a duplicate of the data is not suitable for any reason, it is straightforward to modify IntegriScreen to compute and only send a digest (similar to a TAN) to the server.

However, a visual channel gives users clear control over which data the mobile device observes, i.e., only what is shown on the screen at a given moment – while most kernel-level applications typically get unrestricted access to the whole system and could violate the users' privacy in the background while keeping them oblivious.

Finally, using only a visual channel between the host and the mobile device reduces the likelihood of a smartphone compromise since it never directly communicates with the already compromised host.

4.8 RELATED WORK

Previous work on the trusted path either relies on a trusted hypervisor that supervises a compromised virtual machine or on the use of another trusted device that serves as a second factor.

TRUSTED HYPERVISORS. Trusted hypervisors and secure micro-kernels are a possible choice for the trusted path. Sel4 [127] is a functional hypervisor that is formally verified and has a kernel size of only 8400 lines of code. In work done by Zhou et al. [9], the authors proposed a generic trusted path on *x*86 systems in pure hypervisor-based design. Examples of other hypervisor-based works can be found in systems such as Overshadow [128], Virtual ghost [129], Inktag [130], TrustVisor [131], Splitting interfaces [132], SP^3 [133].

Our approach is most similar to Gyrus [33], a system that enforces the integrity of user-generated network traffic of protected applications by comparing it with the text values displayed on the screen by the untrusted VM. However, Gyrus requires application-specific logic and does not prevent potential UI manipulation attacks.

Not-A-Bot (NAB) [34] ensures that the data received from the client was indeed generated by the user rather than malware by having the server require a proof (generated by a trusted *attester* application) of the user's keyboard or mouse activity shortly before each request. Similarly, BINDER [98] focuses on detecting malware break-ins and preventing data exfiltration by implementing a set of rules that correlate user input with outbound connections. While these approaches are similar to IntegriScreen in ensuring that outgoing requests match the user's activity on the client device, our solution differs in that it allows for a fully compromised client.

TRUSTED DEVICES. The assumption of a fully compromised client mandates an additional trusted device is used to secure the interaction with the remote server. Weigold et al. propose ZTIC [28], a device with simple user input and display capabilities, on which users confirm summary details for a banking transaction. Another approach is taken by Kiljan et al. [97], who propose a simple *Trusted Entry Pad*, that computes signatures of user-input sensitive values and sends them independently to the server for verification. However, such approaches either require the user to input data to an external device, which breaks the normal workflow and duplicates efforts or require the user to confirm transaction details. This leads to habituation and decreases security. The approach of continuous visual supervision improves on previous work by neither requiring additional input nor relying on user attentiveness during transaction confirmation.

4.9 CONCLUSION

In this chapter, we explore the idea of *visually supervising* user's IO data to provide IO integrity against a compromised host. We use a smartphone to capture the client's screen and enforce the integrity of the web rendered on the screen, alongside the input data the user submits on the web form. We show the feasibility of this approach by developing a fully functional prototype on an Android smartphone, evaluating it with a series of experimental tests, and running a user study to measure participants' responses to simulated attacks. Considering the rapid increase in processing power and camera quality of smartphones, but also novel platforms such as augmented reality headsets and smart home assistants, we envision such systems that supervise user's IO will be ubiquitous.

Even though IntegriScreen addresses the problem of input and output integrity and makes significant progress over IntegriScreen, Inte-

GRISCREEN is still not a completely secure and usable system. Usage of smartphones significantly increases the TCB of the system. Moreover, INTEGRISCREEN only considers keyboard input which opens up new attack surfaces. Despite INTEGRIKEY and INTEGRISCREEN provides a significant advantage over their predecessors, both follow the similar system-building approach of the existing trusted path proposals. Such a mere system-building approach does not solve the fundamental problem of a trusted path. Hence, we conclude this first part of this thesis by identifying *how not to build a trusted path*.

Part III

FUNDAMENTALS OF TRUSTED PATH

5

PROTECTION: ROOT-OF-TRUST FOR IO IN COMPROMISED PLATFORMS

Success is sometimes the outcome of a whole string of failures.

— Vincent Van Gogh

5.1 INTRODUCTION

As we discuss in the earlier chapters, web-based interfaces are very prevalent to remotely configure safety-critical systems such as remote PLCs [3, 134, 135] or medical devices [4], and other security-sensitive applications such as online payments, e-voting, etc. The high complexity of modern operating systems, software and hardware components has shown that computer systems largely remain vulnerable to attacks. A compromised computer threatens the integrity and the confidentiality of any interaction between the user and a remote server. It can easily observe and/or manipulate the sensitive IO data exchanged between the user and the remote server or even trick the user into performing unintended actions.

Trusted path provides a secure channel between the user (specifically human interface device - HID) and the end-point, which is typically a trustworthy application running on the host. A trusted path ensures that user inputs reach the intended application unmodified, and all the outputs presented to the user are generated by the legitimate application. A trusted path to the local host is a well-researched area where many solutions focus on using trusted software components such as a trusted hypervisor. Zhou et al. [9] proposed a generic trusted path on *x*86 systems with a pure hypervisor-based design. SGXIO [36] employs both a hypervisor and Intel SGX. However, hypervisors are hard to deploy, have a large TCB, and are impractical in real-world scenarios as most of the existing verified hypervisors offer a minimal set of features.

The recent introduction of trusted computing architectures like Intel's SGX has enabled secure computations and secure data storage on otherwise untrusted computing platforms. However, such architectures do not directly enable secure user interaction because IO operations are handled by the operating system. Additionally, the recent microarchitectural attacks have

shown that execution environments inside enclaves, like the one provided by SGX, can be compromised as well.

Trusted external devices are another way to realize secure IO between a user and a remote server. Transaction confirmation devices [10, 11] allow the user to review her input data on a trusted device that is physically separated from the untrusted host. These approaches suffer from poor usability, security issues due to user habituation and are only limited to simple inputs. In Section 5.2.2, we provide a more detailed discussion on the security and the usability of transaction confirmation devices. Bump in the Ether [12], and IntegriKey(refer to Chapter 3) use external embedded devices to sign input parameters. However, such solutions do not support output integrity; hence, the attacker can execute UI manipulation attacks to trick the user into providing incorrect inputs.

Fidelius [14] combines the previous ideas of Bump in the Ether and trusted overlay to protect keyboard inputs from a compromised browser using external devices and a JavaScript interpreter that runs inside an SGX enclave. Fidelius maintains overlays on display, specifically on the input text boxes, to hide sensitive user inputs from the browser. We investigate the security of Fidelius and discover several issues. Fidelius imposes a high cognitive load on the users as they need to monitor continuously different security indicators (two LED lights and the status bar on the screen) to guarantee the integrity and confidentiality of the input. Furthermore, the attacker can manipulate labels of the UI elements to trick the user into providing incorrect input. The lack of mouse support, which may appear only as a functional limitation, exposes Fidelius to early form submission attacks. The host can emulate a mouse click on the submit button before the user completes all fields of a form. This allows the attacker to perform an early form submission with incomplete input - a violation of input integrity. Fidelius is also vulnerable to microarchitectural attacks on SGX enclaves [136] that extract attestation keys and relay attacks [137] that redirects all user data to the attacker's platform.

The drawbacks of the existing systems show that ensuring the integrity and confidentiality of the IO in the presence of an untrusted host is a non-trivial problem and requires a comprehensive solution. All of the previous trusted path solutions neither protect both input and output simultaneously nor do they consider different modalities of input. We discuss such drawbacks in detail, along with some of the relevant solutions in Section 5.2.2.

5.1.1 *Our solution*

The shortcomings of the existing literature provide the groundwork of our system named ProtectIOn. ProtectIOn is built on the following observations: i) input integrity is possible only when both input and output integrity are ensured simultaneously, ii) all the input modalities are needed to be protected as they influence each other, and iii) high cognitive load results in user habituation errors. ProtectIOn uses a trusted low-TCB auxiliary device that we call IOHub, which works as a mediator between all user IO devices and the untrusted host. Instead of implementing a separate network interface, the IOHub uses the host as an untrusted transport – reducing the attack surface.

INTEGRITY. ProtectIOn ensures *output integrity* by sending an encoded UI to the host that only the IOHub can overlay on a section of the screen. The overlay is possible as the IOHub intercepts the display signal between the host and the monitor. The overlay generated by the IOHub ensures that the host cannot manipulate any output information on that overlaid part of the screen; hence, it can not trick the user. IOHub supports a subset of HTML5 UI elements that are frequently used in the majority of web applications. The IOHub focuses user attention on the overlaid part of the screen by dimming out the rest (also known as the lightbox technique, which is one of the possible ways to focus user attention) when the user moves the mouse pointer on the overlaid UI. By doing so, ProtectIOn aids the user to be more attentive to the security-critical UI on the screen. Note that ProtectIOn does not require any change in the user interaction for IO integrity. Only the input devices that are connected to the IOHub can interact with the overlaid UI elements, making them completely isolated from the untrusted host. All the inputs are signed by the IOHub and sent to the remote server - ensuring input integrity.

CONFIDENTIALITY. ProtectIOn provides IO confidentiality as i) all the input to the IOHub is encrypted and signed, and ii) the overlay information sent from the remote server is encrypted and can only be decrypted by the IOHub. However, the user needs to perform a small task such as triggering a secure attention sequence (SAS) or looking for a secret image, security indicator, etc., to distinguish the trusted overlay.

DEPLOYMENT. IOHUB is a fully plug-and-play device that is compatible with any host system regardless of their architecture or OS and does not require the user to install any software on the host. Note that our realization of PROTECTION uses an external device. However, the current system architecture can be modified, e.g., IOHUB can be integrated into the graphics processor.

5.1.2 *Our contributions*

In summary, we make the following contributions:

1. **Identification of IO security requirements:** We identify new requirements for trusted path based on the drawbacks of the existing literature: i) unless both output and input integrity are secured simultaneously, it is impossible to achieve any of the two, and ii) without protecting the integrity of all the modalities of inputs, none could be achieved (Section 5.2.2).

2. **System for IO integrity:** We describe the design of PROTECTION, a system that provides a remote trusted path from the server to the user in an attacker-controlled environment. The design of PROTECTION leverages a small, low-TCB auxiliary device that acts as a *root-of-trust* for the IO. PROTECTION ensures the integrity of the UI, specifically the integrity of mouse pointer and keyboard input. PROTECTION is further designed to avoid user habituation (Sections 5.3 and 5.4).

3. **System for IO confidentiality:** We also describe an extension of PROTECTION that provides IO confidentiality, where user needs to execute an operation like SAS to identify the trusted overlay on the display (Section 5.5).

4. **Implementation and evaluation:** We also implement a prototype of PROTECTION and evaluate its performance (Sections 5.7, and 5.8).

5.1.3 *Organization of this chapter*

The organization of this chapter is as the following. Section 5.2 provides the detailed motivation, problem statement, state-of-the-art and the goals of this chapter. Section 5.3 provides the system & attacker model, challenges and a brief overview of our solution. We discussed the technical details of PROTECTION in Section 5.4. Section 5.6 provides in-depth security analysis of

ProtectIOn. Section 5.7 and 5.8 provide details of ProtectIOn prototype implementation and corresponding evaluation. Finally, Section 5.9 and 5.10 provides the related research works and concludes this chapter respectively.

5.2 PROBLEM STATEMENT

In this section, we motivate our work to ensure the integrity and confidentiality of IO data between the user and the remote servers. We also analyze existing research works that tackle the relevant problem. We explain how these works lack a proper solution and report the observations we derive from them. Lastly, we present the required security properties of ProtectIOn that we obtain from the observations.

5.2.1 *Motivation: Secure IO with Remote Safety-critical System*

A user communicates with a remote server through a *host* system that is typically a standard PC (specifically *x*86 architecture), which gives the host access to the raw IO data that is exchanged between the user and the remote server. The host consists of large and complex system software such as the operating system, device drivers, applications such as a browser, and a diverse set of hardware components that expose the host to a large attack surface. Due to cost and convenience, general-purpose PCs are prevalent in many safety-critical application domains such as industrial plants and hospitals. For example, the WannaCry ransomware incident showed that NHS hospitals relied on Windows XP platforms [5, 6].

An adversary that controls the user's host can alter user intentions, i.e., it can perform arbitrary actions on behalf of the user, modify the input parameters, or show wrong information to the user. Such an adversary is very powerful and difficult to be detected or prevented by a remote server. Hence, existing defense standards for web UI are ineffective as the browser is untrusted also. The consequences of such attacks might be severe when applications that control remote safety-critical systems are targeted. The attacker can pass the wrong input to a remote safety-critical system such as a medical device, power plant, etc., or leak sensitive information such as credentials for e-banking, candidate preference in the e-voting, etc.

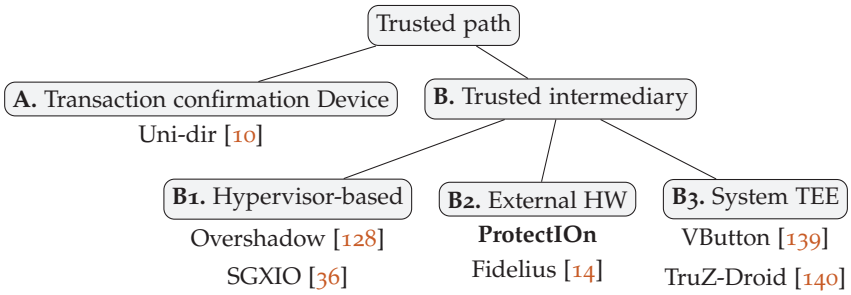


FIGURE 5.1: **Existing trusted path solutions.** Here, we classify some of the existing trusted path works, including our proposal in this chapter.

5.2.2 *Analysis of Existing and Strawman Solutions*

There are two broad categories of existing solutions that address the problem of trusted paths for IO devices in the presence of a compromised host, as illustrated in Figure 5.1. **A.** Solutions where unprotected user interaction first happens and then a trusted component (transaction confirmation device) is used to ensure input integrity. **B.** Solutions where a trusted component captures the user's input/output and then securely mediates them to the destination. The trusted component can be a hypervisor, or external hardware, etc.

A. TRANSACTION CONFIRMATION DEVICES. Filyanov et al. [10] proposed a transaction confirmation device that requires the user to use a separate device to confirm the input parameters. Systems such as ZTIC [11] use an external device with display and smartcard attachment to ensure the integrity of the user inputs. Android OS also provides a similar mechanism to confirm protected transactions [138]. However, these approaches suffer from three significant drawbacks: i) the risk of *user habituation* – users confirming transactions without looking to the actual data [15], ii) *usability* – interacting with a small device can be cumbersome, and iii) only *simple UI* can be supported – transaction confirmation is not suitable for complex interaction, rather than simple text-based inputs.

B1. TRUSTED HYPERVISOR-BASED SOLUTIONS. Trusted hypervisors and secure micro-kernels are also alternatives to achieve a Trusted path. Zhou et al. [9] proposed a generic trusted path on *x*86 systems in pure

hypervisor-based design. SGXIO [36] combines a TEE and a hypervisor to mitigate the shortcomings of TEEs like SGX (e.g., OS controls the IO operations). Nevertheless, solutions based on hypervisors require a large TCB. Formally verified hypervisors offer limited functionalities, therefore making them impractical for average users. One can also argue that a hypervisor that provides a rich set of functionalities has a code size comparable to an actual OS. Also, systems employing TEEs such as Intel SGX open up new attack surfaces that can be exploited by microarchitectural attacks [136]. SecApps [141] uses a trusted GPU kernel and a micro-hypervisor to reserve a part of the GPU memory for trusted display.

B2. EXTERNAL HARDWARE-BASED SOLUTIONS. Several existing works propose a trusted path that utilizes an external trusted device. IntegriKey [114] uses a trusted external device that contains a small program that signs all user inputs and sends the signed input to the remote server. The device works as a second factor for input integrity as the remote server verifies if the signed input matches the input sent by the browser running on the untrusted host. However, as the external device is completely oblivious to the display information that the untrusted host renders, IntegriKey and similar systems that do not consider output integrity are vulnerable to UI manipulation attacks. For example, assume that the user's intended input to a textbox is 100. She types the correct value, but the host maliciously renders 10 on the screen by not showing the last zero. Thinking that she might have mistyped, the user types another 0 that makes the recorded input from the user 1000. This attack violates input integrity as the host can now submit 1000 to the remote server as a valid input, although it does not represent the user's intention.

Observation 1

The lack of output integrity – *the render of user inputs on the screen* – compromises input integrity.

Fidelius [14] addresses the problem with output integrity by rendering overlays using an external trusted device. Fidelius uses the trusted external device and Intel SGX to create a secure channel between the user IO devices and a remote server. The device intercepts user keystrokes and does not deliver any event to the untrusted host when the user types to secured text fields. Additionally, Fidelius renders an overlay with the user inputs on the screen, which is inaccessible by the host. This way, the untrusted

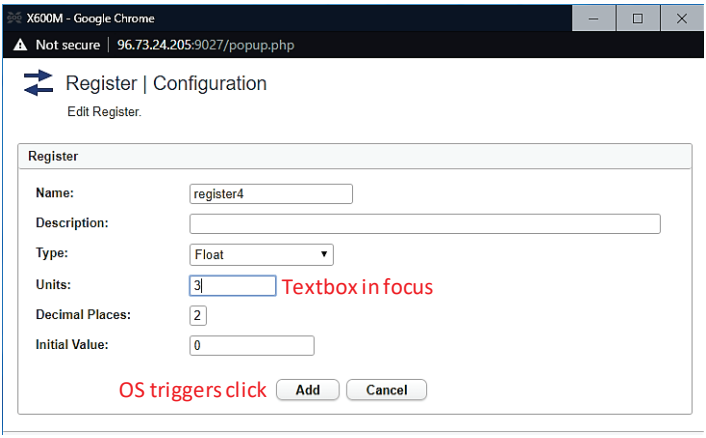


FIGURE 5.2: **Early form submission attack** is possible on Fidelius [14]. The user selects and edits the field *Units* while the OS triggers *add* button, causing misconfiguration of a remote safety-critical PLC (Control by Web X-600M [3]).

host does not have access to raw inputs while the user sees them rendered on the screen as usual. A small, trusted bar on display is also overlaid by the device that shows the remote server's identity and the text field that is currently selected. However, we observe a number of security and functional issues in Fidelius that we explain in the following.

The overlay contains only the render of the user inputs into text fields, but the rest of the screen is rendered by the untrusted host. This allows an attacker to modify the instructions on the UI, such as changing the input unit (typically described in the label of a text field) that could result in an incorrect input. This problem could be mitigated if the trusted bar includes the legitimate labels of the text fields also, although it would significantly increase the cognitive load to users.

Fidelius already introduces a high cognitive load to users as they need to monitor multiple security indicators simultaneously before filling up one text field. Previous research works [15, 142, 143] have shown that systems that require users to observe multiple security indicators do not guarantee security in practice. Also, in specific scenarios, even the training to properly explain these indicators to users could be a significant drawback for a real deployment.

Observation 2

If the *protected output* is provided out-of-context, users are more likely not to verify it. Therefore input integrity can be violated.

Fidelius does not consider the integrity of the mouse pointer and its interaction with UI elements, which broadens the attack surface. The lack of mouse support may appear to be a functional limitation, but it has non-trivial security issues. The OS can arbitrarily trigger a mouse click on the submit button of a form while the user is typing and therefore send incomplete data to the server - early form submission attack. This attack could cause the misconfiguration of a remote system, as illustrated in Figure 5.2. Early form submission may appear to be similar to clickJacking attack, but the fundamental difference between them is that in clickjacking, the browser and OS are considered to be trusted. An untrusted OS can simply issue mouse clicks.

Moreover, Fidelius is also vulnerable to clickjacking attacks where the attacker can spawn a fake mouse pointer and trick the user into following it while the real mouse pointer is on a sensitive text field protected by the system. This allows the attacker to fool the user into providing (possibly incorrect) input while the user thinks that she is interacting with a non-sensitive text field. To prevent such attacks, the user has to look at the security indicators continuously even when she is not doing any security-sensitive task, which is a very strong assumption. Thus, not supporting the mouse causes the integrity violation of the keyboard input also.

Observation 3

If not *all the modalities of inputs* are secured simultaneously, none of them can be fully secured.

Finally, the design of Fidelius [14] is strictly limited to text-based fields only. As Fidelius does not provide output integrity of the forms, it cannot provide confidentiality to other UI elements such as radio buttons, drop-down menus, sliders, etc. Microarchitectural attacks on Intel SGX [136] increase the attack surface of the system significantly.

B3. SYSTEM TEE-BASED SOLUTIONS. VButton [139] uses ARM TrustZone (TZ) to render UI buttons and receive user input from them securely. This is possible on mobile devices because the TZ architecture support

flags on the system bus indicate whether an IO device like a touchscreen communicates with a trusted TZ application or the untrusted OS. Such solutions are infeasible for us because of the two following reasons. I) Secure communication between IO peripherals and TEE applications (like SGX enclaves) is not supported in the x86 architecture – a similar system in x86 would require changes to the system architecture, TEE architecture, and IO devices. II) such solutions require TEE-aware applications and do not work with current browsers. Our goal is to design a solution that can be deployed on current x86 architecture and used with existing popular browsers.

STRAWMAN SOLUTION: CAPTURING SCREENSHOT. This strawman solution uses a trusted device that takes a screenshot when the user executes an action, e.g., mouse click to submit a form. The device then signs the snapshot and transmits it to the server along with the signed input. The remote server verifies the signature and then uses image/text analysis to extract the information from the UI elements such as labels on buttons or markers of a slider, etc. Therefore, the server would detect if the host has manipulated UI elements when presented to the user.

This method is vulnerable to attacks because it does not capture the spatio-temporal user context. This implies that the attacker may show some spatial information on the screen to influence the user that the snapshot may not capture. Furthermore, taking a full-screen snapshot could also reveal private information of the user from other applications. Similarly, taking a snapshot does not guarantee that a specific UI has been presented on the screen, as the attacker may render the legitimate UI shortly before the device captures the snapshot. One way to mitigate this problem is to capture a video of user interaction. But such a method requires the host to send large amounts of data to the server, while the server should support video processing for different browsers, which is both time and CPU-intensive. Lastly, adversarial machine learning techniques [144, 145] make the image/text recognition techniques insecure against advanced adversaries.

5.2.3 *Requirements of Security and Functional Properties*

We can summarize the above-discussed limitations of previous solutions as the following requirements for our solution:

R1. INTER-DEPENDENCY BETWEEN INPUT AND OUTPUT. The first and second observations from the existing solutions show that the output and input security depend on each other, and they should be considered together. Otherwise, the attacker can manipulate the output to influence the user input.

R2. INTER-DEPENDENCY BETWEEN ALL INPUT MODALITIES. Existing web interfaces allow users to complete forms using different modalities for user input, namely the keyboard, the mouse, and the touchpad. The third observation shows that a secure system should simultaneously protect all user input modalities to achieve input integrity (against early-form submission and clickjacking).

R3A. NO COGNITIVE LOAD FOR IO INTEGRITY. A system that protects IO operations should introduce minimal or no cognitive load to its users for input integrity. The system should guarantee the output integrity of the legitimate information necessary to complete a form and avoid asking the user to interact with an external device or monitor security indicators out-of-context.

R3B. USER ATTENTION FOR IO CONFIDENTIALITY. Preserving the confidentiality of user inputs against a compromised host is a challenging task because the host can trick the user into revealing her inputs when the system is not active. Therefore, requiring users to perform a small action, e.g., press a key before entering confidential inputs, is a valid tradeoff between usability and security.

R4. SMALL TRUST ASSUMPTIONS AND DEPLOYABILITY. Our goal is to provide a rich set of IO and security features with minimal trust assumptions that do not rely on a trusted OS, specialized hypervisor, or TEEs such as Intel SGX. Preferably, the solution should be easy to set up for users, i.e., plug-and-play, and integrate well with the existing infrastructure.

5.3 SYSTEM OVERVIEW & MAIN TECHNIQUES

In this section, we present an overview of our solution: PROTECTION. On the high-level, PROTECTION uses the concept of the *bump in the wire* (such as bump in the ether [12]) to provide integrity and confidentiality to the user IOs between the IO devices and the remote server. PROTECTION achieves

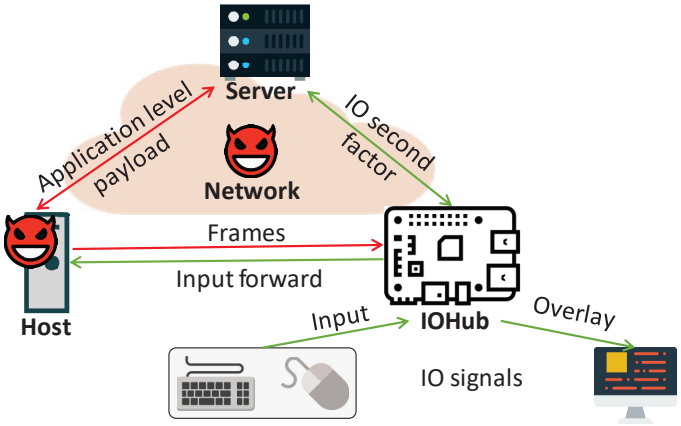


FIGURE 5.3: **High-level approach overview of our solution.** The IOHUB connects the trusted IO devices and the attacker-controlled host. The IOHUB intercepts all the IO communication between the IO devices and the host. Additionally, the IOHUB establishes a secure channel with the remote server using the host as the untrusted transport. Using this secure channel, the IOHUB receives the secure UI information from the server, which is overlaid in the intercepted display signal (frames).

this by utilizing a trusted embedded device as a mediator between all the IO devices and the untrusted host. Hence, our approach falls into the category **B2** (external HW) in Figure 5.1. We call this trusted intermediary IOHUB.

5.3.1 *System and Attacker Model*

We consider a typical scenario where the user wants to interact with a trusted remote web server via an attacker-controlled host. The model is depicted in Figure 5.3, which shows the untrusted host, the remote server, and the user IO devices. We only assume that the monitor, keyboard, mouse (in a word, all the IO devices that we need to protect from the malicious host), and the IOHUB are trusted. One benefit of an external trusted device is that regulations may prevent modifications of systems such as medical devices. However, retrofitting them with external devices, such as the IOHub, is usually possible.

The IOHUB works as a mediator between all the IO devices and the host. Note that the IOHUB has no network capability to communicate with the

server directly. Instead, it relies on the host and uses it as an untrusted transport. We also assume that the IOHub comes with preloaded certificates and keys that allow the IOHub to verify the signatures signed by the server and sign data such as the user input.

DEPLOYMENT OPTIONS. There are several possible ways to deploy the ProtectION system. Here we outline two example cases. The first example deployment is one where a service provider, like a bank, issues a IOHub device to each of its customers. In such a deployment, the issued IOHub is intended to be used with a single application like a web-based online banking application, and it is pre-configured with the public key certificate of that application server (e.g., online banking server). The pre-installed certificate allows the IOHub to verify messages signed by the correct application server. The service provider (i.e., the issuer of the IOHub) can ask what OS the customer uses and configure OS-specific settings like the used SAS value to the issued device (see Section 5.5.2 for details).

In another example deployment, the IOHub is issued by a third-party vendor, and it is intended to be used to protect the user interaction of various security-critical online services. In such a deployment, the IOHub can be pre-configured with the public key of its issuer and a white-list of trusted application server certificates. The issuer of the device can issue authenticated updates to the white-list after its deployment if needed.

ATTACKER MODEL AND CAPABILITIES. Our attacker model assumes that the host (OS, installed applications, and hardware) and the network are attacker-controlled. The attacker can intercept and arbitrarily manipulate (such as create, drop, or modify) the user IO data between the user and the remote server. Furthermore, we assume that the attacker can not break the physical security of the IOHub (more discussion in Section 5.6.3).

5.3.2 *High-level Description of the System*

ProtectION is built upon the security requirements and functional properties that are described in Section 5.2.3. IOHub is active only when the user visits sensitive web applications that require ProtectION security. Initially, the remote server signs and delivers the sensitive UI elements to the host in a format that is understandable by IOHub. Next, the host transfers the sensitive UI to IOHub, and the IOHub verifies the signature to prevent manipulations by the host. As seen in a running example depicted in Figure 5.4,

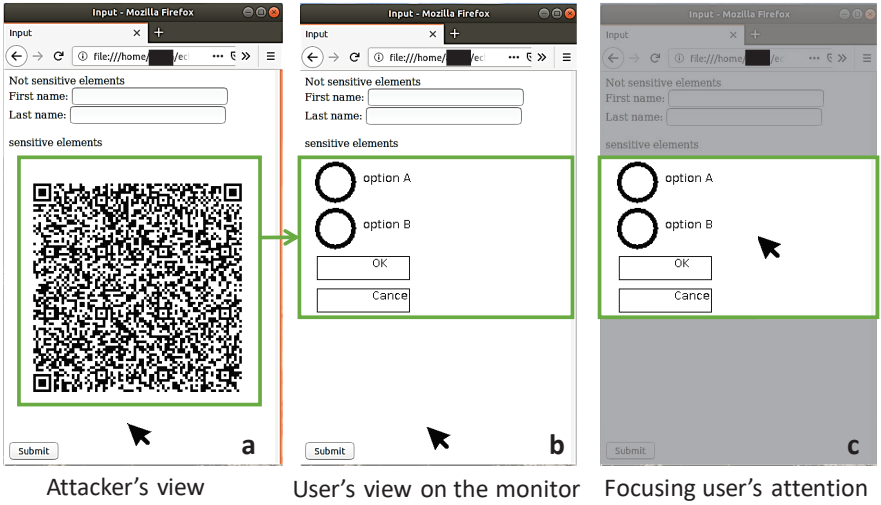


FIGURE 5.4: **ProtectIOn's high-level approach for UI overlays** shows that the IOHUB generates UI overlay to protect IO integrity and confidentiality. a) The attacker only sees the non-protected UI elements, and the protected form is encrypted and encoded (in our case, the IOHUB could decode a QR code and decrypt). b) shows the IOHUB generated form overlay that is hidden from the host. The protected part of the screen provides integrity and confidentiality to all user IO. c) shows that the IOHUB dims out (lightbox) the rest of the screen when the user moves her mouse pointer over the protected region to focus user attention.

the IOHUB then renders the UI with sensitive elements into an overlay on top of the HDMI frame received from the host. Note that the host cannot access or modify the overlay generated by the IOHUB. Also, the overlay covers only a part of the screen, allowing the other feature-rich content on the webpage to run unmodified. Therefore, this ensures that sensitive UI elements are presented to the user as expected by the remote server – *output integrity*. For the overlay, we use QR-codes to transfer data from the host to the device because we avoid using extra software/hardware for a separate channel, and it is easy to visualize.

When the user interacts (types or moves the pointer) with the overlay, IOHUB does not forward any event from the keyboard or the mouse to the host. The interaction is maintained solely by IOHUB, which renders on-screen user inputs and therefore offers a user experience that is identical

to a typical one as if the IOHub is not present. The user clicks on the *submit* button triggers the submission procedure, which consists of the IOHub signing the user inputs and sending them to the server. Note that the text fields of the form and the *submit* button are inside the overlay, which is inaccessible by the host. Hence the attacker cannot execute the early form submission or clickjacking attacks. Finally, the server verifies the signature of IOHub to guarantee that the host has not altered the data. Therefore, the IOHub ensures *input integrity* for all *modalities* of input.

For integrity guarantees, ProtectION uses well-known user attention focusing mechanisms. Unlike systems like Fidelius, these mechanisms do not introduce any cognitive load to the users as ProtectION does not rely on multiple security indicators. Mechanisms such as lightbox aid the user to distinguish the IOHub overlay on the screen from the rest. Thus, the untrusted host cannot trick the user into following malicious instructions when the user interacts with sensitive UI elements. In the case where confidentiality is required, the user manually triggers SAS, using well-known sequences of keys such as *Ctrl+Alt+Del* that highlights the sensitive UIs using mechanisms such as lightbox (see Section 5.5.2 for details). For confidentiality, the host cannot observe the overlay and user input as they are encrypted by the TLS key between the IOHub and the server.

5.4 PROTECTION FOR IO INTEGRITY

In this section, we provide the technical details of ProtectION that guarantees the integrity of IO operations.

5.4.1 IOHub *Overlay of UI Elements*

As we explained in the previous sections, both output and input integrity are necessary to be protected to achieve any of them. ProtectION ensures output integrity by isolating a part of the display that cannot be observed or modified by the untrusted host. IOHub intercepts the HDMI frame from the host and injects a render of the sensitive UI on the screen. The overlay provides output integrity because it restrains the attacker from drawing on top of it to trick the user into providing incorrect inputs.

Our goal is to minimize the TCB, and thus the IOHub does not run a browser, i.e., it cannot interpret or render HTML, JavaScript, etc. Given this constraint, one strawman solution could be sending UI bitmaps from the server to the IOHub, and the IOHub could send back the mouse click and

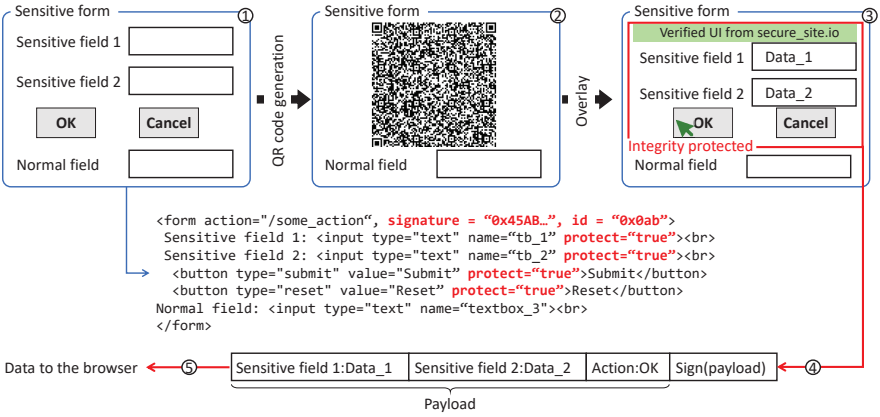


FIGURE 5.5: **Transformation of UI elements: HTML → encoded specification → IOHub generated UI overlay.** ① The actual webpage and the corresponding HTML source shows the UI elements that requires integrity protection. ② These UI elements are transformed into an encoded UI specification (our PROTECTION prototype uses QR code that encodes a UI specification, e.g., Specification 5.1) by the PROTECTION JS. The QR code. ③ AThe QR code decoded and overlaid on the HDMI stream by the IOHUB. ④ Upon the user's action on the overlaid UI elements, the device signs all the input data. ⑤ The IOHUB sends these signed input data them to the remote server. Note that the intermediate QR code transformation (②) is not visible by the user.

the corresponding location. IOHUB could download these pre-generated form bitmaps from the server, and such a solution would handle static forms, but not for dynamic forms. Downloading all possible form bitmaps would be prohibitively expensive.

To achieve a more generic and efficient solution, we follow a different approach. The IOHUB comes with a small interpreter routine that is similar to render engines of browsers in functionality but drastically smaller in size because it only renders a limited number of HTML5 UI elements according to their position, dimension, and label. The interpreter routine reads a given specification and renders the respective UI. The specification is a simple *JSON* file that defines how the content of the overlay should be rendered, e.g., number of elements, order, types, and labels.

The process of rendering the overlay on the screen has two phases: (i) convert the existing sensitive form to specification and (ii) specification to overlay.

```
{"formId": "form1",
 "formName": "form1",
 "domain": "secure_site.io",
 "size": "400*400",
 "SAS": "5:LB",
 "ui": [{
      "id":"textbox_1",
      "type":"textbox",
      "label":"Sensitive field 1",
      "text":"secret data 1",
      "RE":5|(A-Z)+.(A-Z)+},
      {
      "id":"textbox_2",
      "type":"textbox",
      "label":"Sensitive field 2",
      "text":"secret data 2"},
      {
      "id":"button_1",
      "type":"button",
      "label":"OK",
      "trigger":"true"},
      {
      "id":"button_2",
      "type":"button",
      "label":"Cancel",
      "trigger":"false"}],
 "id": "0x0ab..",
 "signature": "0x45AB..."}
```

SPECIFICATION 5.1: **Protected UI specification language.** The UI specification shows the JSON formatted UI specification that is generated from the HTML source provided in the UI illustrated in Figure 5.5.

(1) SECURE FORM → SPECIFICATION. The W3C UI security policy [146] recommends developers to annotate the security-critical UI elements of a page to protect them against malicious JS running on the browser. We use a similar technique by asking developers to manually annotate the sensitive elements in the HTML code (as *protect="true"* attribute). Then For every request, the PROTECTION server-side component parses the HTML source, adds a random identifier (`id`) to the `form` element, signs it, add the `signature` to the `form` and then delivers it to the user's browser. The `id` serves as session identification to prevent the attacker from re-submitting an old input data from the user. On the host-side, PROTECTION JS parses the tagged HTML source and produces an intermediate representation that we call an *UI specification*. This specification can be interpreted by the IOHUB can converted to an equivalent bitmap representation of the corre-

sponding HTML form. One such example of a specification is presented in Specification 5.1. The specification is divided into two parts:

1. *Global form attributes:* The global form attributes describe and define the identity of the form. Our implementation of ProtectION specification contains the following:
    - `formID` is the identifier of the HTML form in the DOM tree.
    - `formName` is the name of the form.
    - `domain` defines where the form originates form. This domain is crucial to determine the corresponding public key to verify the signature of the form.
    - `size` defines the size of the form bitmap that is generated by the IOHub.
    - `SAS` describes the secure attention sequence which is important for confidentiality of the form and the input to it. For more details on confidentiality and SAS, refer to Section 5.5.
    - `id` is a nonce that is injected by the server to identify a specific session with user. This is used as the replay protection for the form.
    - `signature` is the signature of the entire specification signed by the `domain`'s private key.
2. *UI elements:* UI elements on the form is listed under the `ui` attribute. Individual UI elements may contain some of the following UI attributes.
    - `id` is the identifier of the HTML element of the UI element. This id is used as the key to bind the user input data to submit to the remote server.
    - `type` define the type of the element and crucial for the IOHub to generate the correct bitmap representation.
    - `label` is the label of the the UI element that the IOHub use to generate the text label.
    - `text` is the initial text in case the server provide some default value for the corresponding UI element.
    - `RE` is the regular expression for input sanitation.

- `trigger` is important only for the element that either trigger the input to the form to submit (`trigger=true`) to the server ot reset (`trigger=false`) the form to its initial state. More details on `trigger` attribute in Section 5.4.4.

In our implementation, the ProtectIOn JS encodes the specification in a QR code. We choose QR codes to encode UIs as it is one of many robust ways to encode data on a visual channel such as an HDMI stream. Figure 5.5 shows the transformation between the step ① and ②. Step ② is processed by IOHub in the next phase and is not visible to the user.

(ii) specification → overlay. IOHub performs the next phase, which starts with the detection of the encoded specification (QR-code) in the HDMI frames. Then the IOHub validates the signature, renders the overlay according to the specifications, and presents it to the user. The IOHub overlay is depicted in ③ in Figure 5.5, which is the final UI shown to the user. Note that the user does not see the QR code as it gets decoded and overlaid by the IOHub on the fly.

IOHub uses the specification to determine the particular UI element that the user interacts with. When the user clicks on a text field, IOHub allows the user to type input to it. UI elements in the overlay take inputs only from input devices (mouse and keyboard). Therefore a malicious host cannot inject or modify any input of the user.

5.4.2 *Focusing User Attention*

In the previous section, we explain how ProtectIOn provides output integrity for the overlay generated by the IOHub. However, the attacker can show fake information to the user on the untrusted part of the display space that may potentially influence her inputs. An advanced adversary could craft malicious directions and present them to the user as part of the overlay.

To mitigate these attacks, we employ techniques that are proposed against similar threats in the context of browser-based security. The goal of these techniques is to focus user attention on the sensitive UI elements she is interacting with. Huang et al. [147] propose two main techniques that are shown to be effective and can easily be adopted by the IOHub. The first technique is called Lightbox, and it dims out the non-overlaid part of the screen, which is generated by the untrusted host as shown in the rightmost sub-figure in Figure 5.4. The second technique consists of freezing display

frames from the host when the user enters into the overlaid UI. This way, a malicious host cannot grab the user's attention by showing an animation or exploiting other tricks. Lightbox offers more security guarantees because it blocks the untrusted screen completely but is more intrusive to the user. While freezing is less intrusive but does not remove potential malicious information from the screen. Another technique that we can apply is called highlight that enables a highly visible border (typically on bright green or red) around the UI form. A combination of all of the subset of the aforementioned techniques is also a possibility to focus user attention on the overlaid UI.

Lightbox mitigates the attacks presented above. The above-mentioned work [147] shows that the Lightbox and freezing are effective in 98% and 97% of the time (baseline: 69% effectiveness when no protection is provided), respectively, making them suitable candidates for ProtectION. For more details of the user study, refer to Table 2 in [147]. We assume that a similar result should be expected in ProtectION due to the similarity of the application space (web applications). IOHub uses Lightbox as the default technique, but depending on the specific form, the developers can select the appropriate technique.

When the focusing mechanism (e.g., LightBox) is active, the user can still interact with the UI elements, and browser button outside the secure UI elements as the IOHub does not control those. The browser back button does not influence the state of the overlaid UI as long as it does not remove/change the QR code on the web page.

AUTOMATED ACTIVATION. The technique to focus user attention (dimming out or freezing the non-overlaid part of the screen) is triggered automatically in specific situations: The user moves the mouse pointer over the overlaid UI, or the user starts typing into a sensitive UI element. The advantage of the automated trigger is that the user does not need to remember to activate the mechanism. Hence the system is resilient from user habituation and does not require the user to monitor security indicators actively or perform specific actions. Note that the automated activation provides security to user IO data *only* when the integrity of the data is considered.

5.4.3 *Continuous Tracking of Mouse Pointer in the HDMI Frame*

The triggering of the focusing mechanism poses a challenging task to ProtectIOn because the IOHub does not know the exact position of the mouse pointer. We cannot rely on the compromised host to communicate the pointer position reliably to IOHub. Furthermore, the host's pointer is not visible when the user interacts with the overlay rendered by the IOHub as the IOHub always draws on top of the HDMI frames of the host.

IOHub could employ image analysis over the frame received from the host to learn the pointer position. However, we avoid this method because image analysis is time-consuming and vulnerable to adversarial images. In our approach, the IOHub intercepts mouse events and HDMI frames, so it can track the pointer based on mouse data and correlate it with the actual position in the HDMI frame (using shape detection in a small area). Then, the IOHub overlays a mouse pointer that is prominent and easy to follow by the user.

A malicious host can still show a fake pointer to trick the user into following it, but when the focusing mechanism is active (the user interacting with sensitive elements), only the pointer overlaid by IOHub is visible. This way, the pointer tracking and the pointer overlay address three major challenges: i) both the IOHub and the user have the same sense of the pointer position, ii) IOHub precisely knows when to trigger the focusing mechanism, and iii) the user can interact with the overlaid UI seamlessly.

CALIBRATION. When the user connects the IOHub for the first time after booting up, the IOHub performs an automated calibration to find the pointer. The IOHub simulates the mouse and pushes the pointer to the top-right corner of the screen. Then the IOHub searches the pointer at this position in the HDMI frames

POINTER DETECTION. The IOHub ensures pointer integrity by tracking the mouse movements using the raw data from the mouse and the HDMI frame. Figure 5.6 illustrates the main idea:

① Shows raw mouse data that notify the displacement events $(\Delta x, \Delta y)$ over x and y axis which are fired over time series $t_1, \ldots, t_n$. Note that the initial pointer position is known to the IOHub from calibration phase where $(x_0, y_0) = (0, 0)$.

② Shows the HDMI frames $f_1, \ldots f_n$ where the IOHub expects the mouse pointer to be found. For efficiency, the IOHub only scans a

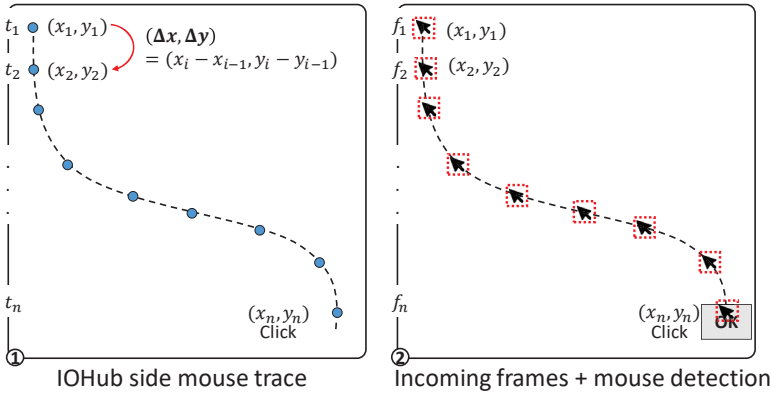


FIGURE 5.6: **Pointer tracking.** ① The IOHUB captures the raw mouse events $(\Delta x, \Delta y)$ from the mouse that is attached to the IOHUB. ② The IOHUB captures the frames from the HDMI channel and checks into the designated pixel position $(x_i + \Delta x, y_i + \Delta y)$ if there exists a pointer. $t_1, t_2, \ldots t_n$ are the time instances when the IOHUB receives the mouse data. $f_1, f_2, \ldots f_n$ are the corresponding HDMI frames that the IOHUB intercepts.

small portion of the HDMI frames (200 × 200 square pixels) that is enough to cover a mouse pointer. Since the operating system can treat mouse movements slightly differently according to their algorithm, this step serves to adjust the position difference.

OVERLAY OF THE MOUSE POINTER. The IOHUB draws a mouse pointer overlay on top of the actual mouse pointer. The host mouse pointer is neither visible on top of the overlay nor can it interact with the IOHUB's overlay. The overlaid mouse pointer is visible on top of the overlay, and it offers the same user experience as the host-rendered mouse pointer.

COPING WITH DISAPPEARING POINTER. Many OS offer a feature where the mouse pointer disappears from the screen when the user types in a text editor/browser. When the user moves her mouse, the cursor appears again in the same position where it disappeared in the first place. From the IOHUB's perspective, it is hard to distinguish between this case and the attacker deliberately removing the mouse pointer from the screen. To handle this case, the IOHUB listens to all the keyboard inputs – the keyboard is also connected to the IOHUB. Therefore, when the IOHUB gets a keystroke event,

it expects the cursor to disappear from the screen. Then, IOHub continues tracking the pointer from the moment that the mouse sends events - this way, the IOHub ensures the consistency of the pointer position.

HANDLING DIFFERENT MOUSE CURSORS. The IOHub is preloaded with template images of the mouse pointer for detection. For our ProtectIOn prototype implementation, we use the default cursors provided by the Ubuntu OS. This allows the IOHub to identify the cursor when it changes on the screen, e.g., from pointer to a hand when the user hovers the pointer over a link on the browser.

HANDLING MOUSE ACCELERATION. The IOHub uses the default mouse acceleration parameters of *libinput* to cope with the pointer acceleration. As the IOHub emulates itself as a keyboard, at the time of initialization, the IOHub sends a command to the host to set the default acceleration. In case the host changes the mouse acceleration, the IOHub will fail to detect the mouse in the HDMI stream. We consider this case as a denial of service.

ENTERING/EXITING SECURE MODE WITH KEYBOARD. In our implementation of ProtectIOn, entering and exiting the secure mode is performed by moving the mouse pointer into or out of the protected UI area (sensitive web form). This could similarly be done with the keyboard. The user could use the *TAB* button that selects the QR code (that is an image on the webpage) on the browser. When the QR code is selected, ProtectIOn JavaScript triggers a change in the JSON specification that is encoded in the QR code. E.g., the specification contains a parameter named *selected* that defines if the user is inside the secure mode or not. By changing this parameter, the ProtectIOn JavaScript signals the IOHub that the user is currently in the secure mode. While inside the secure mode, the IOHub handles the `TAB` signal from the keyboard, allowing the user to switch UI elements within the secure UI. When the user reaches the end of the secure UI, the device returns control back to the ProtectIOn JavaScript when the user presses *TAB*.

5.4.4 *Protected User Interaction*

When the user finishes providing her input via input devices (mouse and keyboard), the IOHub sends these values (with signature to ensure

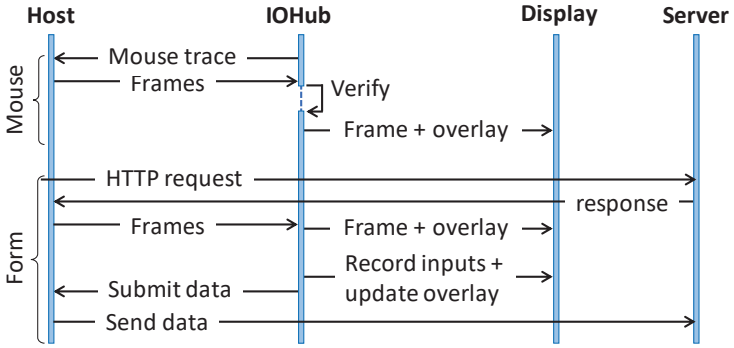


FIGURE 5.7: **Flow of the ProtectIOn main protocol.** The figure shows the sequence of events for two example scenarios: mouse movement and filling up a web-form.

integrity) to the remote server. Sending these signed input values to the server requires an upstream channel from the IOHUB to the server.

UPSTREAM CHANNEL. The data from the IOHUB to the remote server is transmitted using the PROTECTION JavaScript snippet as a helper. The IOHUB emulates itself as a composite human interface device (HID) when it is connected to the host. The IOHUB emulates keystrokes that transmit encoded data to the PROTECTION JavaScript snippet, which then forwards them to the remote server.

SENDING INPUT DATA. Figure 5.7 depicts the user interactions in a sequence diagram. The user input transmission procedure is illustrated in Figure 5.5. This has two phases: *record* and *transmit* as described in the following:

1. **Record.** After the UI elements are correctly overlaid on the screen; the users can interact with these UI elements. The user interaction with the overlaid UI element is no different than a standard UI. The UI specification encodes the behavior of all generated UI elements, making the IOHUB aware of the semantics of the UI objects. E.g., when a user selects a text box and types on with her keyboard, the IOHUB intercepts all keystrokes and renders the characters on the overlay. When the user enters input data in the rendered overlay UI elements (such as textbox, button, slider, radio button, etc.), the IOHUB records that in a (key, value) pair where the key is the identifier of the UI

element (*id* in Specification 5.1) and the value is the user-provided value. The *type* of the UI elements determines what information to record. For example, the IOHub records all keystrokes when a textbox is selected, the value corresponding to the position of the slider is recorded when the user interacts with a slider, etc. One example of the recording of the input data corresponding to the UI illustrated in Figure 5.5 and Specification 5.1 is:

$$Record = (tb\_1, Data\_1); (tb\_2, Data\_2)$$

2. **Transmit.** In the transmit phase, the IOHub waits for the user to select a UI element that has a *trigger* capability, e.g., a submit button on a web form. A trigger element can change the state of the overlaid form, e.g., submit the data of the form to the remote server or reset it. More details are provided in the implementation of ProtectIOn in Section 5.7.2.1. When the user clicks the *OK* button, the device signs *Record* with its embedded private key. One such signed packet is also illustrated in Figure 5.5. The IOHub sends the signed packet to the remote server using the upstream channel.

Upon receiving the signed input data from the IOHub, the remote accepts the input if the signature verification is successful. Note, if an input field is annotated as `protect="true"`, the server does not accept any input without the IOHub signature. This prevents the attacker-controlled host from submitting data.

CHANGING BROWSER TABS OR BROWSERS. The IOHub supports multiple browsing tabs across multiple browsers. The UI specification contains *formId* and *domain* that works as the unique identifier for a specific form served from a specific web server. The IOHub can maintain multiple parallel TLS connections to web servers. Depending on the observed *formId* and *domain* (refer to Specification 5.1), the device retrieves the data that is entered by the user. This way, even if the user switches tabs, the IOHub can still allow editing the forms across tabs.

INPUT VALIDATION. Input validation, i.e., checking the input against a recommended input policy (e.g., regular expression) is one of the most widely used JavaScript functionalities, and it is a critical part of input integrity. The remote server sends the regular expression in the UI specification (*RE* in Specification 5.1) that the IOHub uses to validate the user input.

```
<form action="/some_action">
  Text box 1:<br>
  <input type="text" name="text_box_1">
  <br> text box 2:<br>
  <input type="text" name="text_box_2">
  <encrypted_qr><!-encrypted UI specification->
  0x4a5c4... </encrypted_qr>
  <script> [JS outputs QR code that encodes
  encrypted specification] </script>
</form>
```

SPECIFICATION 5.2: **Example HTML form showing encrypted UI specification.** HTML page from the remote server that contains the encrypted UI specification for IO confidentiality.

FALLBACK FOR LEGACY CLIENTS. PROTECTION is backward-compatible with the clients who do not use the IOHUB. This is achieved by the remote server by showing a QR code briefly on the screen when the user visits the PROTECTION-enabled webpage. The IOHUB intercepts the QR code and sends a signal to the server about its presence. In the absence of the IOHUB, the remote server does not send the PROTECTION JS to the host that acts as a communication channel between the IOHUB and the remote server. Note that the fallback mechanism is application-specific, and the service provider could decide if the fallback is detrimental to security.

5.5 PROTECTION FOR IO CONFIDENTIALITY

In the previous sections, we describe how the PROTECTION JavaScript and the IOHUB together ensure the integrity of the IO. We now augment the design of PROTECTION to achieve IO confidentiality alongside the IO integrity. One of the major components for achieving IO confidentiality is to establish a secure channel (i.e., a TLS channel) between the remote server and the IOHUB. TLS ensures that the untrusted host does not read or modify any data exchanged between the user and the remote server.

5.5.1 *IO Operations*

ESTABLISHING TLS. The IOHUB and the server create TLS using the public certificates. The TLS uses the emulated keystroke streams and HDMI

as the upstream and downstream channels, respectively, as described in Section 5.4. Implementation details are provided in Section 5.7.2.8.

OUTPUT CONFIDENTIALITY. Output confidentiality ensures that information sent from the remote server and the visual render of the user's input is hidden from the host. To enable output confidentiality, the UI overlay mechanism that is described in Section 5.4.1 is modified slightly. The difference is that the specification is not generated on the host side but rather on the server. A small server-side module that is very similar to PROTECTION JS transforms the UI elements to the UI specification (one example is provided in Specification 5.1) and encrypts it with the TLS session key. The encrypted specification is delivered to the client browser inside the `<encrypted_qr>` tag in the HTML file, which is then encoded (as a QR-code) by the PROTECTION JS. The IOHUB decodes the QR code from the intercepted HDMI frames, decrypts the specification, and renders the overlay accordingly. One example is provided in the HTML Snippet 5.2 with the corresponding UI illustrated in Figure 5.8. This feature of PROTECTION allows the remote server to send securely private information to the user in the presence of a compromised host, e.g., bank account statements or any other confidential message.

INPUT CONFIDENTIALITY. When the user enters her mouse pointer into the overlaid UI area, the IOHUB stops transmitting any mouse or keyboard event to the host, making it completely oblivious of any mouse movement or keystroke during that time. However, the user can still see her inputs on the screen as the IOHUB renders the plaintext character on the overlaid UI elements, therefore making them visible only to the user. Likewise, when the user selects a UI element, for example, a radio button that is shown in Figure 5.8, the IOHUB stores the selected value in the recorded data. On form submission, IOHUB encrypts the recorded data with the TLS key and sends them to the remote server.

5.5.2 *Focusing User Attention*

The IO confidentiality could be viewed as a similar problem to *phishing* where the user provides the inputs to an attacker-generated UI (or a phishing webpage) that leaks the sensitive information. Similar to the phishing protection mechanisms, IO confidentiality requires additional attention/-operations from the user. Secure Attention Sequence (SAS) is a sequence

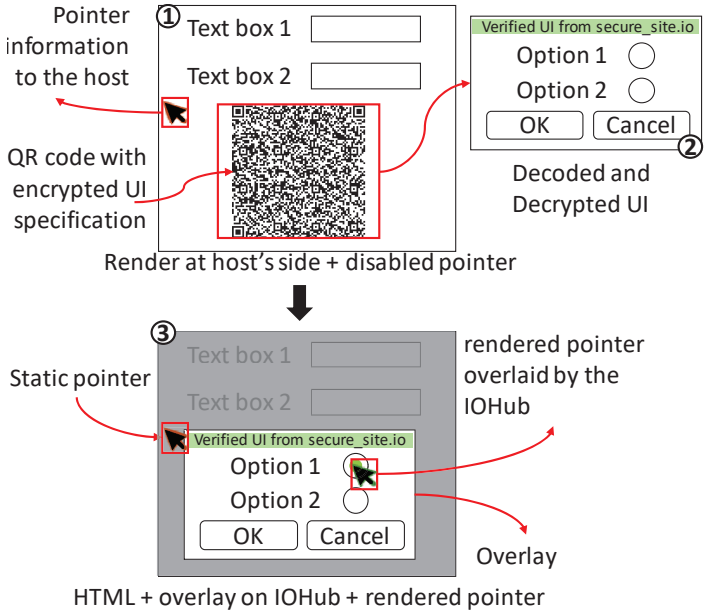


FIGURE 5.8: **ProtectIOn IO confidentiality.** The figure shows ① the browser render of the webpage in Specification 5.2 where the PROTECTION JavaScript produces the encrypted QR code. ② shows the UI overlay that is decrypted and decoded by the IOHUB. ③ shows the user's view when the IOHUB overlays the UI on the HDMI frame, and the user starts to interact with the UI.

of trustworthy actions (such as keystrokes `Ctrl+Alt+Del` in Windows) executed by the user. SAS prevents an untrusted system from triggering an event that is otherwise sensitive to the user. Note that SAS is a well-researched topic in the context of UI/UX design. PROTECTION adapts an off-the-shelf SAS mechanism that provides a visual aid for the user to distinguish overlaid UI and the mouse pointer location. SAS is crucial for IO confidentiality as the untrusted host can trick the user into inputting her sensitive information on a forged form. Hence, the user needs to remember the SAS to distinguish IOHUB generated UIs from host generated UIs. Note that the automated activation is insufficient as, at any given time, the host can maliciously emulate the automated activation to trick the user into providing the sensitive information to an illegitimate UI.

Note that SAS is one of many ways to inform the user securely about the trusted overlay on the screen generated by the IOHub. Evaluation of the effectiveness of SAS over other attention focusing mechanisms is out of scope of this paper. Hence, ProtectIOn uses SAS as an example of the attention focusing mechanism for confidentiality. In principle, ProtectIOn could be integrated with other proposed approaches such as security indicators or secret images [148, 149].

5.5.3 *UI Protection Profile*

The remote server can set a configurable UI protection profile per overlaid protected UI (i.e., QR code). The protection policy is defined in the `SAS` attribute in the example specification provided in Specification 5.1. The policy dictates how the UI would respond to the SAS provided by the user. By default, the overlaid UI is locked from the user and requires the SAS keystroke from the user to unlock the sensitive UI. This information is overlaid on the UI to remind the user to execute it. One example UI protection policy could be `5:LB` (refer to Specification 5.1), which denotes IOHub invokes a lightbox on the HDMI frames except for the UI overlay and the mouse pointer overlay for a cool-down period of 5 seconds. The form remains locked for this cool-down period.

The design of the ProtectIOn system is independent of the secure attention sequence (SAS) value. In principle, each issuer that deploys IOHub to users (see Section 5.3.1) could define its own custom SAS and configure the deployed IOHub to intercept that key sequence. Our recommendation, however, is that the IOHub issuers follow established platform-specific SAS values. For example, if a IOHub is issued for the purposes of protecting user interactions on a Windows platform, we recommend that the device issuer pre-configures the IOHub to intercept the Windows-specific `Ctr+Alt+Del` sequence. Similarly, if a IOHub is deployed to be used on another OS, it should be pre-configured to intercept the SAS sequence commonly used on that platform. (In cases where the same IOHub would be used on multiple different platforms, it could be either configured to intercept multiple SAS values or it could use a single SAS value that the issuer needs to communicate to its user.)

5.6 SECURITY ANALYSIS

In this section, we will informally analyze the security of ProtectIOn. We divide the security analysis into two parts based on two security properties: interity and confidentiality.

5.6.1 *Integrity*

MODIFYING IO OPERATIONS. As only the IOHub can interact with the overlaid UI, the attacker can not manipulate the IO operations with the overlaid UI. Moreover, the attacker cannot submit arbitrary data to the remote server because the latter accepts only inputs signed by the IOHub.

EARLY FORM SUBMISSION. This attack is not possible as the input devices (both mouse and keyboard) are connected to the IOHub, and only the IOHub can interact with the overlaid UI. This makes it impossible for the attacker to emulate a click on the overlaid part of the screen.

ATTACK ON THE MOUSE POINTER TRACKING AND OVERLAY. The attacker may try to defeat the ProtectIOn pointer tracking and overlay mechanism described in Section 5.4.3 by introducing a malicious pointer that is visually more appealing to the user. Note that the IOHub overlaid mouse pointer is prominent and hard to miss. One can visualize it as an arms race between the attacker and the IOHub to grab the user's attention. We argue that this is a suboptimal strategy for the attacker as both of the pointers will be visible on the screen that causes suspicion to the user. Also, when the real mouse pointer enters the overlaid area, the untrusted part, including the malicious mouse pointer, will be hidden by the focusing mechanism. Hence, we can conclude that executing clickjacking-like attacks is not possible in ProtectIOn.

REPLAY ATTACK. The remote server adds a random identifier (`id`) in the form specification alongside the signature. With this identifier, the server keeps track of the user input. When the server receives a form submission data, it first checks if the user submitted with the same identifier sent by the server. Otherwise, the server rejects the data.

NOT RENDERING QR CODE. The host may deny sending the QR code over the HDMI channel. We consider this to be a denial of service and does not compromise the integrity of the IO data.

REDIRECTION. The attacker could redirect the user to a phishing website that renders visually identical UI to that of the legitimate website. A redirection attack cannot break the integrity of the input because a legitimate remote server always requires the signed input from the user. Without a valid signed specification, the IOHUB never renders an overlay or sign any input.

MALICIOUS INSTRUCTION ON THE SCREEN. The attacker may put a malicious instruction/label on the untrusted part of the screen to influence user inputs. However, when the user starts interacting with the overlaid UI, the default focusing mechanism (Lightbox) highlights only the secure UI and hides the rest of the screen.

REPLICATION OF LIGHTBOX. The attacker can replicate the lightbox on any part of the screen. However, this does not compromise the integrity of the input as the legitimate remote server only accepts signed input from the IOHUB.

MULTIPLE HIDS. The attacker can emulate multiple HIDs to avoid the tracking of the mouse pointer. However, this attack is ineffective as the IOHUB only tracks the mouse pointer that is connected to it (over USB interface).

BADUSB. BadUSB [150] is out-of-scope of this paper as in the attacker model (Section 5.3.1), we assume that all the IO devices that are connected to the IOHUB are trusted.

MOUSE ACCELERATION/UPDATES. The attacker can change the mouse acceleration or provide erratic mouse updates on the screen. Such manipulations only cause the IOHUB to lose track of the mouse pointer and stop relaying the mouse signal to the host altogether. The IOHUB uses the acceleration parameters from the default `libUSB` driver to cope with the mouse acceleration. Hence, such manipulation does not affect security.

MALICIOUS QR CODES. The attacker may put fake QR codes on the webpage. Note that the IOHUB verifies the signature from the HTML forms to check the integrity using the pre-configured or white-listed server certificate. This way, the IOHUB does not render any overlays from malicious QR codes.

5.6.2 *Confidentiality*

REDIRECTION. The attacker could redirect the user to a phishing website that renders visually identical UI to that of the legitimate website. Redirection compromises the confidentiality of user inputs only when the user does not trigger the SAS mechanism. The IOHUB is only activated when it detects specifications signed from the whitelisted (maintained in the memory) servers.

FAKE SAS INSTRUCTIONS. The attacker may put fake instructions on the screen that attempt to trick the user into typing a false SAS sequence and then revealing her sensitive information to the attacker. This attack is not possible as long as the user follows the instructions it received from the issuer of the IOHUB and only types in secrets after using the correct SAS value (such as `Ctr+Alt+Del`). Recall from Section 5.5.2 that the SAS value is defined by the issuer of the IOHUB and that the SAS keystrokes are always first intercepted by the IOHUB. (The user is expected to trigger the SAS only when there exists a QR code on the screen that is correctly signed by the remote server. In case there is no QR code or a malformed QR code on the screen, the IOHUB warns the user.)

SIDE-CHANNEL LEAKAGES. Even though, the IOHUB ensures that no mouse or keyboard event arrives at the untrusted host when the user executes some operation over the overlaid UI, one can not rule out all side-channel leakages. Depending on the application, the amount of time that the user spends or the entry/exit position of the mouse pointer may reveal some information to the attacker. IOHUB could allow the remote server to specify additional policies in the specification to prevent such side-channel attacks, e.g., a minimum amount of time that the device should not forward any event to the host after the user enters the overlay. We leave as future work defining such policies and integrating them on PROTECTION.

MODE SWITCHING. The host could remove the QR code when the user is typing confidential data in the sensitive form. The absence of the QR code makes the IOHUB to assume that the secure session has ended, and the IOHUB forwards the plaintext keystrokes and mouse movement to the host. To prevent the leakage of the input data, the IOHUB continues to overlay and operate on the overlay until the user clicks submit or cancel (or any UI element that has a `trigger` capability). This way, the IOHUB locks the UI from the attacker until the user finishes her session.

5.6.3 *Attacks toward* IOHUB

In PROTECTION trust model, we assume that the IOHUB is trusted. However, in the real-world, embedded systems are often vulnerable to attacks as the attacker can use the connection interfaces to reprogram the IOHUB. It is also possible to develop the IOHUB using formally verified languages such as embedded Rust. However, we consider making a security-hardened IOHUB is engineering intensive and out-of-scope of this paper.

DOWNGRADE ATTACK. The host can block the initial QR code from the server to the IOHUB. By doing so, the host forces the server to downgrade the security of the webpage, i.e., not serving the PROTECTION JS. For integrity, this is not a security threat as the server does not accept any input from the host that is not signed by the IOHUB. Hence, the downgrade attack works as a denial of service, which is out-of-scope of this paper.

5.6.4 *Proof for IO Integrity*

In this section, first, we provide a formal proof of the following property: *without protecting both input and output integrity, none of them can be achieved*.

INTERACTION PROTOCOL. To simplify the proof, we model the interaction between the user, the host, and the remote server as a finite state automaton (FSA). The interactions between the server ($\mathcal{S}$), the user ($\mathcal{U}$) and host ($\mathcal{H}$) are depicted in the FSA in Figure 5.9.

We consider a setting where the user $\mathcal{U}$ interacts with $\mathcal{S}$ over browser through web UI. Hence we assume that all the messages (m or m') exchanged between the user $\mathcal{U}$, host $\mathcal{H}$ and server $\mathcal{S}$ is HTTP request and response payload. The HTTP response payloads (originating from $\mathcal{S}$) contains HTML, JavaScript, CSS and other data to construct the webpage at

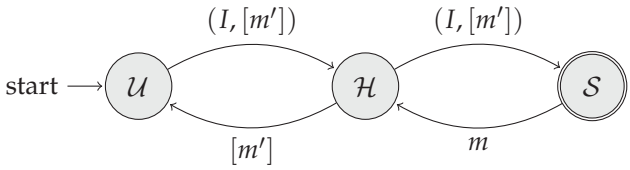


FIGURE 5.9: **ProtectIOn protocol FSM.** The finite state machine (FSM) that depicts the interaction between the user ($\mathcal{U}$), host ($\mathcal{H}$) and the server ($\mathcal{S}$). m stands for the messages sent from $\mathcal{S}$ that is usually HTML, CSS, JS, etc that is rendered by $\mathcal{H}$. The rendered webpage is denoted by $[m]$ that is shown to the user. I stands for the input provided by the user on $\mathcal{H}$ after seeing $[m]$.

the host's browser. The protocol starts with $\mathcal{U}$ sending an initial request to $\mathcal{S}$ that is delivered through $\mathcal{H}$. We denote $[m]$ to be the render of m by the $\mathcal{H}$, i.e, graphical render of the webpage (m) on the host's display. In this initial stage consider $[m'] = \phi$. Upon receiving the initial request, the server $\mathcal{S}$ replies with a message m to $\mathcal{H}$. As $\mathcal{H}$ is malicious, it can transform m to m'. Note that this transformation from the message to render to the user's display is a public knowledge and is deterministic. Hence, for a message m', where $m \neq m'$, then given the corresponding renders $[m']$ and $[m]$, $\mathcal{S}$ can determine that $[m] \neq [m']$. We denote the user input to be I, which corresponds to a specific $[m]$. In this model, we simplify the user input by assuming that the $\mathcal{U}$ only provides an input I only after observing a message transformation $[m]$. The user provides both her input I and transformation $[m']$ observed by her to $\mathcal{H}$. The interaction loop between $\mathcal{H}$ and $\mathcal{U}$ can continue until $\mathcal{U}$ finishes her input. After every input $\mathcal{H}$ hands over new message transformation to $\mathcal{U}$ (either result of the input or new message from $\mathcal{S}$ or both). This simulates the changes in the web UI when the user starts interact with it e.g., input feedback. Once the user provides all her inputs, $\mathcal{H}$ send the sequence of pairs $(I, [m'])$ to $\mathcal{S}$.

To smmarize the interaction protocol above, we define these two mappings as the following:

$$\texttt{Input()} : [m] \rightarrow I$$
$$\texttt{Transform()} : m, I \rightarrow [m'],\ \exists i \in I : i = \phi$$

Both of them are *bijective*.

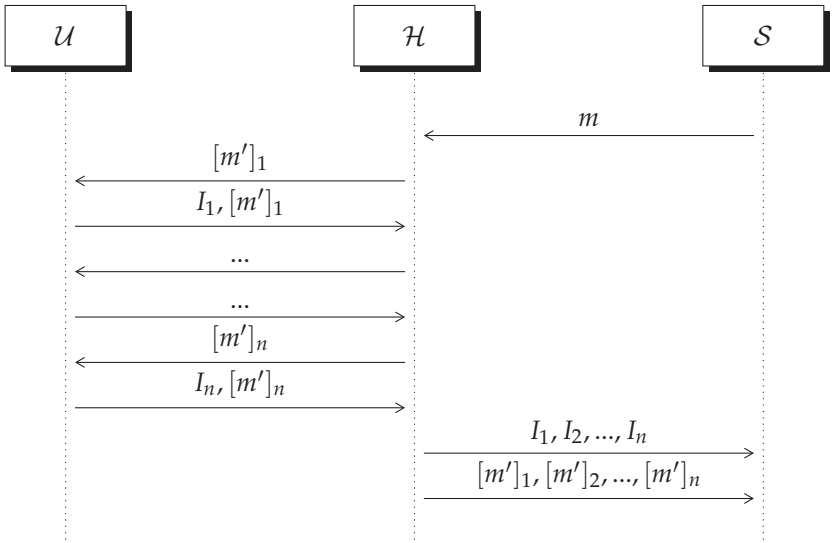


FIGURE 5.10: Protocol transcript between the $\mathcal{S}$, $\mathcal{U}$ and $\mathcal{H}$ that shows one trace from the FSM depicted in Figure 5.9.

One trace of the protocol transcript mentioned above is depicted in Figure 5.10. As described in the FSM (refer to Figure 5.9), $\mathcal{S}$ receives a trace of message transformations $([m']_1, [m']_2, \ldots, [m']_n)$ and corresponding inputs $(I_1, I_2, \ldots, I_n)$. From these traces $\mathcal{S}$ could determine of all the $[m']_i$ are in proper form by verifying if $[m]_i = [m']_i$.

Given the interaction protocol, we can now formally define the definitions in this chapters such as the input intergity and output intergerity as the following:

Definition 5.6.1. Input integrity Assume that $\mathcal{S}$ handed a message m to $\mathcal{H}$ where the proper message transformation is $[m]$. The host changes the message transformation to $[m']$ where $[m'] \neq [m]$. We also define correct $\mathcal{U}$ input to be I when $\mathcal{H}$ sends a correct message transformation $[m]$ to $\mathcal{U}$. We define input integrity as the property where the $\mathcal{S}$ does not accept input I' where $I' \neq I$from $\mathcal{U}$ if the $\mathcal{H}$ changes the message transformation.

Definition 5.6.2. Output integrity Assume that $\mathcal{S}$ handed a message m to $\mathcal{H}$ where the proper message transformation is $[m]$. Output integrity defines that in all circumstances, $\mathcal{U}$ receives the correct message transformation $[m]$ from $\mathcal{H}$.

VERIFICATION PROCESS. After receiving all the traces, $\mathcal{S}$ checks $\forall i = 1 \ldots n$, if

$$[m']_i =^? \texttt{Transform}(m_{i-1}, I_{i-1})$$

where $I_0 = \phi$.

Lemma 5.6.1. If $\mathcal{U}$ does not send all the transformations till $[m']_i$ corresponding to the input I_i, input integrity can not be achieved.

Proof. If $\mathcal{U}$ does not attach all the transformation till $[m']_i$, i.e.,

$$[m']_1, [m']_2, \ldots, [m']_{x+1}, [m']_x, [m']_{x-1}, \ldots, [m']_{i-1}, [m']_i$$

corresponding to inputs $I_1, I_2, \ldots, I_{x-1}, I_x, I_{x-1}, \ldots, I_{i-1}, I_i$, then the server can not verify all the transformations corresponding to the input. $\mathcal{H}$ could modify a specific $[m]_x$ to influence $\mathcal{U}$ input. Hence, the following verification will be missing:

$$[m']_x =^? \texttt{Transform}(m'_{x-1}, I'_{x-1})$$

Where the $\mathcal{H}$ changes message m to m' influence the the user to change her input from I_{x-1} to I_{x-1}. Hence input integrity can not be achieved. □

Lemma 5.6.2. If the channel from $\mathcal{U}$ and $\mathcal{S}$ is not authenticated, input integrity is not achievable. But the channel from $\mathcal{S}$ to $\mathcal{U}$ does not require to be secure as long a $\mathcal{U}$ provides the message transformation $[m']_i$ corresponding to every input I_i.

Proof. The proof is trivial. If the channel from $\mathcal{U}$ to $\mathcal{S}$ is not authenticated, any input provided by $\mathcal{U}$ can be manipulated by $\mathcal{H}$ without a trace. Hence input integrity is not achievable. As long as $\mathcal{U}$ sends message transformation along with the input, a manipulated message transformation bt $\mathcal{H}$ would be detectable by $\mathcal{S}$ (see Lemma 5.6.1). □

Lemma 5.6.3. Ensuring output integrity also ensures input integrity provided there is an authenticated channel from $\mathcal{U}$ to $\mathcal{S}$.

Proof. This proof is also trivial. As we describe in the Definition 5.6.1 and 5.6.2, if all the message transform from $\mathcal{H}$ $[m'] = [m]$, and $\mathcal{H}$ always executes `Transform()` properly, the input integrity is preserved. As ProtectIOn ensures output integrity and all the input from the user is signed by the IOHub, ProtectIOn preserves input integrity. □

Similarly, we can also prove the following property: *If not all the modalities of inputs are secured simultaneously, none of them can be fully secured.*

This is a general case for the proof that is described previously. We can modify the `Input` and `Transform` function to handle multiple modalities of input. We assume there are T input devices providing T-modalities of input (denoted by $I^1, \ldots, I^T$).

$$\texttt{Input()} : [m] \rightarrow I^1, I^2, \ldots, I^T$$
$$\texttt{Transform()} : m, I^1, \ldots, I^T \rightarrow [m'],\ \exists i^t \in I^T, \forall t \in T : i = \phi$$

Hence the verification process at the server side will be changed as the following:

VERIFICATION PROCESS. After receiving all the traces (of input and message renders), $\mathcal{S}$ checks $\forall i = 1 \ldots n$, if

$$[m']_i =^? \texttt{Transform}(m_{i-1}, I^1_{i-1}, \ldots, I^T_{i-1}), \text{where } I^t_0 = \phi, \forall t \in \{1, \ldots, T\}$$

If any of the input modality is missing from the trace, the server cannot verify the input integrity.

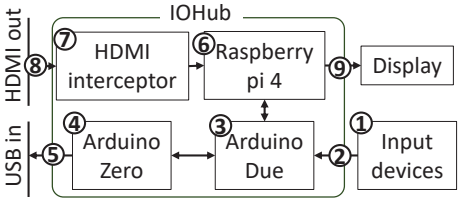
5.7 PROTECTION PROTOTYPE

In this section we provide an overview of our PROTECTION prototype implementation.

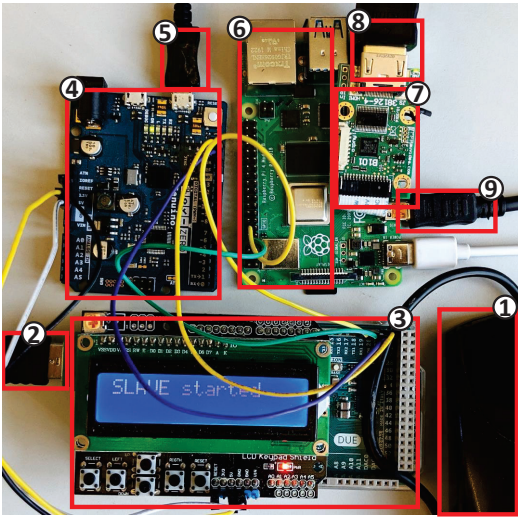
5.7.1 *Setup*

Here, we describe our prototype implementation of PROTECTION as an auxiliary device. Figure 5.11 depicts the PROTECTION prototype in two parts: Figure 5.11a shows the block diagram of our prototype with various components and connections, and Figure 5.11b shows a photo of the actual prototype that highlights all the components described in the block diagram. The prototype IOHUB is connected to a desktop computer with 3.40 GHz Intel Core i7-6700 processor with 8 GB RAM running Ubuntu 18.04.2 LTS. The IOHUB uses off-the-shelf devices and has the following components (we use the same numbering as shown in Figure 5.11a and Figure 5.11b):

1. *Computing component.* We use a Raspberry Pi 4 (⑥) to implement the computing component that executes all the IOHUB logic that includes



(a) The figure shows the basic components and connections between them in our ProtectIOn IOHub prototype. `USB in` and `HDMI out` are the connection interfaced between the IOHub and the host platform.



(b) ProtectIOn IOHub prototype uses Arduino Due and Zero microcontroller board and a Raspberry Pi 4 SBC. The highlighted numbers correspond to the labels in Figure 5.11a.

FIGURE 5.11: **ProtectIOn prototype**. Figure 5.11a and 5.11b shows the schematic and a photo of the ProtectIOn IOHub prototype respectively.

analyzing the HDMI frames, rendering the overlays, executing the TLS protocol, etc. One could use an ASIC to further improve the performance and reduce the code base of the component. The Pi is connected to the display over HDMI (⑨) interface. The code base of the Pi primarily consists of Python and Java. Note that one could implement all the HDMI processing in a FPGA () to further reduce the TCB.

2. *Input interceptor.* The input interceptor is composed of an Arduino Due (③) and an Arduino Zero (④) that is connected to the input device over USB (②) interface. The input interceptor has a USB out interface that connects to the host (⑤) that relays all the user inputs to the host.

3. *HDMI interceptor.* The HDMI interceptor (⑦) is implemented using a B101 HDMI to CSI-2 Bridge [151] that takes the HDMI channel (⑧) from the host and convert it to the camera input signal to the Raspberry Pi 4.

5.7.2 *Implementation of* ProtectIOn *Components*

In the following, we provide the implementation details of the ProtectIOn components presented in the previous sections.

5.7.2.1 *QR code generation & UI specification*

QR code generation phase is executed by ProtectIOn JS that transforms the UI elements of a sensitive web form to a UI specification encoded in a QR code (we use QRCode.js, a JavaScript library to produce QR codes). Section 5.4.1 provides the high-level concept of generating the QR code from the webpage UI elements. UI elements that require IO integrity protection can be marked by the developers in the HTML source. As illustrated in Figure 5.5, the HTML UI elements: '`Sensitive field 1`' and '`Sensitive field 2`' have the additional attribute `protect="true"`.

The ProtectIOn JS iterates through the HTML elements that have the `protect` attribute enabled and extracts the information such as the name of the label or the type of the UI element. IOHub uses preloaded size parameters to specify the size of a text field, button, etc. in case the size is not explicitly mentioned in the HTML source. One important attribute for a UI element in the specification is the `trigger`. For example, in Specification 5.1,

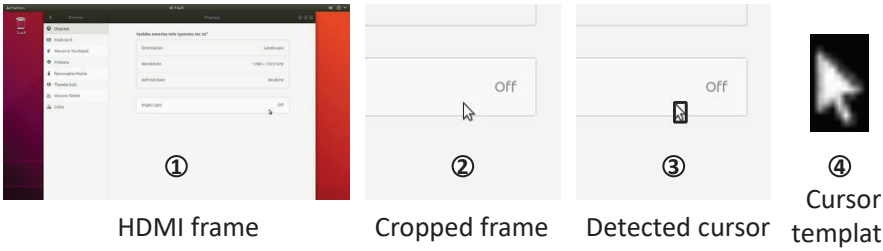


FIGURE 5.12: **Cursor detection on the HDMI frame.** The figure shows PROTECTION mouse pointer tracking. ① shows the captured HDMI frame captured by B101 HDMI to CSI bridge. ② shows the cropped HDMI frame based on the mouse position received by the IOHUB. ③ shows the detected mouse pointer. For testing, we program the IOHUB to put a rectangle around the pointer. ④ shows one of the pointer templates that we used in our OpenCV routine.

the `OK` and the `cancel` buttons have an attribute `trigger`. This attribute is Boolean can be either `true` (corresponding to `OK`) or `false` (corresponding to `Cancel`) value. The value `true` denotes that the `OK` button can submit the values that are provided by the user. The `false` attribute denotes that hitting the `cancel` button abort the form altogether.

The QR code generation phase is between ① and ② in Figure 5.5 where the PROTECTION JavaScript snippet transforms the UI elements to a UI specification language in a QR code that can be interpreted by the IOHUB. The UI specification corresponding to the HTML source (in Figure 5.5) is provided in Specification 5.1. Note that the specification is highly flexible, allowing adjustable size for the form, individual UI elements, gaps between them, etc. This allows the IOHUB to faithfully recreate the UI that is very close to the actual form UI that the served by the web severer.

5.7.2.2 *Bitmap generation*

The IOHUB reads the QR code from the HDMI frame and generate the UI overlay bitmap from it. We have used the `piCamera` library to intercept the HDMI frames and generate the UI on top of it. Our PROTECTION prototype implements the most frequently used HTML input elements [152] that are common in sensitive forms using Java SWT graphic library.

5.7.2.3 *Detection of mouse pointer*

Initially, when the system boots up the IOHub perform the calibration phase (see Section 5.4.3) to synchronize its coordinates of the pointer with the host. The detection of the mouse pointer is implanted partially on the raspberry pi 4 (⑥ in Figure 5.11), while the mouse intercepting is done in the Arduino Due (③ in Figure 5.11). The Due gathers the raw mouse data (in terms of displacement measurements $(\Delta x_i, \Delta y_i)$) and sends them to the Pi over `Serial` interface. To guarantee that the IOHub and the host interpret displacement events likewise, the Pi performs an adjustment operation. This operation consists of the IOHub detecting the exact position of the host pointer in the HDMI frame by analyzing a small square of the frame (200 × 200 sq. pixel) around its pointer coordinates. Considering that the IOHub gets raw HDMI frames and pointer images are static, we use the lightweight `template matching` algorithm of the OpenCV library for the detection.

5.7.2.4 *Mouse Pointer Tracking*

The pointer tracing is also executed in the aforementioned Java program using simple object detection technique supplied by the OpenCV API [153]. Figure 5.12 shows one screenshot of the pointer detection. The Figure shows the entire HDMI frame, the cropped frame of resolution 200 × 200 square pixel (based on the mouse input data), the detected pointer in the cropped frame and the cursor template that is used by the object detection algorithm.

5.7.2.5 *HID Drivers*

We use subsubsection prototype development board as the HID drivers. Figure 5.11b shows an Arduino Due, and a Zero board where the Due connects to the HIDs via the native USB port and the Zero relays the HID data to the Raspberry Pi (RPi). The Due and the Zero boards are connected over I^2C interface. As both Due and Zero only have one native USB port on each of them, we were forced to use two boards as an HID interceptor and relay. The Zero relays the HID signals both to the connected host (over native USB) and to the RPi (over serial interface). The connection from the Zero to the host is one way and emulates a composite HID. While the connection between the Zero and the RPi is bidirectional. The HID drivers are implemented using the native Arduino `keyboard` and `mouse` library. On the RPi, no HID drivers were needed as the RPi receives processed HID

data from the Zero (for the pointer: displacement over x and y-axis and for keyboard, ASCII characters).

5.7.2.6 *HDMI Interceptor, Relay and Overlay*

The RPi along with the Auvidea B101 HDMI to CSI bridge, acts as the HDMI interceptor and relay. The B101 board converts HDMI signals from the host as a camera input (via the CSI interface) to the RPi. This allows the RPi to access the HDMI frames as a stream of JPEG frames. The HDMI out of the RPi acts as the relay that connects to the monitor. On the RPi, we use Picamera API [154] to access the HDMI frames. The B101 is capable of processing 25 frames at 1080p resolution. Hence, this is the hardware bottleneck of our implementation. However, the upcoming B112 board could solve this performance issue.

On the RPi, the overlay and HDMI out is implemented using Java SWT. Using SWT, we create a full-screen window that is shown on the monitor. The SWT class polls the HDMI frames and process them as individual JPEG images via the `BufferedImage` class. This allows the overlays to be drawn on the HDMI images efficiently. The Java program uses a QR code interpreter to extract the UI specification. Based on the UI specification, it creates the geometrical shapes (corresponding to the UI elements) and draw them on the frames. In the current implementation of the ProtectION, the UI elements such as button, text-field, radio button etc. are preloaded in the IOHub memory. Note that the current implementation of IOHub is based on the RPi. But one could implement such functionality on an FPGA, reducing the TCB even more.

5.7.2.7 *Implementation of the upstream channel*

The *upstream* channel, i.e., the data from the IOHub to the remote server is transmitted using the ProtectION JavaScript snippet that is served by the remote web server. The ProtectION JavaScript snippet uses a hidden text field to accept data coming from the IOHub. The IOHub emulates itself as a composite human interface device (HID) when it is connected to the host. The IOHub emulates keystrokes that transmit encoded data (base64) to the ProtectION JavaScript snippet that is sent to the remote server via `XMLHttpRequest` call.

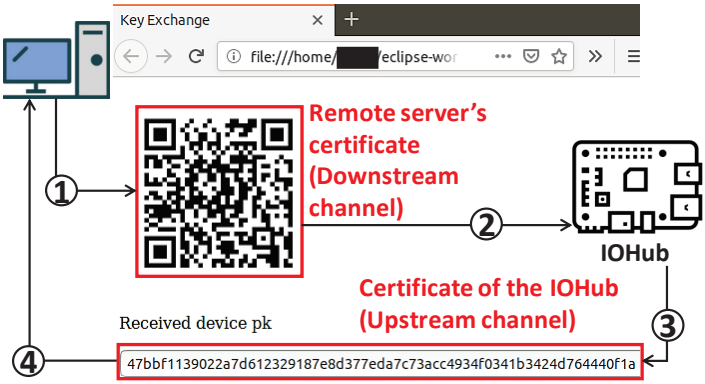


FIGURE 5.13: **Establishing TLS between IOHub and the remote server.** A snapshot of the key exchange web page that is used to communicate the public certificates of the IOHUB and the remote server.

5.7.2.8 *Establishing TLS*

For the IO confidentiality, the IOHUB and server create a TLS channel. When the user opens up a secure webpage, key exchange is the first step that takes place. We assume that the remote server already has the IOHUB's certificate, or some offline registration takes place. An instance of the key exchange protocol of PROTECTION is illustrated in Figure 5.13. The flow of the key exchange mechanism is as the following:

① The server delivers a web page with a QR code that encodes the signed public key of the server (server hello in TLS).

② The device captures every frame until it detects a QR code. Then, it decodes the QR code and verifies the public key and derives the shared secret using Diffie-Hellman protocol [155].

③ The device then sends its signed public certificate to the host, which forwards it to the server.

④ The remote server gets the signed certificate from the IOHUB, verifies it, and finally derives the shared secret.

Operation	Average time/accuracy
Detecting mouse pointer (A)	1.76 ms
Detection QR code (B)	14 ms
Decoding QR code + Overlay (C)	6 ms
Effective display latency $(A + B + C)$	21.76 ms
Mouse latency	250 μs
Keyboard latency	170 μs
Image analysis accuracy of mouse pointer	0.997

TABLE 5.1: **IOHub performance**. The table shows the latency and accuracy corresponding to PROTECTION prototype operations.

5.8 PROTOTYPE EVALUATION

We evaluate the performance of our prototype by measuring the overheads introduced by PROTECTION to the system and whether they influence the user's interaction. Initially, we measure the default latency introduced by IOHUB when the user interacts with applications that do not require protection. Table 5.1 provides the relevant latencies and the accuracy of the pointer detection. The delay in forwarding keystrokes is 170 μs, and for frames is 21.76 ms. This allows the IOHUB to achieve the maximum display frame rate of 47.69 per second (e.g., most of the movies are shot and shown in 24-30 fps). However, an optimized implementation of the technique to encode information in the HDMI frame would reduce the processing time of a frame significantly and increase further the frame rate as a result. The B101 HDMI to CSI HDMI interceptor has a hardware limit of 25 frames at 1080p resolution. We report 0.997 accuracy of the pointer detection mechanism that involves image analysis and pointer motion tracking. The accuracy is evaluated from 4196 captured frames. We observe that the misdetection happens only when the pointer is not completely visible, i.e., the pointer is on the border of the screen and the OS displays it partially. Note that one could improve the logic of IOHUB to run the adjustment phase (see Section 5.4.3) only when the pointer is within the screen completely.

Our prototype of PROTECTION does not require the user to install any additional software in her machine to facilitate the communication between the remote server and the IOHUB. Instead, the IOHUB communicates with the remote server by using the host as an untrusted transporter. Therefore,

we start by measuring the delay of sending data from the device to the host and vice versa:

1. IOHUB → *host* The IOHUB transmits data (encrypted) to the host by simulating keystrokes. In our system, IOHUB sends the keystrokes in a chunk of 256 bytes of data to the host. The keystroke has an average latency of 5 ms, which is undetectable by humans.

2. *Host* → IOHUB The host sends data to the device by encoding them into the HDMI frame. The QR-code is generated locally in the browser and displayed on the screen. For a specification of a form with two/-four elements, QR-code generation takes 14 ms. The IOHUB detects the QR-code, decodes it, and creates the overlay. This process takes 6 ms for the same form considered previously.

3. *Initial Page Load* The first time the user visits a web page that employs PROTECTION, the remote server, and the IOHUB should exchange a cryptographic key to protect the communication. This step requires only one additional `xmlHttpRequest` to the server; therefore the delay is relatively low. Initially, the browser encodes the server's public key into a QR-code that is decoded by the IOHUB, which sends the response to the server by simulating the keystrokes.

4. *Frame processing for mouse* IOHUB processes every frame of the host for pointer detection. This takes 1.76 ms, which does not impact the frame rate. The image analysis routine achieves an accuracy of 0.997.

5. *Keystroke latency* The IOHUB intercepts all user's keystrokes and forwards them to the host or renders them on the screen. When rendering on the screen, the latency is 170 μs.

6. *Cursor latency* Similarly to keystrokes, the IOHUB intercepts mouse events also. However, the latency of event forwarding is 250 μs.

7. *Codebase comparison* In Table 5.2, we provide the code base and executable binary sizes of IOHUB with respect to some of the most popular open-source browsers, JavaScript interpreter engines, and OS's. All of the codes are measured with the `cloc` open-source code line counting tool. The table shows that PROTECTION has a significantly lower code base, resulting in a smaller attack surface.

8. IOHUB *cost* We estimate that our IOHUB prototype costs around 140 USD (Rpi4 = $35 + HDMI-CSI =$30 + Due = $35 + Zero = $40). An

	Projects	LOC
Browser	Chromium [a] [156]	25,163,547
	Mozilla Firefox [157]	20,928,358
JS Engine	Chrome V8 [158]	2,009,183
	Firefox SpiderMonkey [159]	2,908,550
OS	Ubuntu 19.10 w/o kernel	600,712
	Arch Linux w/o kernel	71,188
	Linux Kernel	36,680,915
IOHub	HDMI interceptor + overlay	1,911
	USB stack	893
	Crypto stack	3,500
	RPi tiny core Linux	121,899
	IOHub total codebase	128,203

TABLE 5.2: **ProtectIOn IOHub code-base comparison** with respect to some of the open-source browsers, JS engines and OSs.

a Browsers such as Google Chrome and Microsoft Edge are based on the Chromium project

integrated, mass-produced device would be, of course, significantly cheaper.

5.9 SUMMARY OF RELATED WORK

In Table 5.3, we summarize the existing research work based on their trust assumptions, IO security features, and usability. Note that it is desirable to have a lower trust assumption, higher security features, and higher usability. The trust assumption is further refined into hardware trust assumption that includes TEE and external trusted hardware, and software trust assumption, which includes isolated device drivers/APIs and trusted hypervisor/OS. The IO security features involve input that includes keyboard, pointer and touch input, and output that only includes the display. Lastly, the usability aspect is divided into two, the requirement of security indicator (SI), and if the solution supports plug-and-play (PnP). PnP implies that the solution can be integrated into the existing system without introducing any major

Security Requirements		R4				R1 R2				R3a/b
Category	Solutions	Trust Assumption				IO Security Features				Usability
		Hardware		Software		Input			Output	
		Requires TEE	External trusted HW	Isolated API/Drivers	Hypervisor/ OS	Keyboard	Pointer	Touch	Display	No SI / PnP
Hypervisor/OS-based	Shadowcrypt [160]			✔	✔		✔			✔
	Browser-based [161]			✔	✔				❑	✔
	InContext [128]				✔		✔			✔
	Overshadow [128]				✔					
	Virtual ghost [129]				✔					
	TrustVisor [131]				✔					
	Inktag [130]				✔					
	Splitting interfaces [132]				✔	✔			✔	
	SP^3 [133]				✔	✔				
TEE-based	SGX IO [36]	✔		✔	✔	✔				
	SchrodinText [162]	✔			✔				✔	
	BASTION-SGX [38]	✔				✔				✔
	Slice [163]	❑								
	TrustOTP [164]	✔				✔			❑	✔
	VeriUI [165]	✔		✔		❑			❑	
	AdAttester [166]	✔		✔				❑	❑	
	TruZ-Droid [140]	✔		✔		✔			❑	✔
	TrustUI [167]	✔		❑				❑	❑	✔
	VButton [139]	✔		✔		❑		✔	✔	
External HW	CARMA [168]	✔	✔							✔
	ProximiTee (Chapter 6)	✔	✔	❑		✔				✔ ✔
	Fidelius [14]	✔	✔	✔		✔			❑	
	FPGA-based [32]		✔			✔			✔	
	IntegriKey [114]		✔	❑		❑				✔ ✔
	Terra [169]		✔	❑						
	ProtectIOn		✔			✔	✔	✔	✔	✔ ✔

✔ requires/supports ❑ partially requires/supports

TABLE 5.3: **Summary of existing trusted path solutions** by their trust assumptions, security features, and usability. A lower trust assumption, a high number of security features and high usability are desired from a trusted path solution. SI and PnP stand for security indicator and plug and play respectively. The table also categorizes the trust assumptions, IO security features and usability in-terms of the required security and functional properties that we list in Section 5.2.3).

changes into them and supports different architectures and OS out of the box.

INTERPRETING THE TABLE. The top of the table provides the required security and functional properties that are provided by PROTECTION. We list these properties in Section 5.2.3. The trust assumption requires as minimum assumptions as possible (property R4). High number of IO security features are more desirable because of properties R1 and R2. The last category that is the usability of a system (in terms of low cognitive load on the users – R3a and R3b) can be improved if the security is not dependent on a

security indicator, and the system provides a plug & play solution. Hence the systems with more entries in this category have better usability.

5.10 CONCLUSION

ProtectIOn provides a remote trusted path in the presence of an attacker-controlled host. The guiding principles behind our solutions are that (i) user input and output integrity cannot be considered separately, (ii) all user input modalities must be protected simultaneously, and (iii) user input integrity protection should not rely on user tasks that are prone to habituation and easily forgotten. By following these principles, we design a novel system that provides strong user input integrity protection in the presence of a powerful adversary that controls the entire host platform.

Part IV

RELAY ATTACK ON TEE: EFFECTS ON TRUSTED PATH

6

PROXIMITEE: HARDENED SGX ATTESTATION BY PROXIMITY VERIFICATION

How difficult it is to be simple!
— Vincent Van Gogh

6.1 INTRODUCTION

In the previous chapter, we discussed ProtectIOn, where we investigate the fundamental security properties required by a trusted path. This section shows that porting a trusted path application into a trusted execution environment such as Intel SGX is non-trivial. Trusted execution environments (TEEs) like Intel's SGX enable protected applications that are called enclaves that are isolated from the operating system. The CPU microcode enforces the isolation and enclave life-cycle mechanism. Remote attestation is a key feature of SGX and other similar TEE architectures. It allows a remote verifier to check that the attested enclave was correctly constructed before provisioning secrets to it[1].

6.1.1 *Relay attacks*

While SGX's 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. An attacker that controls the OS (or other software) on the target platform can relay incoming attestation requests to another platform. Such relay attacks are a long-standing open problem in trusted computing, as already a decade ago, Parno identified such attacks in the context of TPM attestation [21].

Upon a first look, it might seem that relay attacks do not pose a problem for TEEs. If the attacker relays the attestation to another machine, the same security guarantees should hold since the data will only be available within the remote TEE, and the enclave code that can access the provisioned secrets is verified. However, such simple reasoning is incorrect.

1 We discuss SGX attestation in detail in Background Chapter (refer to Section 2.2.1).

In this chapter, we provide the first careful analysis of the *implications* of relay attacks on SGX and show that by relaying, the attacker increases his capabilities to attack the attested enclave significantly. One example of increased adversarial capabilities is physical side-channel attacks. If the attacker redirects the attestation to a platform that he physically controls, he can mount various physical side-channel attacks, like [170, 171, 172, 173], that would not have been possible without the relay. Another example is enhanced side-channel attacks. While controlling the OS is in the SGX attacker model, it is not unrealistic that an attacker might be in the situation of controlling only user-privileged code on the target platforms. This degree of control, however, allows the attacker to redirect attestation to another platform where he controls the OS, which allows him to launch software-based side-channel attacks, such as [43, 44, 45], that leverage system privileges to attack enclaves. In Section 6.2, we explain further examples of attacks that are enabled by attestation relay.

A typical "solution" to relay attacks is to assume *trust on first use* (TOFU). However, in many application scenarios, TOFU is neither secure nor practical. For example, solutions where attestation is performed immediately after a fresh OS installation cannot be applied to settings where OS reinstallation is simply not possible. Besides, all TOFU variants assume that the target platform OS is trusted, even if momentarily, which violates SGX's trust model.

The SGX attestation protocol is designed to be anonymous. The protocol is based on EPID group signatures [41], and thus the remote verifier cannot distinguish whether the correct enclave on the target platform was attested or if the attestation was redirected to another platform. Upon first inspection, it may seem like relay attacks are only possible because of such anonymity features and that relaying could be easily prevented if attestation protocols were designed to be non-anonymous. However, such simple reasoning is incorrect as well. We show that all SGX attestation variants, including the "linkable" attestation mode and the recently introduced Data Center Attestation Primitives (DCAP) [42] are vulnerable to relay attacks. We also explain why relay attacks would remain possible, even if all anonymity features would be removed from the attestation.

6.1.2 *VM pinning and Revocation*

Modern data centers provide flexible resource management for optimal allocation of resources such as storage, memory, network interface, etc.

Clients are provided with virtual machine (VM) instances, and more than one client VMs can run on the same physical platform. Due to the flexible nature of resource allocation, the VMs are often not physically bound to a specific platform. I.e., a data center operator can dynamically allocate a piece of its physical infrastructure and migrate the VM. This "elastic" nature of VM enables the data center operator to claim more physical infrastructure (CPU cores, DRM, NIC, disk) as the client demand increases. However, in many applications where the sensitive nature of the data is critical, it's necessary to ensure that the user's VM stays bound to a physical platform. Such property is highly desirable in many application scenarios. E.g., a client would like to keep all her banking data inside her country of residence, or regulations such as GDPR may mandate the operators to store the user data in a specific country. This is particularly challenging when considering TEEs like Intel SGX due to its lack of platform identity. Aside from the relay attack, revocation of a physical platform is a challenging aspect of data centers. Revocation enables the data center operators to revoke a certain set of physical platforms either for maintenance or discovery of a vulnerability.

6.1.3 *Our solution*

We propose a new solution, called ProximiTEE, that prevents relay attacks by leveraging a simple 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 not acceptable. Attestation of servers at cloud computing platforms and setup of SGX-based permissioned blockchains are two such examples.

In ProximiTEE, the remote verifier establishes a secure connection to the embedded 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 simple distance-bounding protocol [174]. 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.

6.1.4 *Main results.*

Parno [21] identified distance bounding as a candidate solution to TPM relay attacks already ten years ago, 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 evaluate ProximiTEE, we implemented it using a USB 3.0 prototyping board. The main purpose of our evaluation is to demonstrate that the attacker cannot redirect the attestation *over the Internet* to an attacker-controlled platform without being detected. We focus on such relay attacks, as it offers the most increased capabilities to the attacker (e.g., physical attacks). The secondary purpose of our evaluation is to determine whether proximity verification can prevent relay to a co-located platform, like another server on the same server rack. Such relays are typically less harmful, but ideally, they should be prevented as well.

In our evaluation, we *simulate* a strong attacker that i) is only a single network hop away from the target, ii) performs the required protocol computations instantaneously, iii) has infinitely fast hardware interface, and iv) has enabled software-based packet forwarding optimizations on the target platform. We measure the legitimate challenge-response latency on our prototype to be 185 μ s on average. In the case of the simulated relay attack, the average latency is about 264 μs. These two latency distributions are distinguishable and allow us to set our proximity verification protocol parameters such that the attacker's probability of performing a successful relay attack is negligible (3.55×10^{-34}), while legitimate verification succeeds with a very high probability (0.999999977). Importantly, the attacker cannot increase his success probability with repeated attempts, as attestation is triggered by the trusted remote verifier. Our experiments also show

that enclave revocation using periodic proximity verification is both secure and practical.

The performance overhead of proximity verification is small: the initial proximity verification adds only a small delay to the attestation protocol, and the periodic proximity verification consumes only a very minor fraction of the available USB 3.0 channel capacity. Our implementation shows that the complexity of such a device can be small: the software TCB of our prototype is 3.8 KLoC.

EMULATION ATTACKS.. Additionally, we consider a stronger attacker that has obtained leaked, but not yet revoked, attestation keys and can *emulate* an SGX-enabled processor. Proximity verification alone cannot prevent 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 reinstallation) and increased security (e.g., OS not trusted even temporarily).

6.1.5 *Contributions*

This chapter contributions are organized as follows:

1. *Analysis of relay attacks.* While relay attacks have been known for more than a decade; their implications have not been fully analyzed. In Section 6.2, we provide the first such analysis and show how relaying amplifies the attacker's capabilities for attacking SGX enclaves.

2. PROXIMITEE*: Addressing relay attacks.* In Section 6.3, we propose a hardened SGX attestation mechanism based on an embedded device and proximity verification to prevent relay attacks. PROXIMITEE does not rely on the common TOFU assumption, and hence, our solution improves the security of previous attestation approaches. Note that

the distance bounding approaches are well-known in the literature, but using such a method in the context of SGX is non-trivial.

3. *Experimental evaluation.* We implement a complete prototype of ProximiTEE and evaluate it against a very strong and fast attacker. Our evaluation in Section 6.6 is the first to show that proximity verification can be both secure and reliable for TEEs like SGX.

4. *Addressing emulation attacks.* We also propose another attestation mechanism based on boot-time initialization to prevent emulation attacks. This mechanism, described in Section 6.7, is a novel variant of TOFU with deployment, security, and revocation benefits.

6.1.6 *Organization of this Chapter*

The rest of this chapter is organized as the following. Chapter 6.2 describes the problem of relay attack and the trust on first use (ToFU) properties of all the variants of Intel SGX remote attestation. In Section 6.3, we address the relay attack by using the distance bounding protocol using a trusted embedded device. Section 6.4, 6.5 and 6.6 describes the security analysis, implementation and evaluation of the distance bounding mechanism. In Section 6.7 we discuss the emulation attacker, which is stronger than the relay attacker as the attacker has access to at least one leaked attestation key. In this section, we describe the bootstrap mechanism to address the emulation attacker along with the security analysis and implementation of the mechanism. In Section 6.8, we provide a system design that enable trusted path using ProximiTEE. Section 6.9 provides additional discussion, and Section 6.10 concludes this chapter.

6.2 RELAY ATTACK ANALYSIS

In this section, we provide an analysis of relay attacks on SGX.

6.2.1 *Relay Attacks*

We consider a system model shown in Figure 6.1 that consists of three parties: the target platform, the remote verifier, and the attacker's platform. The remote verifier is a trusted party that wishes to connect and attest to a specific SGX platform. The target platform is the SGX platform to which the remote verifier intends to connect. Finally, the attacker's platform is a

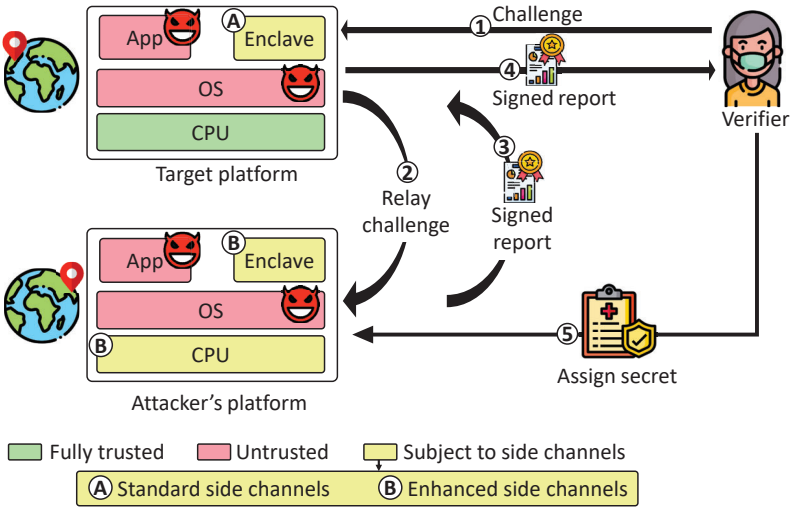


FIGURE 6.1: **An example relay attack.** The attacker redirects attestation to his own platform which gives him increased (side-channel and kernel-level) abilities to attack the attested enclave. The verifier attest to the enclave running on the attacker's platform rather than the target platform and assigns her secret to the attacker's platform.

platform owned by the attacker, connected to the target platform through the Internet.

ATTACKER MODEL. We consider the following attacker model that we call the *relay attacker*. The relay attacker controls the OS and all other privileged software on the *target* platform at least *temporarily*, particularly at the time of the remote attestation. The OS compromise on the target platform maybe later detected and disinfected. We consider the case in which the target platform resides in a data center or otherwise in a facility with restricted physical access. Hence, the attacker hence *does not* have physical access to the target platform (or any other co-located platform in the same facility).

The relay attacker controls the OS and all other privileged software on the attacker's 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 attacker has not been able

to extract attestation or sealing keys from his platform or any other SGX processor.

THE PROCESS OF THE RELAY ATTACK. The relay attacker can redirect the attestation requests intended for the target platform to his platform, as shown in Figure 6.1. This is a realistic attack for two reasons. First, in the SGX attacker's model, the attacker can control the OS and, hence, easily redirect any network request the target platform receives. Second, even if the attacker cannot compromise the OS in the target platform, it might be able to exploit some vulnerability of the untrusted application managing the enclave. The exploit might allow the attacker to manipulate the application's control flow to redirect attestation requests to any platform he desires.

The process of the relay attack is the following:

① The verifier sends an attestation challenge to the target platform.

② The attacker-controlled OS on the target platform relays the attestation challenge to the target platform over the network. Note the attacker's platform also has the same enclave running.

③ After receiving the relayed challenge, the attacker's platform generates a signed report and relays it back to the target platform.

④ The target platform sends the signed challenge from the attacker's platform to the verifier.

⑤ Assuming the report is from the target platform, the verifier assigns her secret to the enclave running on the attacker's platform over the secure channel.

6.2.2 *Relay Attack Implications*

Although relay attacks have been known for a long time [21], their implications to modern TEEs like SGX have not been carefully analyzed. Next, we perform the first such analysis.

The main consequence of attestation relay is that it *increases the attacker's ability to attack the attested enclave* through side-channels which are a well-known limitation of SGX (see Section 2.2.4). In Figure 6.2 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

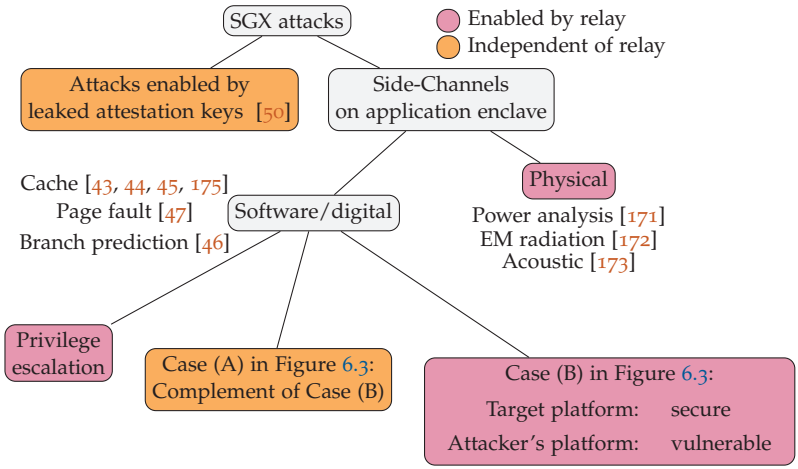


FIGURE 6.2: **Relay attack implications on Intel SGX remote attestation.** The tree shows the types of attacks that are enabled by relay and ones that are independent of relay.

done whether or not the attacker also executes a relay attack, which we call "independent of relay."

ATTACKS USING LEAKED ATTESTATION KEYS. Our first observation is that attacks are based on leaked attestation keys (e.g., ones obtained through the Foreshadow attack [50]) are independent of relaying. If the attacker 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.

PHYSICAL SIDE CHANNELS. One major benefit of the relay, from the attacker's point of view, is that it enables *physical* side-channel attacks against application enclaves. Once a secret has been provisioned to the attacker's platform, she has as much time as she likes to perform the attack. Some examples of physical side-channel attacks are acoustic, electric, and electromagnetic monitoring, which has been shown to be both effective and inexpensive means to extract secrets from modern PC platforms (see [170] for a summary of known attacks). Since the attacker does not have physical access to the target platform, such attacks are clearly not possible without a relay. Hardening programs like enclaves against physical side channels is difficult and currently an open problem [170]. Therefore, developers cannot

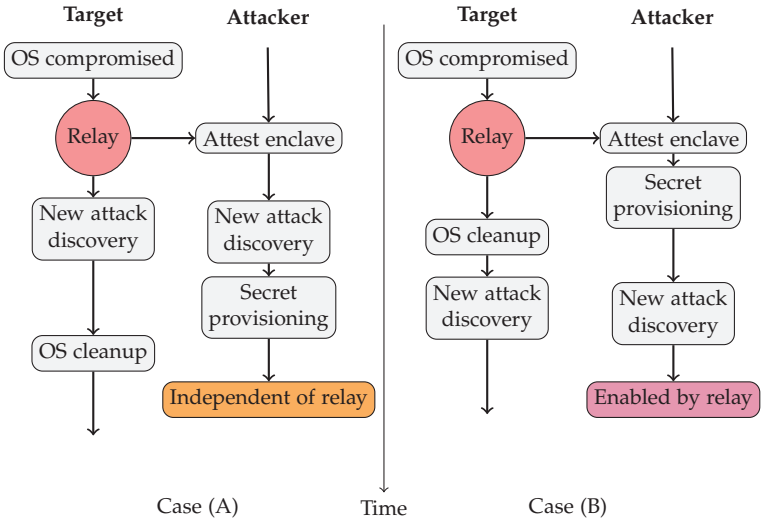


FIGURE 6.3: **Relay timing of attack.** In Case A the attack success is independent of relay. In Case B attestation relay enables the attack.

easily defend their enclaves against physical side channels that are enabled by attestation relay.

PRIVILEGE ESCALATION FOR DIGITAL SIDE CHANNELS. Another possible benefit of relay attacks is that it may enable *privilege escalation*. In cases where the attacker has only compromised the user-space application that manages the enclave and not the OS, the application can redirect the attestation to the attacker's remote platform where he controls the OS as well. In such cases, the relay enables *digital* side-channel attacks that require system privileges. Several such attacks have been recently demonstrated against SGX [43, 44, 45].

ATTACKS THAT DEPEND ON THE TIMING OF EVENTS. The third, and perhaps the most subtle, implication of relay is that it can also enable software-based side-channel attacks that would not be possible to launch on the target platform due to *timing of certain events*. 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 that are independent of relay, and Case B covers event sequences in which relay gives extra capabilities to the attacker. Below, and in Figure 6.3, 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 attacker could perform it on the target platform as well. An example of such a case is shown in the timeline depicted in Figure 6.3, where a new attack is discovered after secret provisioning but before the target platform, OS is disinfected.
>
> *Case B: attack enabled by relay.* This case 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. A timeline of such a case is shown in Figure 6.3, where at the time of attestation and secret provisioning, the enclave is hardened against all known digital side-channel attacks (using tools like Raccoon [176], ZeroTrace [177] or Obfscuro [178]). 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 attacker performed relay and the secret was provisioned to the attacker's machine, the new side-channel is exploitable. Without the relay, the attack is not possible.

6.2.3 *Limitations of Known Solutions*

Next, we review commonly suggested solutions and their limitations.

TRUST ON FIRST USE. A common "solution" in the research literature is to rely on *trust on first use* (TOFU) [179]. Simple TOFU 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. OS re-installation is not always possible. Moreover, trusting the OS, even if momentarily, is undesirable (and violates the SGX's trust model).

SGX ATTESTATION VARIANTS. As we explain in Section 2.2, SGX supports different variants of remote attestation. Unfortunately, none of these schemes prevents relay attacks without some form of TOFU assumption.

1. *The EPID attestation scheme* is based on group signatures, and thus the remote verifier cannot distinguish between attestation responses that are received from the expected target platform or the attacker's platform. To accept a successful attestation, the remote verifier must rely on trust on first use.

2. *The linkable EPID attestation mode* allows the remote verifier to check if he has attested the same platform before, but the first attestation protocol run is vulnerable to relay attacks, and therefore also, in this case, the remote verifier must assume TOFU.

3. *The DCAP scheme* allows corporations to operate their own local attestation services after an enrollment phase. However, if the attacker controls the target platform during the enrollment, he can replace the enrolled platform identifier PPID with the identifier of his own platform PPID' and enroll the attacker's platform instead. Thus, also the DCAP variant scheme requires trust on first use. In addition, the entire corporate key management system must be trusted at the time of the enrollment (and after it).

NON-ANONYMOUS ATTESTATION. Because SGX's attestation protocol support anonymity features, like the EPID signature scheme, one may think that relay attacks are caused by such a privacy protection mechanism. However, such reasoning is incorrect. Even if all anonymity features would be removed from attestation, the problem of relay attacks would still persist. The root cause of relay attacks is that certified keys can be securely installed to processors at the time of manufacturing, but the processor ownership by private individuals or companies is established much later. Therefore, common PKI mechanisms do not eliminate relay attacks — unless the processor manufacturing and distribution model is completely changed such that factories start to manufacture and certify customer-specific processors batches on demand (which would be very expensive).

OTHER TOFU VARIANTS. Recent research papers use slightly different TOFU variants. For example, the ROTE system [180] 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. As another example, in the VC3 system [181] 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.

In general, TOFU solutions suffer from 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.

6.3 PROXIMITEE

Our goal is to design a solution that addresses the above limitations of previous solutions. In short, our solution should be *secure* (no TOFU assumption, 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 our solution in detail, and analyze its security.

6.3.1 *Approach Overview*

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

Our main idea is to use the combination of such a 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 use a simple API that facilitates communications between the enclave and the device.

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 has significantly smaller software TCB, attack surface, and complexity compared to a general-purpose commodity 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 attacker model introduced in Section 6.2 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).

6.3.2 *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 three example 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 a time, such a deployment enables the authority to control the identities used in a (Byzantine) blockchain consensus process.

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 a 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.

6.3.3 *Solution Details*

Now, we explain the ProximiTEE attestation mechanism in detail.

ATTESTATION PROTOCOL. Figure 6.4 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

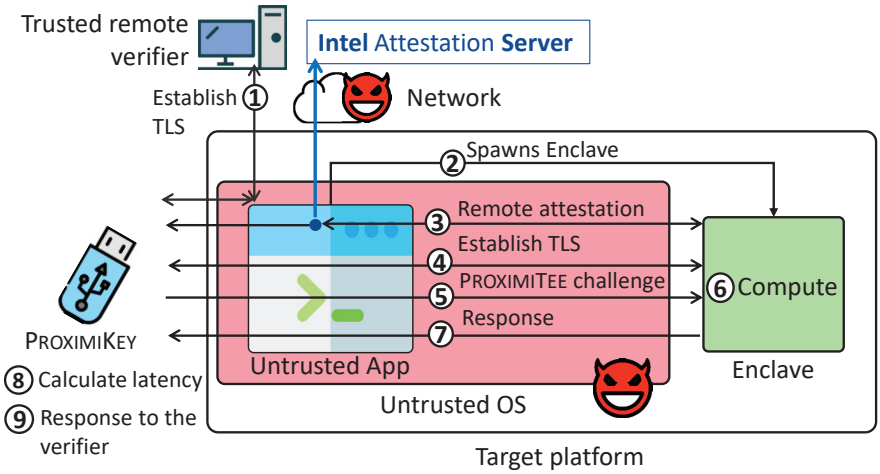


FIGURE 6.4: **ProximiTEE attestation.** The remote verifier establishes a secure channel to the PROXIMIKEY device that first attests the enclave and then verifies its proximity.

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 or using a custom DCAP procedure (see Section 2.2.3). 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.

⑥ The enclave increments the received challenge by one $(r + 1)$.

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

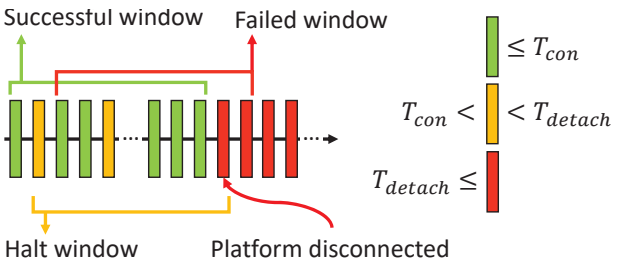


FIGURE 6.5: **Sliding window** for periodic proximity verification with three different types of challenge-response latencies.

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 at least $k \times n$ responses, where $k \in (0, 1]$ is a percentage of the total number of responses n.

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

PERIODIC PROXIMITY VERIFICATION. After the initial connection establishment, 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 in a sliding window data structure, as shown in Figure 6.5.

As elaborated in Section 6.6, 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 6.5). 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 responses has latency $\geq T_{detach}$, the current window legitimate, and ProximiKey keeps the connection active.

2. *Halt window: prevent communication.* If one of the responses has 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 thus revokes the attested platform.

6.4 SECURITY ANALYSIS

In this section, we will informally discuss ProximiTEE's attestation and revocation security.

6.4.1 *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 attacker cannot compromise the specific ProximiKey used. If the attacker manages to extract keys from other ProximiKey devices, he cannot trick the remote verifier into connecting 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 attacker that is connected to the target platform with a fast network connection. To consider the best case for the attacker, we make several assumptions in his favor. For example, we assume that he can instantly perform all computations needed to participate in the proximity verification protocol. However, he cannot break cryptographic hardness assumptions. We define the attacker's success as the event in which proximity verification succeeds with an enclave that resides on the attacker's platform and denotes the probability of such event P_{adv}. We define legitimate success as the event in which proximity verification succeeds with an enclave that resides in the target platform and denotes its probability P_{legit}. In Section 6.6 we show that it is possible to find parameters ($n = 50$, $k = 0.3$ and $T_{con} = 186\ \mu s$) that make proximity verification very secure ($P_{adv} = 3.55 \times 10^{-34}$) and reliable ($P_{legit} = 0.999999977$).

6.4.2 *Revocation security*

To analyze the security of the periodic proximity verification, which 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 attacker who does not have physical access to the target platform (recall Section 6.2.1), 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 attacker can manage to keep the previously established connection alive within a "short delay" after the ProximiKey was detached from the attested target platform. For most practical purposes, we consider a delay of 10 ms as sufficiently

short. We denote the attacker'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 6.6 we experimentally shows that revocation can be secure ($P'_{avd} = 3.55 \times 10^{-34}$) and reliable ($P'_{fp} = 1.6 \times 10^{-4}$) while consuming only a minor fraction of the available channel capacity.

6.5 IMPLEMENTATION

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

6.5.1 ProximiKey

We have two prototypes, one based on USB 2.0 and another based on USB 3.0, to show the easy deployability of the prototype. Our USB 2.0 ProximiKey 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 a high-speed 480 Mbps connection. There are two ways an Arduino Due board can be connected to a host system: i) by the programming port which is serial over the USB connection, and ii) the native USB port where the Due acts as a native USB device. The USB over serial relays all the communication packets via the UART chip, which has around 1 ms delay and achieves the maximum of 115200 baud per second throughput. We opted for the native USB port as it uses a faster USB 2.0 decoder and can achieve significantly better speed (according to the USB specification, the speed between the host and the device is negotiated dynamically).

Our USB 3.0 device prototype is based on Cypress EZ-USB FX3 SuperSpeed USB controller prototyping board [182] that is equipped with a 32-bit 200 MHz ARM926EJ core, 512 KB SRAM, and a USB 3.1 peripheral. The board communicates with the target platform over a native USB 3.0

connection that provides up to 5 Gbps of bandwidth. FX3 provides direct memory access (DMA) out of the box through its API for efficient communication with the connected platform. We use the ARM mbed TLS [183] cryptographic library for the TLS.

PORTABILITY. We wanted the ProximiKeys software to be device-agnostic in our implementation. We achieved this by 1) using the mbed TLS library for cryptography and 2) limiting the device-specific functionality in isolated modules. Thus, to use another device as ProximiKey, only the mbed TLS library has to be ported.

CHANNEL ENCRYPTION. The implementation supports three modes of channel encryption between the ProximiKey and the target platform. Mode selection is controlled by a combination of a run-time parameter and flags set at compile-time.

1. *None.* In this mode, the communication is carried out in plain text. This mode is primarily used for debugging, measuring the channel capacity, and evaluating the overhead of the encryption over the distance bounding measurement.

2. *AES-ECDH-HMAC.* This mode provides a minimal secure channel between the ProximiKey and the target platform. With this mode, Curve25519 Diffie-Hellman key exchange is performed. Relevant parts of messages, specifically the challenge-response packets, are encrypted with AES-128-CTR. AES-HMAC is used for message authentication codes. SHA256 is used as a hashing function. Note that the USB 2.0 implements this mode only due to the storage constraint. We use the Arduino cryptographic library [184] to implement the crypto suite.

3. *TLS.* With this configuration, we use a TLS channel for the communication between the ProximiKey and the target platform. As the ProximiKey only has limited SRAM, `TLS_PSK_WITH_AES_128_CCM_8` was chosen as the cipher suite as this provides the best trade-off between the code size and the security. However, note that the limitation of the cipher suite is purely due to the restricted hardware resources of the FX3 board. The target platform is not restricted by this and supports more cipher suites.

CHALLENGES. While implementing the ProximiKey, we faced various challenges.

1. *Limited System Memory* The board features 512 KB memory. By default, 180 KB are allocated for code and 32 KB as a data area. However, a simple FX3 application with the required firmware libraries already comes in at roughly 100 KB. This turned out to be a problem when integrating mbed TLS. Their SSL example program was reported to measure 160 KB. Fortunately, by using a minimal configuration, we were able to reduce the size of the binary enough. This came at the cost of only supporting `TLS_PSK_WITH_AES_128_CCM_8` and limiting the maximum content length. The content length limitation is not a problem in our case, as we control both sides of the connection and our messages are small.

2. *System Clock Resolution* The firmware API call `CyU3PGetTime` only provides a resolution of 1 ms. Since we deal with roundtrip times on the scale of μs between the ProximiKey and the target platform, this timing resolution is insufficient. However, the GPIO clock operates at a faster rate. By using a counter on an I/O pin, we are able to get a resolution of roughly 0.1 μs.

6.5.2 ProximiTEE *Enclave API on the Target Platform*

The ProximiTEE enclave API consists of two parts: i) the untrusted application that facilitates the communication channel over the USB interface with the ProximiKey, and ii) the enclave component that executes the challenge-response protocol with the ProximiKey.

HOTCALLS. ECalls and OCalls are expensive. To overcome this, Weisse et al. propose a solution called HotCalls [185]. The key concept is that instead of having one thread that calls ECall and OCall functions, thereby entering and leaving the enclave context, we have two threads: one running in an untrusted context, the other in the enclave context. Both threads have access to a shared data structure (SDS) in unencrypted memory. We designate one thread to be the responder, while the other is the requester. The responder is dedicated to polling the SDS. The requester issues a request by passing its arguments to the SDS, including an ID of the function he wants to call. The responder detects these requests and calls the function indicated by the ID. Upon completion, it writes any results back to the SDS. The requester polls the SDS until the responder has indicated completion. It can then read the return values from the SDS. In order to have HotOCalls, the application thread acts as a responder, and the enclave thread is the

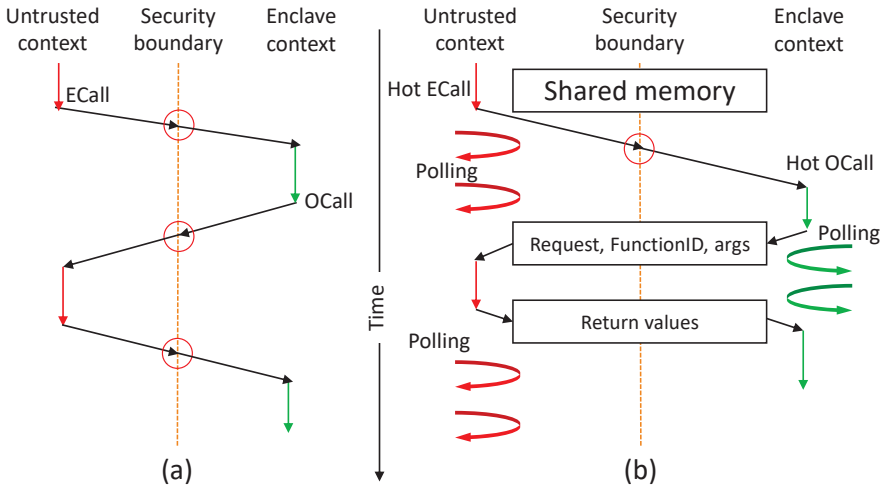


FIGURE 6.6: **Program flow control for (a) Intel SGX SDK vs (b) HotCalls.** The figure shows efficient communication between the untrusted application and the trusted enclave. In Intel SGX SDK, the developers need to rely on the `ECALL` and `OCALL` to context switch to and from the enclave. In HotCalls, the application and the enclave communication over the shared memory – eliminating the need for time-consuming context switch.

requester. This is what we have used in our implementation. For HotECalls, the roles are reversed. Figure 6.6(a) shows the control flow of a normal OCall. Figure 6.6(b) shows the control flow of the same function call but using HotCalls.

We evaluate the effect of incorporating HotCalls with our simulated client. When using SDK calls, we get an average of 12.8 μs per round. When using HotCalls, we can reduce it to 2.7 μs per round, an improvement of a factor of 4.7.

6.6 EXPERIMENTAL EVALUATION

In order to evaluate the security and reliability of our implementation of PROXIMITEE, we conducted a series of experiments.

6.6.1 *Evaluation Focus: Internet Relay*

For the purposes of our evaluation, we make the distinction between two types of relay attacks. In the first type, the attacker redirects the attestation *over the Internet* to another platform that is under his physical control, and therefore in a *different location*. As we explained in Section 6.2.2, such a relay attack amplifies the attacker's capabilities the most, as he can now attack the attested enclave using physical side-channels, he has unlimited time to launch digital side-channels, or he can wait for the discovery of new attack vectors.

In the second type of relay attack, the attacker redirects the attestation to another *co-located platform*, like another server on the same server rack. In most cases, attestation relay to a co-located platform does not improve the attacker's chances of attacking the enclave, because typically the attacker has similar control over the co-located platform. The only exception is privilege escalation in cases where the attacker has user privileged on the target platform and system privileges on the co-located platform.

6.6.2 *Experimental Setup*

To demonstrate that ProximiTEE prevents relay attacks (over the Internet), we performed two types of experiments. First, we tested the legitimate attestation execution with ProximiTEE and measured the challenge-response latencies between our prototype and the target platform. Second, we *simulated* a relay attack, where the attacker redirects the attestation to another platform.

ASSUMPTIONS AND OPTIMIZATIONS. To consider the best possible case for the attacker, we made several generous assumptions in his favor when designing our experimental setup and post-processing of our measurement:

1. *Single network hop.* Since we do not want to make any assumptions about the precise network path that the relayed attestation needs to travel, we connected the attacker's platform to the target platform via a direct 1-meter Ethernet cable, as seen in Figure 6.7. With such a setup, our goal is to simulate the most direct connectivity and the best possible latency that the attacker could achieve in relay attacks that take place over the Internet. In most realistic attacks, the attacker would need to relay the attestation over multiple network hops, which increases the round-trip latency significantly.

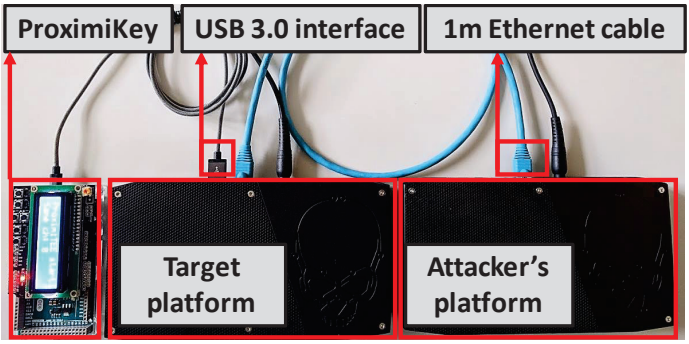


FIGURE 6.7: **ProximiTEE experimental setup** consists of the PROXIMIKEY device prototype, the target platform, the attacker's platform and the connection interfaces between them.

2. *Instant protocol computation.* Since the attacker might have a faster processor on his platform than the one we used in our experiments. We simulated an attacker who is able to perform all computations needed for the proximity verification protocol instantly. Instant replies were simulated by fixing the randomness for the challenges and having precomputed responses for that randomness on the attacker's machine.

3. *Packet forwarding optimizations.* Since the attacker controls the OS on the target platform, he can perform software-based optimizations to reduce the packet forwarding delay. We experimented with several such optimizations. First, we tested the standard `ping` tool, which gave a latency of around 380 μs for a one-meter Ethernet connection. After that, we used the `ping` tool in so-called flood mode and measured a reduced average network latency of around 153μ s (command `ping -s 300 -af`). Flood mode achieves faster round-trip time as 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. We also tested other possible OS-level optimizations but did not observe a material reduction in measured latencies, and thus in our experiments, we only use the kernel queue filling.

4. *Infinitely fast network interface.* Since the attacker's platform might have a faster network interface hardware than the one used in our

experiments; we chose to simulate an attacker that has an infinitely fast network interface. In our experimental setup, both the target platform and the attacker's platform have identical network interfaces. We assume (in favor of the attacker) that the transmission time spent on the wire is negligible, and most of the round-trip latency is due to processing in the network interface. This allows us to simulate an attacker with an infinitely fast network interface by first performing latency measurements and then in a post-processing phase cutting down all the measured latencies by half. Note that the target platform's network interface cannot be replaced by the attacker has he does not have physical access to it.

EXPERIMENTS. 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. To measure latencies, we used FX-3's GPIO pins that provide 100 nanosecond level accuracy. We performed a total of 20 million rounds of the protocol for normal attestations and simulated attacks and measured the challenge-response latencies for each. We measure all of them inside the EZ-USB FX3 code. For cross-validation, we tested the PROXIMIKEY with the high precision oscilloscope and witnessed identical timing patterns.

6.6.3 *Latency Distributions*

The histogram in Figure 6.8 on the left represents the challenge-response latencies in the legitimate proximity verification with the FX3 board. The histogram on the right shows latencies in a simulated attack (including a post-processing phase where we reduce the attacker's measured network latencies to half to accommodate the assumption of the attacker's infinitely fast network interface).

As can be seen from Figure 6.8, the vast majority of the benign challenge-responses take from 145 to 250 μs (average is 185 μ s, 95% of samples are in between 150 μ s and 200 μs). The vast majority of the round-trip times in the simulated attack take from 200 to 750 μ s (average is 264μs, 95% of samples are in between 209μs and 650 μs). Hence, the average delay of our simulated attacker is only 80 μs. To put this into perspective, even the highly-optimized network connections between major data centers in the

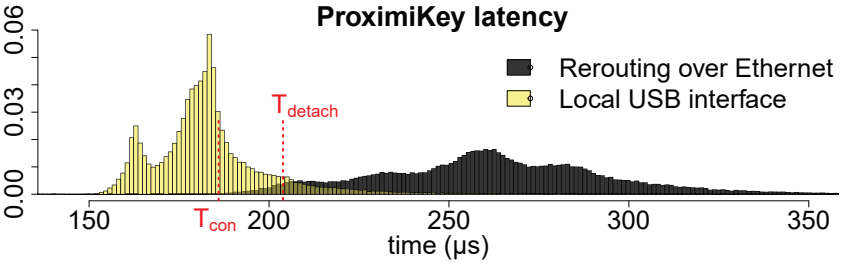


FIGURE 6.8: **Latency distributions** for legitimate challenge-response rounds (left) and simulated relay attack (right) using FX3 (USB 3.0) prototype.

same region exhibit latencies from one-millisecond upwards [186] which is one order of magnitude more than in our simulated setup.

Besides the latency observed on the side of the embedded device, we measured the time required to compute responses to received challenges on the side of the target platform. We repeated these tests on three different SGX platforms and observed results that varied from 6 to 10 μs. We also measured if the computational load of the target platform influences the time required to compute responses. Under maximum system load (all eight cores busy), the maximum observed time increased to 20 μs. Under moderate system load (1 or 2 cores busy), we experience no notable increase in the required computation time.

6.6.4 *Initial Proximity Verification Parameters*

As explained in Section 6.3.3, 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 (v) the legitimate success probability P_{legit} that should be high. We find suitable values for these parameters in the following order:

FINDING SUITABLE THRESHOLD T_{con}. Finding a suitable threshold T_{con} is a non-trivial task. A very low threshold requires a high number of the

challenge-response rounds since the protocol requires at least a fraction k of the observed responses to be less or equal to T_{con} and a low threshold has a very low cumulative probability value in the latency distribution (see Figure 6.11). Conversely, a very high threshold value enables some latencies measured during an attack to be classified as legitimate replies, hence increasing the chances of the attacker breaking the proximity verification. To address this challenge, we perform a trial over multiple threshold candidates to evaluate their viability.

Figure 6.10 shows the legitimate success probability P_{legit} for different number of rounds ($n \in \{10, 20, 50, 100\}$). We iterate through multiple threshold times ($T_{con} \in \{183\ \mu s, 184\ \mu s, 185\ \mu s, 186\ \mu s, 189\ \mu s\}$), and $186\mu s$ provides high success ratio for different values of k. $P_{legit} = 0.(9)_7 77$ for rounds $(n = 50)$, and $P_{legit} = 0.(9)_1 529\ for rounds (n = 100))$[2]

We test T_{con} up until $186\mu s$ because as can be observed in Figure 6.8 for these values, we observe extremely small occurrences (1.33×10^{-3}) of latency responses during an attacking scenario. It 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 $p_{\mathcal{H}} = \Pr[144 \leq x \leq 186] = \sum_{i=144}^{186} \Pr[x = i] = 0.693$ (where 144 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 6.8) $p_{\mathcal{A}} = \Pr[x \leq 186] = \sum_{i=160}^{183} \Pr[x = i] = 1.33 \times 10^{-4}$.

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

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

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

2 $0.(9)_n x$ denotes n times 9 after the decimal symbol followed by x (e.g., $0.(9)_2 5 = 0.995$)

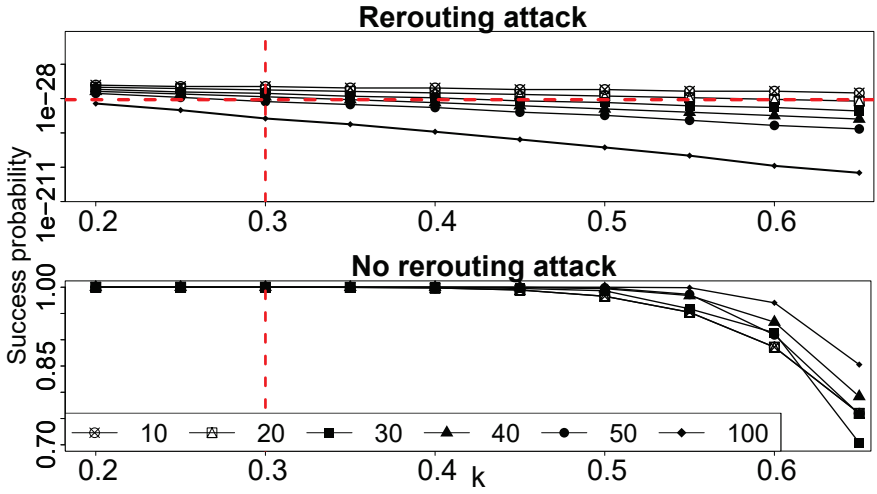


FIGURE 6.9: **Finding suitable fraction** k**.** The graph shows the legitimate enclave's success probability in an ideal scenario and the attacker's success probability in rerouting attack scenario with varying k.

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 6.9 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 $T_{con} = 186\mu s$. We observe a sharp decrease in the legitimate success probability at $k = 0.3$. Hence, fix $k = 0.3$ to achieve the maximum P_{legit}. Additionally, in the graph of attacker's success probability, the red horizontal line is placed at $10^{-30} \approx 2^{-100}$. Hence we propose to choose any round configuration bellow this horizontal line, where $n \geq 40$. With number of rounds set to $n = 50$ and $k = 0.3$, we have $P_{legit} = 0.(9)_7 7$ and $P_{adv} = 3.55 \times 10^{-34}$. Similar result could be also observed in Figure 6.9 where the success probability of the legitimate enclave decreases significantly after $k = 0.55$ for $T_{con} = 186\mu s$.

GENERALIZING THE NUMBER OF ROUNDS n. Figure 6.8 extends this analysis to the general number of challenge-response rounds spanning from $n = 2$ to 100. Here we compute the probability of the attacker returning the reply within $186\mu s$ for at least $k = 0.3$ fraction of challenges. The y-axis denotes the attacker's success probability which diminishes overwhelmingly

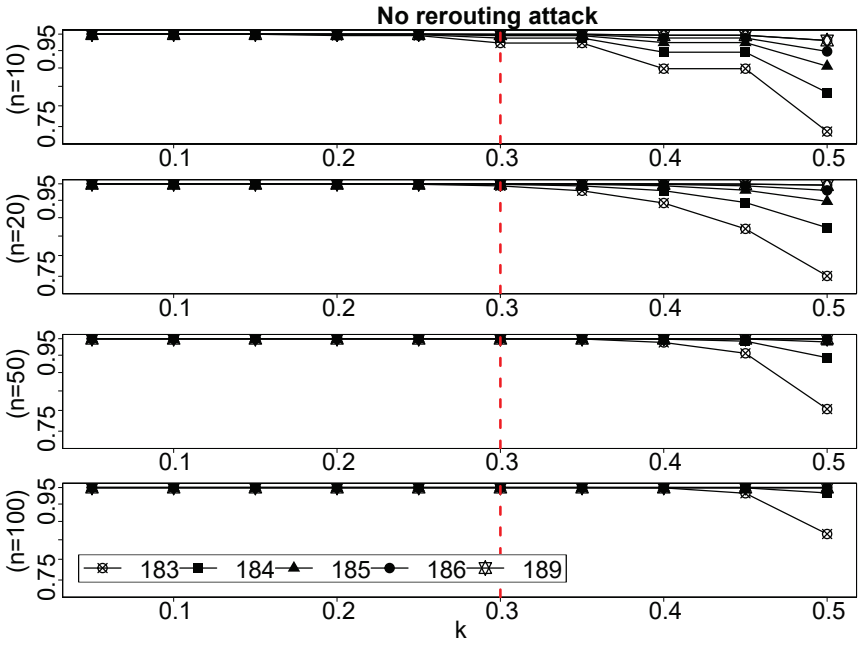


FIGURE 6.10: **Legitimate attestation success probability for different T_{con} values.** The chosen value $T_{con} = 186\ \mu s$ gives success probability 0.999999977 for number of trials at least 15 out of $n = 50$ rounds when $k = 0.3$.

with the increasing number of challenges (keeping the fraction constant at $k = 0.3$).

MAIN RESULT. Figure 6.12 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 $T_{con} = 186\mu s$ (see Figure 6.8), the threshold fraction $k = 0.3$ and the number of rounds $n = 50$ which yields a very high legitimate success probability $P_{legit} = 0.(9)_7 77$ and a negligible attacker's success probability $P_{adv} = 3.55 \times 10^{-34}$.

6.6.5 *Periodic Proximity Verification Parameters*

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

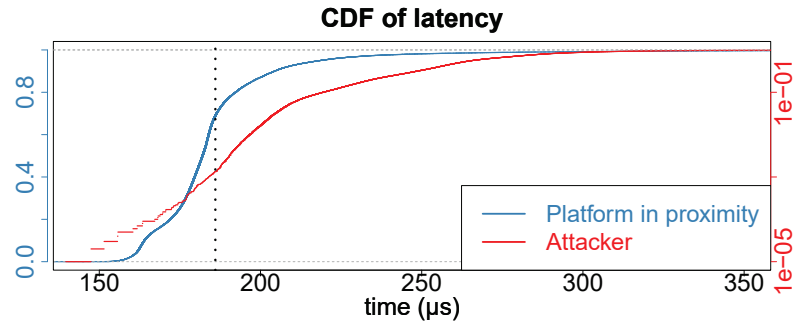


FIGURE 6.11: **Cumulative distribution function for latencies.** We set the threshold T_{con} at 183 μs which has a cumulative probability of 0.693 in the experiment where no rerouting attack takes place with probability of 1.33×10^{-4}.

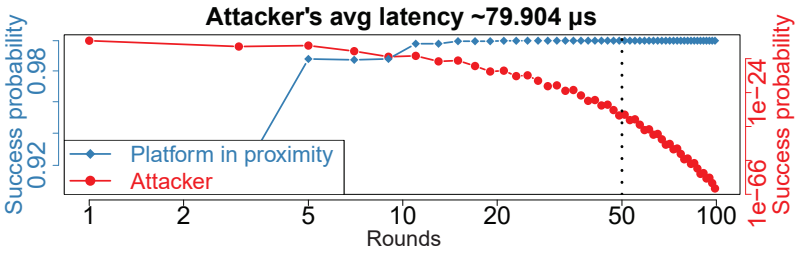


FIGURE 6.12: **ProximiTEE distinguishing relay attack.** The figure shows the attacker's success probability P_{adv} and the legitimate success probability P_{legit} for different number of rounds n given a fixed k.

not terminated sufficiently fast. Second, the probability of false positives P'_{fp} should be very low. 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:

1. We find out a suitable latency T_{detach} that defines the yellow or red round in Figure 6.5. 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 terminates the communication very fast, leaving a very small attacking window.

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 6.8) 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 6.5 for a *single round* as following:

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

FINDING SUITABLE SLIDING WINDOW SIZE w. Sliding window size is analogous to that of the number of rounds n. We keep the size of the sliding window as $w = n = 50$ as it only requires the PROXIMIKEY to remember the past 50 interactions and achieve high probability for the legitimate enclave and negligible success probability for the attacker. Similar to the previous approach, only if 20 out of 50 ($k = 0.4$) challenge-response round where responses are within 470 μs, PROXIMITEE yields success probabilities as the following:

$$\Pr[A \in \text{success window}] = P'_{adv} = P'_{fn} = 2.71 \times 10^{-67}$$
$$\Pr[A \in \text{failed window}] = \Pr[A \in \text{red}]^2 = 0.970$$
$$\Pr[legit \in \text{success window}] = 0.(9)_7 65$$
$$\Pr[legit \in \text{failed window}] = P'_{fp} = \Pr[legit \in \text{red}]^2 = 5 \times 10^{-5}$$

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 ≥ 510 μs (T_{detach}) are received, the ProximiKey terminates the connection and revokes the platform. The downtime that can happen as a result of false-positive during a connection of 10 years is around 2 minutes.

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 ~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 12 ms of the start of a relay attack and terminated in 24 ms.

SUMMARIZING THE PERIODIC VERIFICATION RESULTS. Based on the above strategy, we set the periodic proximity verification parameters as follows: $\Pr[A \in \text{success window}] = P'_{adv} = P'_{fn} = 3.55 \times 10^{-34}$, $\Pr[legit \in \text{success window}] = 0.(9)_7 77$ and $\Pr[legit \in \text{failed window}] = P'_{fp} = \Pr[legit \in \text{red}]^2 = 1.6 \times 10^{-4}$ and T_{detach} = 205 μ s (see Figure 6.8). If at least two latencies above 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.

6.6.6 *Performance Analysis*

In addition, we evaluated the following two performance metrics:

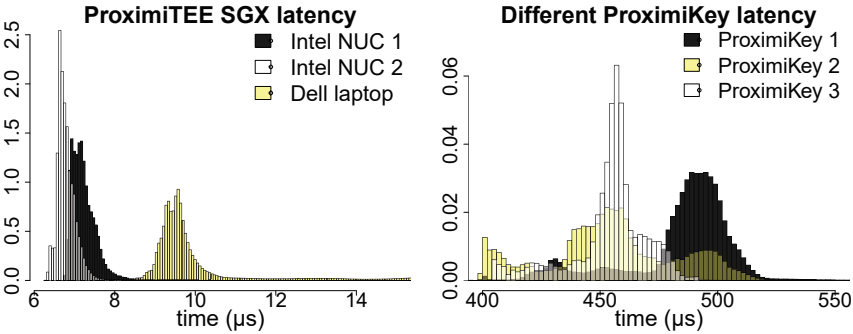


FIGURE 6.13: **Effect of different target platforms/ProximiKey on the latency.** We evaluates latencies using three different SGX platforms. The Intel NUCs were few microseconds faster. Additionally, we evaluated latencies using three different Arduino boards. The latencies are consistent.

1. *Start-up latency.* The initial proximity verification takes 2 ms. The complete connection establishment, including attestation and TLS handshake, takes less than 1 second.

2. *Operational latency and data overhead.* Our solution adds around 200 μ s of additional latency for TLS and transport over the native USB interface of the FX3. 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 156.14 KBytes/s of data which is only 2.4×10^{-3}% of the USB 3.0 channel capacity.

6.6.7 *Additional Experimental Results*

Here we provide results from additional experiments.

We evaluated the consistency of measured latencies across different computing platforms. Figure 6.13 shows the frequency distribution of latencies across three SGX platforms and three ProximiKey devices. We conclude that measurements are consistent results 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 6.14 shows (on the right) that the USB cable has very small effect on

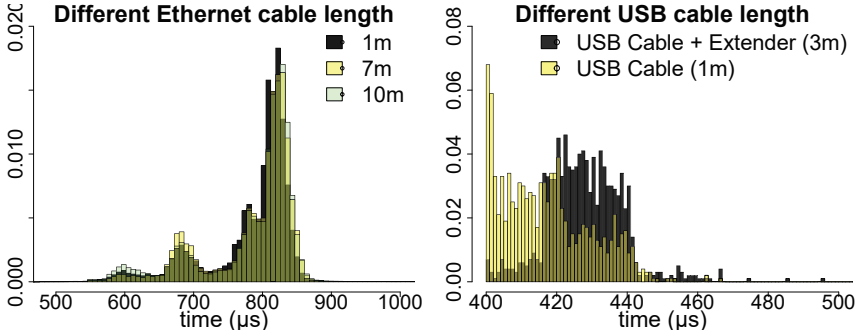


FIGURE 6.14: **Effect of different Ethernet/USB cables on the latency.** We evaluated latencies two different USB cables: one with an USB cable (1m) and another with an USB extender of length 2m attached. Additionally, we evaluated latencies using three different Ethernet cables (1, 7 and 10 m). Latencies are consistent. Note, that the latency is sampled in the experiment conducted with non-`ping` flood mode.

the latency (around 10 μ s average difference). It also shows (on the left) no significant differences between the different Ethernet cable lengths.

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 a program. CPU pinning may significantly bring down execution time due to the elimination of core switching and the ability to reuse L1 and L2 cache. Figure 6.15 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 6.16 shows the enclave execution times with a varying degree of CPU stress testing. We used `stress-ng` to stress a 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 Figure 6.16, we experienced a shift of 12 μs 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

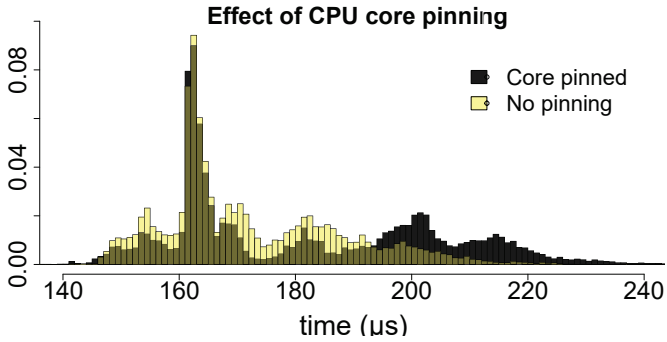


FIGURE 6.15: **Effect of CPU core pinning on the enclave application.** Restricting the enclave application to a specific core has a very minor effect on the observed latency.

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, not a security threat.

6.6.8 *Preventing Relay to Co-Located Platform*

The main purpose of our experimental evaluation was to show that our inexpensive PROXIMITEE prototype can effectively prevent relay attacks where the attacker redirects the attestation to another platform that is under his physical control in a *different location*. Next, we discuss whether PROXIMITEE can prevent attestation relay a *co-located* platform, like another server on the same server rack.

If the two co-located platforms are connected through traditional networking technologies like Ethernet (as in our experiments), our evaluation already shows that such relay attacks can be effectively prevented using a simple and inexpensive embedded device like our prototype. However, in some modern data centers, computing platforms are connected with faster inter-connect technologies like InfiniBand connections that can enable latencies as lows as 7μs [187].

The ability to distinguish relay attacks depends on three key factors. The first is the latency of the channel through which the relay is performed (e.g., 7 μs for InfiniBand). The second is the time required to compute responses to challenges on the target platform (e.g., 6-10 μs in the SGX platforms

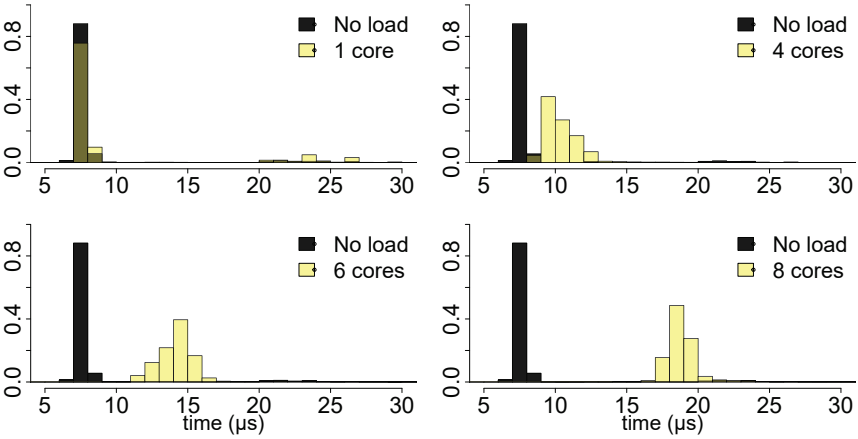


FIGURE 6.16: **Effect on latency experienced by the ProximiKey with different number of stressed CPU cores.** We evaluated latencies while running CPU intensive benchmark on different number of cores. Note that with higher number of busy cores, the means of the distributions start to shift towards right but stayed within $T_{con} = 186\mu s$. We used `stress-ng` Linux stress-testing application.

that we tested). And the third is how much variance the round-trip times between the embedded device and the target platform have (e.g., 10-20 μs in our USB 3.0 prototype). The local communication variance and the response computation time should be less than the relay latency to enable robust proximity verification.

We conclude that our simple prototype cannot prevent all possible relays to co-located platforms when very fast interconnect technologies like InfiniBand are used. To address such relay attacks, one needs a faster and more accurate embedded device that exhibits less variance. For example, PCIe-connected FPGAs can have latencies as lows 1 μs [188]. Besides using an embedded device on a faster interface, one can also increase the number of distance-bounding protocol rounds and reduce the success probability for legitimate attestation P_{legit}.

6.7 ADDRESSING EMULATION ATTACKS

We consider attestation key extraction from SGX processors difficult and rare, in contrast to the previously considered relay attacks that require only

OS control or other malicious software on the target platform. However, the recently demonstrated Foreshadow attack [50] that exploited the Meltdown vulnerability [49] showed how to extract attestation keys from SGX processors. Although 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, new vulnerabilities like the ZombieLoad attack [189] may be discovered. Before microcode patches are deployed, in rare occasions, leaked but not revoked attestation keys may be available to the attacker.

6.7.1 *Addressing the Emulation Attack*

ATTACKER MODEL. We consider an *emulation attacker* has all the capabilities of the relay attacker (cf. Section 6.2) and additionally has obtained at least one valid (not yet revoked by Intel) attestation key from any SGX platforms but the target platform. The attacker might obtain an attestation key by attacking one of his processors or by purchasing an extracted key from another party.

THE EMULATION ATTACK. In the attack, the attacker uses a leaked attestation key to emulate an SGX-processor on the target platform. Since the IAS (or any other attestation service) successfully attests to 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 attacker 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.

6.7.2 *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

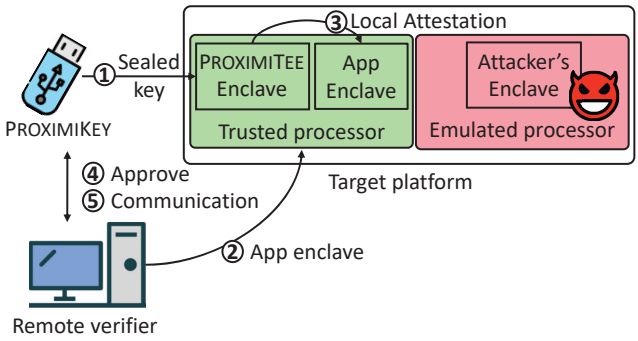


FIGURE 6.17: **ProximiTEE boot-time attestation.** After the boot-time initialization (refer to Figure 6.18) the PROXIMITEE enclave executes a local attestation with the verifier uploaded application-specific enclave.

solution can be seen as a *novel variant* of the well-known TOFU principle, and the main benefit 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 6.2. The only difference is that in this case, we assume that the UEFI (or BIOS) on the target platform is trusted.

SOLUTION OVERVIEW. Figure 6.17 illustrates an overview of this solution. During initialization, that is depicted in Figure 6.18, 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.

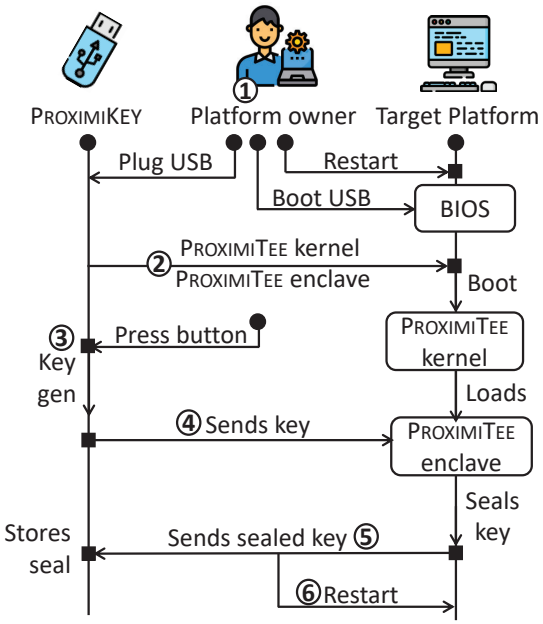


FIGURE 6.18: **ProximiTEE boot-time initialization.** The PROXIMIKEY uses a minimal kernel Linux image to boot and load PROXIMITEE enclave on the target platform and seal a platform specific secret to the PROXIMIKEY's memory.

BOOT-TIME INITIALIZATION. The boot-time initialization process is performed only once. This process is depicted in Figure 6.18 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 6.17 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 (refer to Section 2.2.2) 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.

6.7.3 *Security Analysis and Implementation*

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 attacker 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 as the platform owner indicates this by pressing a button – an operation that the attacker cannot perform. The ProximiTEE enclave seals the key during initialization. Different SGX CPUs cannot unseal each other's data, and therefore even if the attacker 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 attacker in possession of leaked sealing attestation keys from other SGX processors cannot produce a local attestation report that the ProximiTEE enclave would accept, and therefore the attacker cannot trick the remote verifier into establishing a secure communication channel to a wrong enclave.

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.

1. *Smaller TCB size and attack surface.* 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 in the BIOS (or UEFI). In our solution, the CA can remain offline when a new platform is enrolled.

2. *Reboot instead of reinstall.* 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.

3. *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.

IMPLEMENTATION. We implemented a complete prototype of our second attestation mechanism. On top of our previous PROXIMITEE implementation (refer to Section 6.5), the boot-time initialization solution requires the PROXIMITEE kernel. We have modified an image of Tiny Core Linux [190] 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.

6.8 BUILDING TRUSTED PATH WITH PROXIMITEE

One important limitation of SGX is the lack of *trusted path*. As defined in [10], 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 attacker who 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 the user's inputs to a different enclave from the one intended by the user. Under the SGX security model, the lack of a 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 [36]. The main drawback of general-purpose (commercial) hypervisors is their significant complexity

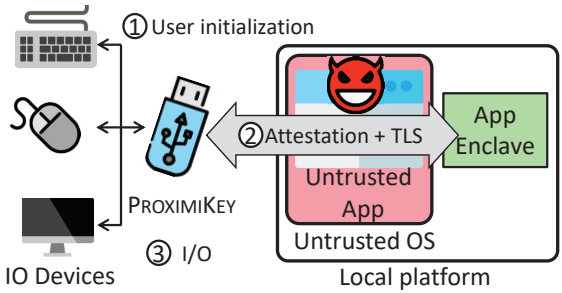


FIGURE 6.19: **Trusted path to local enclave using ProximiKey.** The IO devices are connected to ProximiKey that is connected to the local platform. The ProximiKey performs attestation using one of our mechanisms and then mediates all IO communication.

and attack surface. While the research community has also produced small and formally-verified hypervisors, like the seL4 project [127], their adoption in practice can be problematic. In addition to the secure hypervisor itself, a 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. There are solutions to establish trusted path to TEEs in sensor network such as SAKE [191] that uses a primitive called ICE (Indisputable Code Execution) [192].

6.8.1 *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 mediates all user inputs and outputs between I/O devices and enclaves, as shown in Figure 6.19. A 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 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 [193].

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.

6.8.2 *Local trusted path*

Now, we describe the process of establishing a trusted path to an enclave on a *local* platform. As shown in Figure 6.19, 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 a 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.

6.8.3 *Trusted Path to a Remote Enclave*

Next, we describe how such a trusted path can be extended to an enclave that resides on a *remote platform*, as shown in Figure 6.20. Both the local and the remote platforms 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 one of our attestation mechanisms. As the end result of the

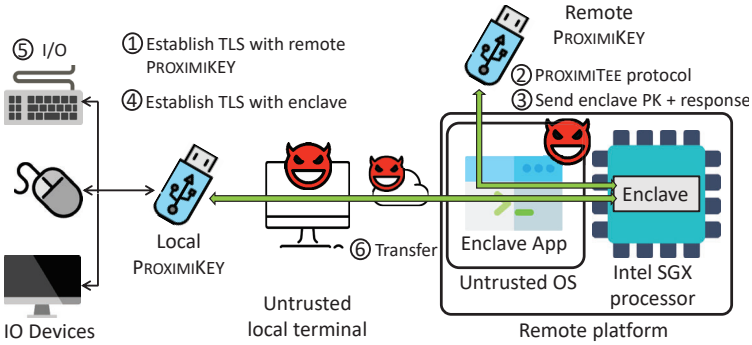


FIGURE 6.20: **Trusted path to a remote enclave using ProximiKey.** This setup uses two embedded devices. The local ProximiKey is connected to the local platform and the remote ProximiKey is connected with the remote platform.

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.

IMPLEMENTATION. Figure 6.21 shows our trusted path implementation that is based on an 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 an 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 in the enclave.

6.9 RELATED WORK

6.9.1 *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. Other TEEs that meets these requirements are ARM TrustZone and RISC-V keystone. Specifically, RISC-V keystone have a

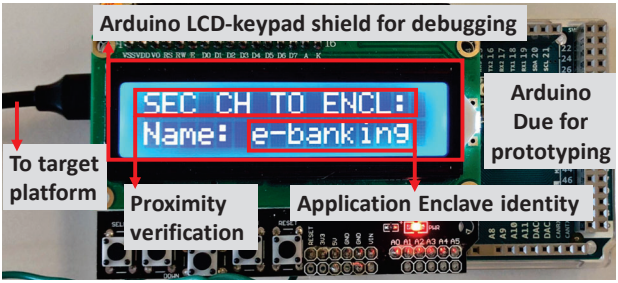


FIGURE 6.21: **ProximiTEE trusted path implementation.** The figure shows the PROXIMIKEY based on Arduino Due prototype board that is used as an I/O hub for trusted path implementation. The PROXIMIKEY also uses a small LCD display to show the identity of the SGX enclave that is is communicating with.

large number of similarity with SGX which makes it a suitable candidate for PROXIMITEE.

6.9.2 *DRTM proximity verification*

Presence attestation [194] enables proximity verification DRTM-based TEEs [89]. The TEE shows an image that is captured by a trusted camera and communicated to a remote verifier. The same approach cannot be used with SGX since it lacks trusted path for secure image output. Catching the cuckoo [195] uses timing side channel to verify proximity with a TPM emulator. The TPM latency is in the order of seconds, making it invisible for any practical distance bounding.

6.10 CONCLUSION

Relay attacks have been known for a decade, but their implications to modern TEEs like SGX have not been carefully analyzed. In this chapter, we have presented the first such analysis and shown that attestation relay increases the attacker's ability to attack an attested enclave. We have also proposed PROXIMITEE as a solution to prevent relay attacks using proximity verification. Our experimental evaluation is the first to show that proximity verification can be made secure and reliable for TEEs like SGX. As an additional contribution, we have also presented a novel boot-time

initialization solution for addressing a stronger emulation attacker who has leaked attestation keys.

Part V

TRUSTED PATH IN DISAGGREGATED TEES

7

PIE: PLATFORMWIDE-TEE FOR PERIPHERAL INTEGRATION

It is looking at things for a long time that ripens you and gives you a deeper meaning.
— Vincent Van Gogh

7.1 INTRODUCTION

Till now, in this dissertation, in ProtectIOn 5, we have seen the fundamental properties that are required by an application to implement a trusted path while using an external trusted device as an intermediary. In ProximiTEE 6, we discuss how this external intermediary can perform a distance-bound protocol to ensure that the peripherals are communicating with a specific platform in a scenario where the trusted path application is an Intel SGX enclave. All these works, including the previously discussed approaches like IntegriKey and IntegriScreen uses an external trusted device to facilitate a trusted path between the peripheral device and the application. In all of these works, we assume a traditional CPU-centric model where the remote services are executed primarily in the CPU (or an enclave running on the CPU cores) on the trusted endpoint and traditional user-facing components such as IO devices.

7.1.1 *Disaggregated Computation*

Recently we have seen a seismic shift to the traditional CPU-centric computation model. Complex devices such as GPU, network cards, accelerators, etc., are common in the datacenters as well as consumer-centric platforms. Unlike IO peripherals, these complex devices, which we call peripheral, have compute capability. For example, GPUs provide much higher performance in tasks such as machine learning over a large data set, scientific simulations. Datacenter applications that require large bandwidth often use kernel transport layer security (kTLS) to offload cryptographic operations from the CPU to the NIC, thus saving crucial CPU cycles. In these modern platform architectures, effectively CPU's main role is shifted towards a

coordinator role, i.e., setting up the computation for certain workloads [196] in the relevant peripheral and then collecting the results, possibly to feed them to yet another peripheral. Cloud computing architectures are even adopting a *disaggregated* model called *composable disaggregated infrastructure* [23, 63, 64, 69] in which data centers no longer just consist of a number of connected servers, but of functional blocks connected with high-speed interconnects. Each block provides a pool of a particular resource, be it GPUs, CPUs, storage, FPGAs, to allow for fine-grained resource allocation and acceleration. When more resources are requested, only a particular block needs to be augmented, rather than requiring the provisioning of full-fledged monolithic servers. Moreover, unlike the trusted intermediary connected to the platform over the USB interface, these peripheral devices can connect over interfaces such as PCI-express, and having an external intermediary is not feasible in such high-speed interfaces.

At the same time, the security of modern systems has also come under scrutiny due to the numerous vulnerabilities related to the high complexity of operating systems and hypervisors [19, 20]. Because of this, it has become more attractive to rely on smaller and lower layers, i.e., firmware or even immutable hardware, to enforce security and to reduce the underlying trusted computing base (TCB). Most notably, this has led to the rise in trusted execution environments (TEEs). TEE designs vary to a large degree, but, in general, they isolate execution environments without having to trust operating systems and hypervisors [16, 17, 18]. TEEs rely on hardware primitives of the CPU and only consider the CPU package to be trusted, while all the other hardware components of the platform are explicitly assumed malicious.

These two developments present an apparent disconnect: on one side, modern computer architectures are increasingly relying on peripheral for performance and scalability. On the other, TEEs provide strong security guarantees only if code and data are confined within the CPU. Using peripheral in existing TEEs either requires to trust the OS (e.g., SGX [16]) or to bloat hypervisors or specialized OS (TrustZone Secure OS [197]) with drivers. E.g., the keyboard input to an SGX enclave can be read and altered by the untrusted OS, whereas in the case of TrustZone, the security of that input depends on the large TCB, including drivers for unused peripherals. Since the *hardware* TCB of a TEE is statically decided at design time by the CPU manufacturer, end-users need to rely not only on a fixed hardware TCB but also potentially need to add drivers of other devices into the software TCB if other enclaves want to make use of them. In other words,

current TEEs struggle to support peripheral while adhering to the principle of least privilege.

7.1.2 *Our Proposal*

We propose a TEE with a *configurable* software and hardware TCB including arbitrary peripheral, a concept that we name *platform isolation environment* (PIE). PIE executes applications in *platform-wide enclaves*, which are analogous to the enclaves provided by TEEs, except that they span several hardware components. For example, a platform-wide enclave can be composed of a GPU (or only some GPU cores) and the CPU and the custom code running on them. Like in traditional enclaves, a platform-wide enclave can be remotely attested. However, the PIE attestation not only reports a measurement of the software TCB but also of the hardware components that are part of the platform-wide enclave.

The shift towards configurable hardware and software TCBs has wide-ranging implications concerning integrity, confidentiality, and attestation of a platform-wide enclave. Attestation, for one, should cover all individual components of a platform-wide enclave atomically to defend against an attacker that changes the configuration in between attestations to separate components. Moreover, the untrusted OS may remap peripheral devices at runtime with an untrustworthy device, which should not receive access to sensitive data. We carefully design PIE to not be vulnerable to such attacks and present an in-depth security analysis.

We mitigate the above-mentioned attacks with two new properties of platform-wide enclaves: *platform-wide attestation* and *platform awareness*. Platform-wide attestation expands the attestation to cover all components within a platform-wide enclave, and platform awareness enables enclaves to react to changes in their ecosystem, i.e., remapping by the OS. We achieve this by introducing two new events into the enclave lifecycle, *connect* and *disconnect*, which allow tracking the liveliness of one enclave from another.

We validate the challenges and design choices in a prototype that we develop on top of RISC-V and Keystone [24]. We make the key design decision to facilitate the communication between peripheral and the CPU with shared memory. This not only reduces the cost of context switches in enclave-to-enclave communication but also allows enclaves to communicate directly with peripheral, as these are memory-mapped and allows us to reuse existing drivers. In particular, our prototype modifies the way Keystone uses the RISC-V physical memory protections (PMP) to let enclave

memory overlap, which enables shared memory. We perform an extensive security analysis of our prototype, analyzing the implications of our design with respect to side channels, the enclave's interactions with peripherals and their lifecycles, and how attestation can now be extended to reflect the configuration of a platform and thus form a dynamic hardware TCB.

We further evaluate PIE in two case studies: first, we demonstrate an end-to-end prototype on an FPGA with simple peripherals emulated on a microcontroller; and second, we take an existing accelerator [198] and integrate it into PIE, adding support for multi-tenant isolation. In the first case study, we developed a prototype on top of an FPGA that is running a RISC-V core with Keystone. The FPGA is connected to an Arduino microcontroller that emulates IO peripherals and sensors. It required around 2.5 KLoC combined for the driver and the firmware changes to enable remote attestation. While the first case study focuses on IO peripherals and sensors that often require exclusive access by an application, in the second case study, we demonstrate how to adapt an existing accelerator [198], so it can support multi-tenant isolation and remote attestation. Here we enable multiple PIEs to concurrently use the accelerator while still giving meaningful isolation guarantees to remote verifiers. The TCB of Keystone increased by around 600 lines of code (LoC), and the additional logic in the context switch increased by 220 cycles (from around 4700 to 4900 cycles).

7.1.3 *Our Contributiuons*

In summary, the contributions of this chapter are the following:

1. *New security properties* We extend traditional TEEs with a dynamic hardware TCB, i.e., the enclave's TCB only includes the driver and firmware of the used peripheral. We call these new systems platform isolation environment (PIE). We identify key security properties for PIE, namely *platform-wide attestation*, and *platform awareness*. Additionally, we propose a software design for PIE that abstracts the underlying hardware layer and try to integrate with the existing driver ecosystem.

2. *Security analysis* We analyze the security aspects of PIE in detail. This includes the security implications of PIE's design decisions and a number of relevant side channels.

3. *Two case studies* We demonstrate two case studies: first, we present an end-to-end prototype based on Keystone [24] on an FPGA running a

RISC-V processor [25] including multiple external peripherals emulated by Arduino microcontrollers. Our modifications to the software TCB of Keystone only amount to around 600 LoC. Second, to extend the scope of PIE from IO peripherals to complex devices, we perform a case study based on a GPU-style accelerator [198] and integrate it within PIE while also enabling multi-tenant isolation.

7.1.4 *Organization of this Chapter*

The rest of the chapter is organized as the following: Section 7.2 provides the problem statement, system and the attacker model. Section 7.3 proves an overview of our approach. Section 7.4 and 7.5 present the main idea and the programming model of PIE respectively. Section 7.6 provides an proposal of modifying PIE to support local physical attacker. Section 7.8 and 7.7 describe the security analysis of PIE, and the PIE prototype & its performance respectively. Finally, Section 7.10 and 7.11 describes the related research works and conclude this chapter respectively.

7.2 PROBLEM STATEMENT

Modern platforms are composed of complex heterogeneous peripheral, from simple IO devices such as keyboard, mouse to complex devices such as a GPU for machine learning or fast NIC to process huge amount of network traffic and execute cryptographic operations on them. All these components are connected to the CPU over buses (e.g., PCI, USB, etc.). Many modern applications that leverage trusted paths are critically dependent on such peripheral, and often they handle sensitive data, e.g., patient records for machine learning. Thus, these peripherals' authenticity and integrity are critical, and the data they handle must remain confidential. We list three such trusted path applications in the following ranging from the simple IO operation to complex accelerator that provides their own execution environment:

APPLICATION 1: TRUSTED PATH FOR IO DEVICES. A user uses her IO devices (keyboard, mouse, display, etc.) to interact with an online banking application running on her device. The user wants to verify that the banking application is exclusively accessing her IO devices. In this case, the user wants to harden her system against a potentially compromised software stack.

APPLICATION 2: TRUSTED PATH TO SENSOR READINGS. Safety-critical industrial and medical devices rely on measurements of critical sensors. E.g., a pressure sensor provides data to the industrial controller, who decides when to open a safety valve.

APPLICATION 3: TRUSTED PATH TO ISOLATED EXECUTION ON ACCELERATORS. Many science fields require a vast amount of computation power, which a general-purpose processor can no longer provide. In recent times, specialized hardware which is very efficient for one specific problem is used in the form of accelerators, e.g., machine learning accelerators in the cloud. Such applications often handle sensitive data such as patient records and require isolation from the OS and other applications running on the same GPU.

In all of the above applications that handle sensitive data and use a peripheral device can be deployed securely with one of the following three existing approaches: 1) designing a fully dedicated system, or 2) renting a dedicated virtual machine and placing trust in the hypervisor, or 3) relying on the OS. None of these approaches to be satisfactory due to lack of generality, cost, and the need to trust codebases with millions of lines of code [56, 57]. Existing TEEs such as Intel SGX, RISC-V Keystone, ARM TrustZone, etc., provide security guarantees only to the applications running on the CPU cores leaving peripheral unprotected. Moreover, SGX and Keystone enclaves rely on the untrusted OS to communicate with peripheral. On the other hand, ARM TrustZone provides isolated communication between the enclaves and components such as a touchscreen, fingerprint sensor but requires trusting the entire secure OS, including device drivers not used by the enclave.

7.2.1 *Attacker Model*

The attacker model is tightly coupled with the type of peripheral. We separate the peripheral into two main classes due to their distinct effect on the attacker model:

1. *Peripheral with physical interaction:* Peripheral that interact with their environment range from input-only, such as input peripherals (e.g., mouse, keyboard) and sensors (e.g., temperature sensor) to output-only devices (e.g., a monitor) and combined IO devices (e.g., touchscreen). For any such device, a local physical adversary can manipulate the environment and thus the input (and potentially the output).

E.g., a physical adversary can point a laser at a light sensor, thus changing the sensor's reading but not the room's overall light intensity. Hence, any peripheral that interacts with its physical environment cannot tolerate a physical adversary.

2. *Peripheral without physical interaction:* There are peripheral units that do not explicitly interact with their environment. They draw power and produce heat, but their input and output are not related to the environment. GPUs and other accelerators are the prime examples of this class of peripheral, for whom a local physical adversary can be tolerated.

For the most part of this chapter, we assume a remote attacker who remotely controls the entire software stack - the OS and hypervisor. While the remote attacker model is a weaker assumption compared to the local physical one considered in the existing TEEs, the former covers a wide class of peripheral (e.g., peripheral with physical interaction) that cannot tolerate physical attackers. Hence, the attacker cannot access the platform of the peripheral physically or hot-swap a device. Note that the untrusted OS is still in charge of managing peripheral devices and thus is able to remap the devices or send a reset or power-off signal. In addition, an adversary may launch DMA attacks using rogue peripherals.

Later in this chapter, in Section 7.6, we also discuss how we can make some modifications to our proposal to adopt with a local physical attacker.

In both of the cases (remote and local attacker), we assume that the CPU firmware is trusted. Similar to existing TEE proposals, side-channel attacks remain out of scope [16] in our adversary model. However, we will discuss the implications of our proposal on existing side-channel attacks and defenses in Section 7.7. Finally, we consider denial-of-service attacks to be out of scope in this paper.

7.2.2 *Challenges*

As mentioned above, several approaches could be pursued to integrate peripheral into a TEE. Among them, we investigate approaches that reuse components of existing systems as much as possible, both in terms of software and in terms of hardware. This approach leaves the OS in a supervisor role, liaising between the isolated environments and peripheral, similar to a memory in traditional TEEs (refer to Section 2.2). But, this decision leaves leeway for privileged adversaries to break the system's

isolation. We, therefore, need to consider both existing threats to traditional TEEs and emerging threats due to the nature of a reconfigurable hardware TCB. We analyze these in more detail in the next three paragraphs.

SECURE COMMUNICATION. Traditionally, the OS or the hypervisor act as the bridge between applications and peripheral. They are responsible for set-up these communication links properly and can not only observe the data exchanged between different components but also tamper with it. As they are not trusted in our attacker model, we need to ensure that each component establishes a secure link with the other. This is not trivial, as the OS is untrusted and may not cooperate. Finally, the fact that several accelerators may need to support a form of multi-tenant isolation (e.g., multiple tasks on a GPU) requires careful consideration, as sensitive data within an isolated environment in PIE should remain confidential irrespective of what else is running on the system.

REMOTE ATTESTATION. Remote attestation is a key part of any TEE. However, with multiple peripheral devices and enclaves on the CPU making up a distributed enclave, the straightforward approach to just individually attest to every component is vulnerable to time-of-check-time-of-use attacks. Several attestations (one for each component of the TEE) must be linked with a guarantee that nothing has changed in the components already attested since the last attestation. Without this guarantee, an attacker could tamper with the configuration of already attested enclaves and thus tricking the remote verifier.

RUNTIME ATTACKS. Remapping attacks are also relevant during runtime, as the OS still manages the memory. Well-timed disconnects or memory remappings could result in leakage of confidential data, e.g., if an attacker remaps a peripheral device and replaces it with a malicious device, the CPU enclave should not share sensitive data with the new device.

7.3 OVERVIEW OF OUR APPROACH

In this section, we provide an overview of our approach PIE and introduce *platform-wide enclaves*. Platform-wide enclaves dynamically extend the TCB of traditional TEEs running on the CPU to the peripheral. Platform-wide enclaves consist of multiple distributed enclaves that run on various hardware components such as the CPU and peripheral as shown in Figure 7.1.

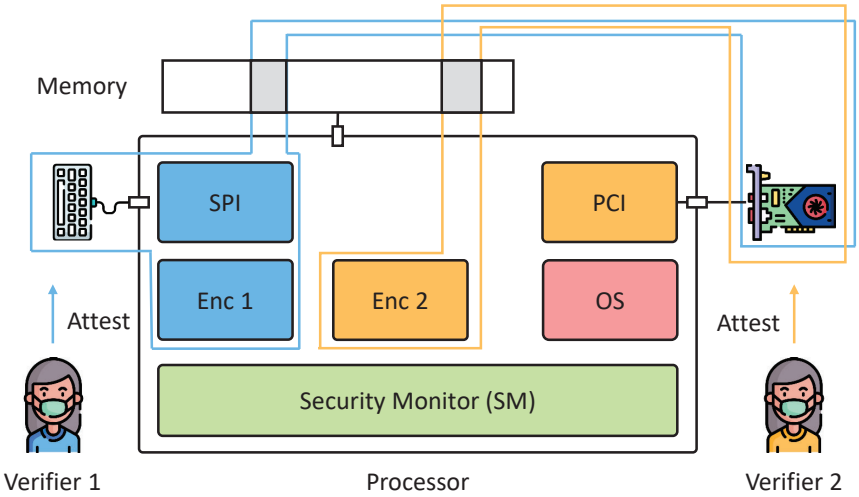


FIGURE 7.1: Two platform-wide enclaves are highlighted by blue and yellow outlines. The blue one consists of `Encl1`, a keyboard that is connected over the memory-mapped SPI bus, and a GPU connected over PCI through DMA. This case represents a trusted path scenario where the user requires both input and output. The second consists of `Encl2` and a GPU connected over PCI through DMA. The second one represents a use case such as executing machine learning workloads in the GPU cores.

Platform-wide enclaves aim to provide similar security properties as traditional enclaves, such as integrity, attestation, and data isolation from other enclaves and the attacker-controlled OS.

7.3.1 *Enclaves within a platform-wide enclave*

A platform-wide enclave consists of multiple enclaves that run on different hardware components and securely communicate with each other. A platform-wide enclave typically contains several interconnected processor-local enclaves and peripheral enclaves. In the following, we describe the two main enclave types that form a platform-wide enclave.

PROCESSOR-LOCAL ENCLAVES. Processor-local enclaves are equivalent to traditional enclaves, and their runtime memory must be isolated from the

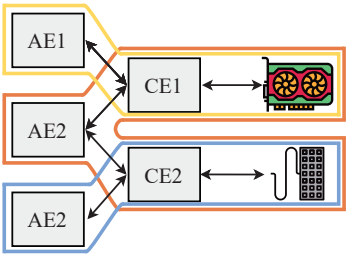


FIGURE 7.2: Three example platform-wide enclaves in PIE's software design with application enclaves (AE), controller enclaves (CE), a GPU, and a keyboard. Note that the red platform-wide enclave is spanning over two external devices and controller enclaves isolate the data from the different application enclaves.

OS and should only be accessible to the enclave itself. To achieve that, we use physical memory protection (PMP) from the RISC-V privilege standard [55] as introduced by Keystone.

We further differentiate two types of processor-local enclaves: application enclaves, and controller enclaves, which encapsulate the application-specific, and driver logic, respectively. As seen in Figure 7.2, AE_1, and CE_1 are the application enclave, and controller enclave in the blue-outlined platform-wide enclave of Figure 7.1. The controller enclave also provides isolation between the application enclaves in a scenario where multiple application enclaves want to access a certain peripheral. Therefore, controller enclave enforces access control on the connected application enclaves in terms of how a certain peripheral can be accessed, e.g., exclusive access to a keyboard or shared concurrent access to a GPU.

ENCLAVES ON PERIPHERAL. Most peripheral run some firmware or even some custom code (e.g., graphic shaders) which has to be included in the TCB of a platform-wide enclave. E.g., the GPU and its firmware in Figure 7.1 is part of the yellow platform-wide enclave. Since a remote verifier also wants to attest to the peripheral, they have to be modified to support attestation. However, we stress that these modifications remain rather small (refer to Section 7.8.2) and usually only involve small changes in the device firmware.

7.3.2 *Communication with peripheral*

First, we need to understand how devices are managed by the platform using a structure called *device tree*.

DEVICE TREE. The device tree is a list that accurately describes the physical memory mappings of a platform. It describes the central processor, i.e., its speed, its ISA, and at what address its cache starts. It also includes the DRAM base address and various other components on the die, such as various internal and external buses. It is usually used by the bootloader and the OS to bootstrap the system. As some peripherals cannot be detected automatically, they must be present in the device tree, as otherwise they will not get recognized by the OS. The device tree is usually burnt into ROM and available to the bootloader and the OS. It can therefore be considered trusted.

SECURE COMMUNICATION. To enable processor-local enclaves and peripheral enclaves to securely communicate, we make the observation that these devices generally communicate over mapped address regions: They either use an address range that is not reflected in DRAM, so-called memory-mapped-input-output registers (MMIO), or a shared DRAM region accessed via direct memory access (DMA). To maximize compatibility with existing drivers and peripheral, we chose not to change this behavior. Instead, we isolate the address regions that are used in this communication. Existing hardware mechanisms like PMP already allow restricting access to a specific address region. Until now, such hardware mechanisms have been predominantly used to restrict memory access, but in our design, they also allow to restrict access to other address regions that are not in the DRAM range[1]. Note that these address regions from peripheral are either i) static, i.e., hardcoded and provided to the SM in the form of a trusted device tree file, or ii) dynamic, i.e., configured at runtime by the SM. In our design, the SM always maintains a complete overview of all such regions and only allows a single enclave to access an address region of a peripheral.

While we made the changes mentioned above to the SM to support peripheral with both MMIO and DMA, they also enable a new way for enclaves to communicate: shared memory. This reflects a major difference

1 E.g., DRAM could occupy the address range `0x8000000 - 0xF0000000`, whereas other peripheral such as UART could reside at `0x4000000 - 0x4001000`.

to traditional TEEs because until now, most traditional enclaves could only communicate through the untrusted OS[2].

7.3.3 *Changes within a platform-wide enclave*

The untrusted OS manages peripheral devices; hence the OS could remap any device or send a reset signal. E.g., a GPU that is handing sensitive data could be shut down by the OS and remapped to a different GPU during runtime. In such a scenario, the enclave should stop sending sensitive data to the GPU until the remote verifier re-attests the new GPU. Hence, the enclave has to react to these external events, i.e., it has to be aware of the platform's state. In traditional TEEs, enclaves are self-sufficient isolated entities and are only dependent on themselves. Therefore, they can only be in two states: running or stopped. Platform-wide enclaves are more complex since they can contain multiple enclaves, all of which could be running, stopped, or even killed. Platform-wide enclaves have to react correctly upon any of these events to keep the data confidential. We achieve this by expanding the enclave lifecycle and adding two new events: connect and disconnect. The asynchronous nature of these events requires a detailed analysis of the security of the entire system, e.g., a well-timed disconnect could lead to data leaks across shared memory regions. We solve this issue by assigning ownership of the shared memory among the enclaves that are accessing that memory. Upon any external events, if one of the participating enclaves dies, the sole ownership is transferred to the remaining one (more details in Section 7.4.3). Therefore, the components in the platform-wide enclave are *platform-aware* since they are aware of any change within their ecosystem.

7.3.4 *Attestation of a platform-wide enclave*

Since a platform-wide enclave consists of multiple distributed enclaves, attestation poses another challenge. Individual attestations to each enclave that make up a platform-wide enclave could be vulnerable to timely manipulations by an attacker to cause time-of-check-to-time-of-use (TOCTOU) issues. To provide a platform-wide attestation, we need to chain attestation reports of all the components of a platform-wide enclave. This includes the attestation report of the enclaves and peripheral firmware. Attestation of the

2 Concurrent work [199] has also shown how shared memory can improve the performance of enclaves significantly.

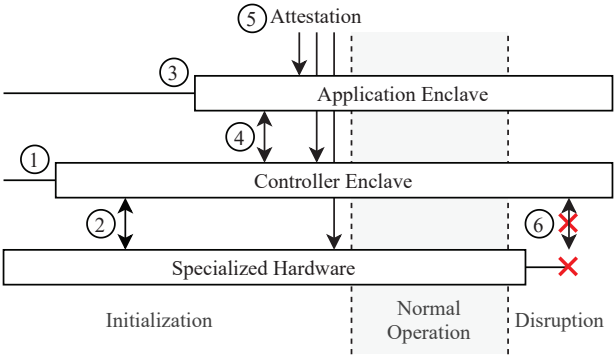


FIGURE 7.3: An example scenario that illustrates interaction between platform-wide enclave components.

peripheral firmware is achieved by signing a challenge message with the key embedded in the peripheral (refer to Section 7.3.1). The attestation of a platform-wide enclave could either be a one-time attestation that results in a huge chain of reports or individual attestations of all entities that can be combined by the verifier. We show that individual attestations provide more flexibility for the verifier and are secure against TOCTOU attacks by adding unique identifiers to enclaves and appending all connected enclaves' IDs to the attestation report.

7.3.5 *Summary of Interactions*

To summarize all interactions between the components of a platform-wide enclave, we present an example scenario in Figure 7.3 with an application enclave, a controller enclave, and a peripheral device. The scenario is as the following:

① The OS creates and configures the controller enclave and hands over control to the SM. The SM then revokes the OS's access permissions to the private memory regions of the controller enclave.

② The OS requests SM to connect controller enclave with the peripheral device. The SM sets up a new shared memory region and enables access only to the controller enclave.

③ Similar to the controller enclave, the OS creates and configures the application enclave, and then, once again, the SM revokes access to the private memory of the enclave.

④ The OS calls the SM to establish a shared memory region between the controller enclave and the application enclave.

⑤ After a remote verifier attests to all enclaves (using the application enclave as the entry point), sensitive data can be transmitted, and the normal operation starts.

⑥ Any disruption, i.e., a disconnection of the peripheral device, leads to an asynchronous disconnect, where the sole ownership of the shared memory between the device and the controller enclave moves to controller enclave (see Section 7.4.3). Moreover, the enclaves may halt execution until re-attested.

7.4 PLATFORM ISOLATION ENVIRONMENT

In this section, we describe Platform Isolation Environment or PIE in detail. PIE is based on the idea of platform-wide enclaves that integrates peripheral devices to the traditional processor-core enclaves while maintaining small hardware and software TCB. First, we discuss the changes needed to incorporate into the peripheral to make them compatible with PIE. Then we introduce a shared memory model that allows enclaves to communicate with each other and peripheral securely. Next, we discuss how the enclave life cycle changes given these modifications and how a remote verifier can get proof of the state of a platform-wide enclave. Finally, we provide a software design for PIE that makes PIE for the software developers easy to adapt.

7.4.1 *Changes to peripheral*

There exists a wide range of peripheral devices that have unique behavior and integrate differently into PIE. In this chapter, we try to cover most devices but stress that some special cases require further analysis. We go from the simplest peripheral device we can imagine, a simple sensor, to one of the most complex, a sophisticated accelerator for a data center. Most other peripheral devices should fall in between these two examples and thus may require modifications between these two extremes.

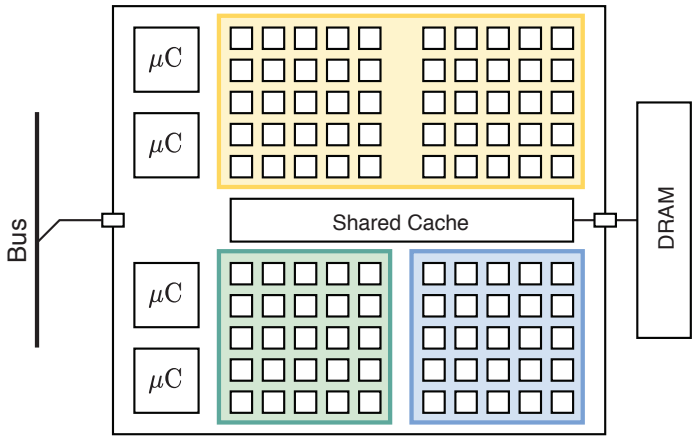


FIGURE 7.4: **Example of a TEE on an accelerator** with multiple enclaves running simultaneously while being isolated from each other. In this example, there are three distinct enclaves running on a subset of the compute units of the accelerator indicated by a yellow, green, and blue overlay.

1. *Sensors and IO peripherals* e.g., a temperature sensor only requires a minimal form of attestation to be integrated into PIE. They must contain some key materials to sign some statements about themselves. This is mandatory for (remote) attestation of a platform-wide enclave that includes an attestation report of such a sensor. Usually, these sensors do not contain any secret data from a processor-local enclave and hence do not need to protect such data.

2. *Accelerators* on the other hand, tend to be very complex and require more extensive modifications. Like the former, they must support attestation, but they may also support isolation for multiple enclaves' secret data. Let us assume data-center applications, where multiple stakeholders want to move multiple compute-intensive tasks from the CPU to the accelerator. The individual tasks' data should remain confidential and isolated, not only on the CPU but also on the accelerator (as seen in Figure 7.4). Thus, such an accelerator requires isolated and attestable domains – in other words – enclaves that run on the peripheral.

7.4.2 *Shared memory*

Shared memory is a common mechanism for software to communicate across multiple cores or DMA regions of a peripheral. In our prototype, shared memory regions are centrally maintained by the CPU to enforce isolation as all peripheral is connected to the CPU.

1. *Shared Memory between Processor-local Enclaves* As mentioned before, processor-local enclaves rely on PMP entries for isolation. We reuse the functionality of PMPs also to protect shared memory regions. Therefore, our proposal does not require any changes to the processor itself, as PMP is already part of the RISC-V standard [55], and thus, it is already part of many processors. The SM, however, requires some modifications. For example, to store the configuration details for every shared memory region in local memory, the SM needs to reconfigure the PMP entries on a context switch similar to stock Keystone. It also must guarantee that at most two entities have access to the same shared buffer at a time. Additionally, the SM flushes the buffers' content when one enclave is destroyed not to leak stale data.

2. *Shared Memory with peripheral* Peripheral are connected to the CPU over buses, which are, in turn, controlled by bus controllers. As mentioned in Section 7.3, peripheral communicate over memory-mapped address ranges either in the form of MMIO registers or DMA memory regions. To isolate these address regions, we rely on existing hardware mechanisms, mainly PMP. Until now, such hardware mechanisms have predominantly been used to restrict memory access, but they can also be used to restrict access to any other address that is not in the DRAM range. Our prototype reuses the concepts from shared memory between processor-local enclaves by assuming the peripheral to be another processor-local enclave. As such, it can directly share an address range with a real processor-local enclave. The SM represents a peripheral internally as a special case of a processor-local enclave that cannot be scheduled or called, but it may share some address regions with other enclaves.

POLLING AND INTERRUPTS. peripheral are synchronized with the processor with either polling or interrupts. Polling requires the CPU to check at a predetermined rate if new data is available from the peripheral, and thus, it can immediately be used in PIE. On the other hand, interrupts are

more complicated as they enable the peripheral to notify the CPU that new data is available with the processor's hardware support. In RISC-V specifically, interrupts can be delegated from the highest privilege mode to lower ones. So, in our prototype of PIE, the SM can delegate individual types of interrupts either to an enclave or to the OS[3]. Therefore, enclaves could also contain interrupt-handlers to, e.g., handle interrupts for a specific peripheral. Note that in our prototype, we only focus on polling.

7.4.3 *Enclave life cycle*

Traditional enclave's life cycle includes three distinct states: idle, running, and paused. E.g., the enclave is first created and started in the idle state. Then the enclave transitions to the running state after a call from a user. Due to a timer interrupt by the OS scheduler, it is paused. It resumed again as soon as the scheduler yields back to the enclave.

ATTACHING PERIPHERAL. Before going into the lifecycle details, it is crucial to understand how peripheral are *attached* to the platform and initialized. There are two types of initialization procedures: statically compiled in the device tree or dynamically mapped by a bus controller. The device tree describes the specific address ranges and model numbers of all statically connected peripheral devices. It is usually stored in on-chip ROM and is provided to the OS by a zero-stage boot-loader, and thus, it can be considered trusted. Dynamically mapped devices are mapped by a bus controller and a driver to a DMA region. In our proposal, the bus controller's driver, which sets up the DMA region, has to be trusted.

CHANGES DURING RUNTIME. In PIE, we introduce two additional life cycle events to describe what happens when a shared memory region is altered. These are *connect* and *disconnect* that are needed due to the asynchronous nature of peripheral as they can prompt a disconnect event at any time.

The asynchronous disconnects are very critical as an enclave could end up continuing to use a memory region that is no longer protected due to a disconnect. Additionally, enclaves might want to provide graceful degradation and should not crash completely upon a disconnect. We solve both

3 In RISC-V external interrupts are handled by the platform interrupt controller (PLIC) and then multiplexed on top of the external interrupt signals to the core. Thus the SM has to contain a driver for the PLIC to figure out which peripheral the interrupt is from.

issues by splitting the disconnect event into an asynchronous disconnect and a synchronous disconnect. We consider both enclaves or peripheral of a shared memory region to have shared ownership over that region. If one of the entities dies, the other entity gains the sole ownership of the memory region. As such, an asynchronous disconnect leads to the sole ownership of a previously shared memory region. In turn, the untrusted OS can issue a synchronous disconnect command to the SM to free the shared memory region and notify the enclave of the disconnect. We mandate that before any connect command, the enclave must first receive a synchronous disconnect. If this was not the case, an attacker could disconnect a benign peripheral and reconnect a malicious one without the enclave noticing.

We illustrate the behavior of a platform-wide enclave in various circumstances using an example scenario. *enclave 1* (E_1) connected to *enclave 2* (E_2). E_2 then is connected to a peripheral (HW). We denote the shared memory spaces as $S_{\{E_1,E_2\}}$, and $S_{\{E_1,HW\}}$that is shared among E_1 & E_2, and E_1 & HW respectively.

1. E_1 *is killed.* In such a situation, the specific shared memory space $S_{\{E_1,E_2\}}$ should be destroyed. To do that, the SM performs an asynchronous disconnect of E_2 for $S_{\{E_1,E_2\}}$ resulting in sole ownership of $S_{\{E_1,E_2\}}$ by E_2. Upon the following synchronous disconnect $S_{\{E_1,E_2\}}$ gets fully destroyed.

   A specific application may require any sensitive data from E_1 that is still on HW to be cleared. In such a scenario, E_2 will tell HW to clear this data on the following synchronous disconnect. Note that how the peripheral handles this call is also dependent on the implementation of that peripheral firmware enclave. For example, a peripheral that handles sensitive user data may decide to terminate the session completely (by zeroing out all the internal states) and destroy the shared memory between the peripheral and E_2.

2. E_2 *is killed.* All shared memory regions associated with E_2 (this includes the shared memory spaces with both E_1 and HW) are immediately modified by the SM during the asynchronous disconnect. They are now solely owned by E_1 and HW, respectively. Zeroing out $S_{\{E_2,HW\}}$ also implicitly notifies HW that E_2 has died, forcing the peripheral to reset.

3. *HW is killed/disconnected* In the asynchronous disconnect, the SM immediately modifies $S_{\{E_2,HW\}}$ to $S_{\{E_2\}}$. At some later point, the OS

must issue a synchronous disconnect, which invalidates $S_{\{E_2\}}$. This also results in the destruction of $S_{\{E_1,E_2\}}$ in case E_1 accesses HW through E_2. From then on E_2 is available to connect to a new HW (after attestation).

7.4.4 *Attestation of a platform-wide enclave*

We extend the existing notion of attestation from processor-local enclaves to platform-wide enclaves that run on multiple components of the platform. Traditionally, attestation ensures the current state of an enclave through a measurement of the code. The standard attestation report of a traditional enclave contains the measurements of both enclaves and the low-level firmware (e.g., the security monitor in RISC-V keystone). Both of which are signed by the platform key (known as the device root key). In contrast, an attestation of a platform-wide enclave must also reflect all included components. A potential attestation mechanism for a platform-wide enclave would be a lengthy report containing all the components' measurements. Contrary to that, we provide the verifier with an option to decide which other enclaves he wants to attest. When the verifier attests a specific component of a platform-wide enclave, a list of identifiers of all the connected components is provided alongside the attestation report. These identifiers are assigned by the SM on the processor and can be used to specify which enclave one wants to attest. A verifier can then chose to attest some or all the connected enclaves from the list of identifiers if he wishes to do so.

ENCLAVE IDENTIFIERS. Upon creation of a new processor-local enclave, SM assigns a unique identifier to it. This identifier uniquely determines the enclaves participating in a specific shared memory region. When the enclave is killed, the identifier may be reused for other enclaves (refer to Section 7.7).

ATTESTATION FLOW. Figure 7.5 depicts an example platform-wide enclave and the sequence of the attestations between its different components.The PIE enclave contains three components *enclave 1* (E_1), *enclave 2* (E_2) and a peripheral firmware. Note that the platform-wide attestation process starts from the verifier who initiates a remote attestation request of E_2. The attestation report of E_2 includes a list of connected enclaves' identifiers, notably E_1. The verifier then executes a series of individual remote attestations of all connected enclaves. Note that both individual

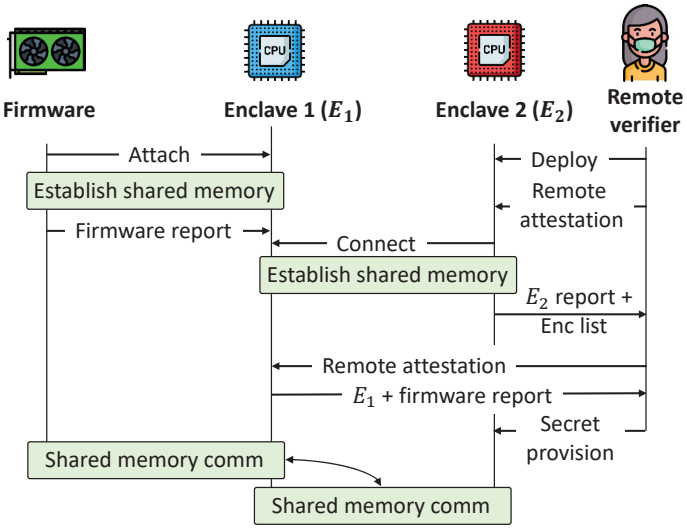


FIGURE 7.5: **Flow of the (remote) platform-wide attestation process** between the remote verifier, E_1, E_2 and the peripheral firmware. At the end of this process, the remote verifier receives the attestation report of E_1, E_2, and peripheral firmware, including their connection parameters such as DMA size, peripheral access control policy, etc.

attestations of E_1 and E_2 include each other's identifier in their list of connected components. Note that both the attestation reports of E_1 and E_2 are signed by the same platform key. This proves to the remote verifier that both the enclaves are running on the same platform.

For peripheral, the attestation mechanism is different. First of all, a peripheral needs to contain some key material and a signed certificate from the manufacturer. This allows a verifier to observe the legitimacy of the peripheral. Secondly, the verifier from Figure 7.5 needs to be able to verify that the peripheral is directly talking to E_1. This is facilitated by the SM, who checks the address regions for MMIO registers. DMA regions can even be established by an untrusted entity such as the OS. However, the attestation report of both the peripheral and E_1 contains the physical memory region that they share.

7.5 PIE SOFTWARE DESIGN

In this section, we introduce PIE's software design which is one possible way for the application, driver, and firmware developers to adapt their software to be compatible with PIE without making a significant changes.

7.5.1 *Software components*

PIE's software design consists of three entities: application enclaves, controller enclaves, and peripheral firmware as shown in Figure 7.2. application enclaves and controller enclaves are processor-local enclaves. peripheral are the components that are connected to the platform over buses. Contrary to a monolithic design where the application and driver is in one big enclave, our modular approach aims to provide high flexibility and increase code reuse.

APPLICATION ENCLAVES. Application enclaves are similar to the traditional enclaves in Intel SGX or Keystone. In such TEEs, the enclaves cannot access peripheral without using the OS as a mediator, as the OS handles all drivers. In PIE, application enclaves also cannot communicate with a peripheral directly. The application enclaves use shared memory to communicate with a controller enclave that is a peripheral-specific enclave containing the driver logic. The rationale of separating the driver from the application logic is two-fold, i) to avoid requiring the developers to ship driver code with their application, and ii) one controller enclave per peripheral allows multiple application enclaves to communicate with that specific peripheral in parallel.

CONTROLLER ENCLAVE. The controller enclave contains the driver that facilitate communication with a peripheral. Note that application enclaves, standard non-enclave applications, and the OS cannot access the peripheral directly. The only way to communicate with a peripheral is through a device-specific controller enclave. Such a design choice isolates the peripheral drivers: one compromised driver does not affect other peripheral. The controller enclave maintains an isolated communication channel over shared memory (e.g., in RISC-V, the PMP entry corresponding to a shared memory ensures that only participating enclaves have access to that shared memory) to application enclaves and the peripheral. To simplify the configuration,

we assume that only one active controller enclave per peripheral exists at a time. However, any controller enclave can be replaced at the user's request.

ISOLATION OF MULTI-APPLICATION ENCLAVE SESSION. In PIE, multiple application enclaves could connect to a single controller enclave to have simultaneous access to a peripheral. In such a scenario, the controller enclave keeps separate states corresponding to each of the application enclaves. Note that this is primarily a functional and then a security requirement as operations in one application enclave could affect the state of computation of another application enclave in case there is no isolation. For some peripheral, the controller enclave may need to reset the state of the peripheral when it switches to a session with a different application enclave (temporal separation). However, for peripheral such as GPU that support multiple isolated workloads in parallel, the state does not have to be reset.

7.5.2 *Platform-wide attestation in the software design*

The platform-wide attestation enables a remote verifier to verify the state of the all platform-wide enclave components. The attestation proceeds as the following:

REMOTE ATTESTATION OF THE APPLICATION ENCLAVE. This is the first step of the platform-wide attestation to ensure that the platform is running the intended version of the application enclave. The application enclave attestation report includes the list of identifiers of the controller enclaves that have shared memory channels with that application enclave.

REMOTE ATTESTATION OF THE CONNECTED CONTROLLER ENCLAVES. The user then executes a series of individual remote attestation for the controller enclaves. The controller enclaves send the attestation report of themselves along with the certificate that is received from the connected peripheral. These reports are signed by the same platform key as of the application enclave attestation report. This proves that the application enclave and the connected controller enclaves are running on the same physical platform. Additionally, the controller enclave also states that the initiating application enclave has a shared memory channel with it.

For each remote verifier, the TCB is constrained to only the specific platform-wide enclave that she is verifying. Hence a remote verifier does

not need to trust any application enclave, controller enclave or peripheral that she is not using/attesting

7.6 DATA CONFIDENTIALITY IN A LOCAL PHYSICAL ATTACKER SETTING

A local physical attacker is what is traditionally considered when developing TEEs [16, 24]. With such an attacker, only the processor package is trusted, alongside some low-level software, called security monitor in Keystone and μCode on Intel SGX. This software is crucial as it incorporates the TEE functionality into RISC-V or Intel processors, respectively. In addition, we also trust the peripheral package since a rogue peripheral obviously breaks any secrecy and isolation requirement. We assume the user to be cooperating and interested in getting a correct measurement from the peripheral. So active conspicuous attacks are out of scope. If, however, the physical world is completely under the control of the attacker, then the attacker can change some properties of the physical world, such as the temperature or GPS data. Obviously, this will lead to a maliciously modified sensor reading and can result in malfunction. However, this attack vector is considered out-of-scope because it usually presents a major hurdle for primitive adversaries (i.e., users). Nevertheless, a sophisticated attacker can take advantage of this and cause tampered sensor readings. Note that this only concerns measurements of the physical world, and external accelerators are not affected by this. By trusting the package, we assume that the local attacker can not physically tamper with the processor or the peripherals. However, all connections between the processor and peripherals are fair game and can be fully controlled by an attacker. E.g., on a commercially available desktop platform, the processor package and peripheral ICs (USB, PCI Express) are trusted. However, we also assume that the attacker can also plug her own compromised peripherals into the platform.

7.6.1 *Cryptographic Engines and Key Management*

PIE relies on memory encryption engine (MEE) and bus encryption engine (BEE) to protect from a local physical attacker. Figure 7.6 shows these cryptographic engine in context to various PIE programming model components.

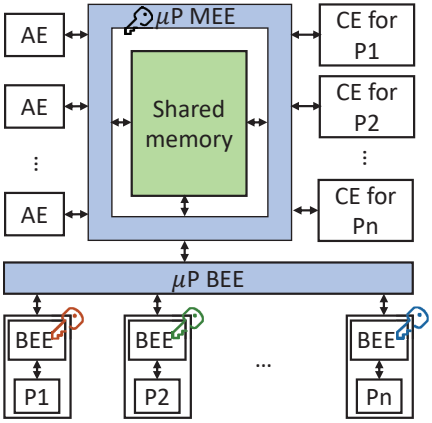


FIGURE 7.6: Different cryptographic engines in relation to different components of PIE. AE, CE and MEE stands for application enclave, controller enclave and memory encryption engine respectively.

MEMORY ENCRYPTION ENGINE (MEE). All the enclaves that are running on the CPU cores (application enclave and controller enclave) and all the shared memory regions are encrypted by the MEE. Due to this, the attacker who has physical access to the memory module can not observe or manipulate the shared memory contains. As only the CPU cores access these memories, only one key is sufficient to encrypt and preserve the integrity of the data in all the memory regions. Note that the CPU keeps the integrity tree of all of the shared memory and enclave memory regions.

BUS ENCRYPTION ENGINE (BEE). MEE is not sufficient in the communication between the CPU core enclaves and peripheral enclaves. In MEE, as mentioned before, the CPU keeps the integrity tree. This is sufficient as only the CPU cores write and read to and from the DRAM. However, when we consider the communication between the CPU core enclaves and the peripheral enclaves, the integrity tree only at the CPU core does not ensure the integrity of the data coming from the peripherals. Due to this, we leverage the bus encryption engine that is present both in the CPU and the peripherals. During the initial attestation process, the CPU and the peripherals derive the session key that is used as the BEE keys. Note that every peripheral derives its own key to communicate with the CPU. This was, an attacker-controlled peripheral can not observe or modify the data on the bus coming from another peripheral.

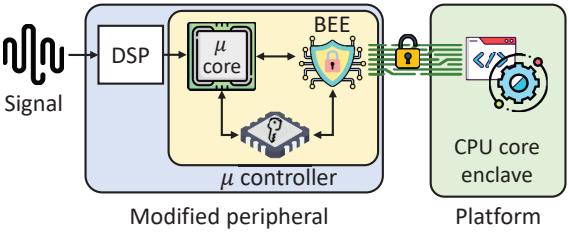


FIGURE 7.7: **An example peripheral with modifications for PIE** that adds the bus encryption engine (BEE) and a keystorage to communicate with the CPU.

7.6.2 *Modified Peripherals*

The peripherals themselves require some changes in order to be part of a platform-wide enclave. Here we describe one potential peripheral configuration and what modifications are necessary. The modifications to the peripherals enhance their functionality by remote attestation and bus encryption for communication. However, the complexity of the peripherals and their modifications might change depending on the use case. An example peripheral could be an analog sensor (e.g., a thermometer) that contains a small digital signal processor (DSP) that converts the analog reading to a digital signal. Such devices usually come with limited memory to store the configuration parameters/ look-up tables or some past sensor readings. The only required hardware changes are the necessary secure storage for keys and certificates provisioned by the manufacturer. All cryptographic algorithms can be performed by the microcontroller since the performance requirements are rather low. An example of such a modified peripheral with a microcontroller, BEE, and key-storage is depicted in Figure 7.7.

7.7 SECURITY ANALYSIS

In this section, we perform an informal security analysis of PIE. We split the security analysis into three separate parts. First, we show how isolation from a malicious OS and other malicious peripheral is achieved. Then we analyze the attacker-controlled life cycle events of platform-wide enclaves, and finally, we discuss the security of platform-wide attestation.

7.7.1 *Isolation*

First, we will talk about the isolation property. The isolation property is in the context of malicious OS, other enclaves, malicious peripherals, and various side channels.

MALICIOUS OS. In PIE, the address regions that are used by platform-wide enclaves are protected using PMP entries [55]. Recall that in stock keystone [24], PMP is based on the physical memory range and only allows the specific enclave to access its private memory. On top of this, we use additional PMP entries to protect shared memory regions. Note that only the highest privilege level, i.e., the SM, can modify PMP entries. During a context switch, the SM re-configures all PMP entries such that the correct memory ranges are available again. The SM has a complete overview of all enclaves and shared memory regions and sets up all PMP entries on its own. The processor will throw an access fault exception upon any memory access into protected memory regions. The hardware page table walker also must behave according to the configured PMP rules. Therefore, miss-configured page tables cannot be used to leak any data from protected memory ranges.

The SM enforces a shared memory region to be strictly shared between two entities (e.g., a processor-local enclave and a peripheral device). The SM also verifies that no overlap exists between the memory ranges similar to the stock keystone.

ROGUE DMA REQUESTS. Malicious peripherals can try to access protected memory through rogue DMA requests. Mechanisms to restrict DMA requests already exist in other architectures, e.g., AMD IOMMU [200], Intel VT-d [201], and ARM SMMU [202]. These mechanisms process every DMA request and verify its validity according to some access policy. Any memory access attempt that does not fit the access policy is blocked. Currently, the RISC-V standard does not contain a mechanism to limit such DMA requests. However, an input-output variant of a PMP called IOPMP [203] is an upcoming proposal in RISC-V. IOPMP enforces the configured PMP rules for non-RISC-V peripherals. Since our current prototype does not have any peripheral interface open to DMA requests, we do not need such protection. However, platforms that support DMA could implement mechanisms like IOPMP.

MALICIOUS APPLICATION OR CONTROLLER ENCLAVES. The attacker-controlled OS can spawn malicious application enclaves and controller enclaves. Users remotely attest before providing any secret to the application enclave. During the platform-wide attestation, the user checks the attestation report of both the application enclave and controller enclave and aborts if they do not match with the intended enclave measurements. The platform-wide attestation also reveals any misconfiguration of communication links by an attacker. Note that this only verifies the static configuration of communication links. Upon any change to this setup, the external verifier might need to re-attest (refer to Section 7.7.2).

We require the controller enclave to provide isolation between multiple connected application enclaves (refer to Section 7.5.1). Hence an attacker-controlled application enclave cannot access the confidential data of other application enclaves in the same controller enclaves.

Vulnerabilities within any of these enclaves could break the isolation guarantees of the data in that specific enclave. However, such an attack remains contained in the compromised enclave and cannot spread to connected enclaves. E.g., if a vulnerability in a controller enclave is found, only the data within that enclave is revealed. Any data that does not pass through the compromised controller enclave remains confidential. In this way, we provide defense-in-depth and reduce the potential impact of vulnerabilities.

MALICIOUS PERIPHERAL. If an attacker manages to compromise the exact device that is used by an enclave, then any data on the peripheral is forfeit. However, any data not passed to the malicious device remains confidential.

We stress that certain manipulations of specific peripherals are always possible for an attacker. Consider, for example, a temperate sensor. Any local physical attacker can increase the real-world temperature and thus manipulate the sensor reading. However, as we describe in our attacker model in Section 7.2.1, the physical attacker is out of the scope of this chapter. Note that this only applies to sensors; accelerators cannot get tampered with in this manner.

SIDE CHANNEL ATTACKS. While we do not evaluate any defenses against side-channel attacks, we discuss potential side-channel attacks against our proposal and how they could potentially get mitigated. Many parts of PIE remain the same as in traditional TEEs where side channels have been widely investigated [43, 204, 205], however, we note that PIE

creates some new side channels that may not be present in traditional TEEs such as bus contention.

1. *Traditional side-channel attacks against TEEs.* Microarchitectural side channels in traditional TEEs leverage shared resources such as the cache [43], branch predictor [46], and memory translation [47]. There exist several defenses against such attacks. Spatial partitioning of the cache in the form of cache coloring can fully defend against all cache-based side-channel attacks [18, 206, 207]. Similarly, other proposals have called for cache randomization [204, 208]. Processor features such as transactional memory have also been shown to mitigate cache attacks with low overhead [205]. To the best of our knowledge, all of these proposals can be applied to PIE due to the similar internal structure to traditional TEEs.

2. *Side channel attacks within peripheral.* Peripheral contain shared resources such as caches, and thus are equally vulnerable as the processors [209, 210, 211]. However, mitigating these attacks is an orthogonal problem.

3. *Bus contention* The introduction of peripherals into TEEs also implicates the bus as a new shared resource. An attacker could measure the throughput of her connection over the bus and observe any contention on the bus leading to less throughput. Bus contention, however, only exposes the access patterns of the peripherals. In extreme cases, the timing of bus contention could leak data, e.g., one side of the branch performs bus accesses while the other does not. However, in normal cases, the data between the peripherals and their corresponding processor-local enclaves usually remains inaccessible to the attacker.

7.7.2 *Lifecycle events*

As described in Section 7.4.3, there are two additional events for platform-wide enclaves in PIE. `Connect` is used to connect two entities over a shared buffer. `Disconnect` facilitates a disconnect between the two enclaves. The `disconnect` is split into a synchronous and a asynchronous event. The asynchronous disconnect only occurs when one of the entities unexpectedly dies and results in the transfer of the sole ownership of the memory region to the remaining enclave. This enclave can then try to continue its execution. However, it will realize that the other entity has died as it does not react to any activity on the shared memory region. At a later point, the untrusted

OS can issue a synchronous disconnect to notify the enclave and free the shared memory officially. Note that the SM mandates a synchronous disconnect before another `connect` command. Due to this architecture, a stale shared buffer will never be made accessible to any untrusted entity until a synchronous disconnect occurs, during which the enclave will officially get notified. The separate handling of synchronous and asynchronous disconnect events enforces protection for any secret data during an enclave's entire life cycle.

7.7.3 *Attestation*

When the remote verifier attests to an enclave, he receives identifiers of all the connected enclaves. The SM generates these identifiers and makes sure that no two running enclaves share the same identifier. Hence, an enclave could be assigned with an identifier that belonged to an enclave in the past. Of course, strictly increasing identifiers implemented with monotonic counters could be used for the identifier, but such a solution needs non-volatile storage on the CPU that is expensive.

Now assume that the attacker kills an enclave and lunch another enclave with a *different* binary (defined as `code`), but with the exact same identifier. . I.e., she can kill enclave A and launch A' ($\texttt{code}(A') \neq \texttt{code}(A)$) with the same identifier ($\texttt{ID}(A) = \texttt{ID}(A')$). However, when a remote verifier attests A', he sees that the measurements mismatch as $\texttt{code}(A') \neq \texttt{code}(A)$ and rejects it.

Lets assume a more complex scenario with two pairs of enclaves: A, B and A', B', where $\mathsf{code}(A') \neq \mathsf{code}(A)$ but $\mathsf{code}(B') = \mathsf{code}(B)$. A remote verifier attests to an enclave A that is connected to B and establishes a shared secret with A. Before the verifier attests to B, she kills B. The attacker then spawns a new enclave B' where $\texttt{ID}(B) = \texttt{ID}(B')$. The remote verifier will then attest to B' and find that the code measurement looks fine. However, we stress that B' cannot be connected to A because then A would need to receive a synchronous disconnect and would need to be re-attested (due to the configuration of A). Now the attacker kills A and replace with A' (where $\texttt{ID}(A) = \texttt{ID}(A')$) and connect A' and B'. The verifier then sees that B' has the correct measurement and is connected to the identifier of A (as $\texttt{ID}(A) = \texttt{ID}(A')$). However, the verifier will want to provide its data to A using the shared secret they have established in the previous attestation. Obviously, this cannot succeed as the new enclave A' cannot know the secret.

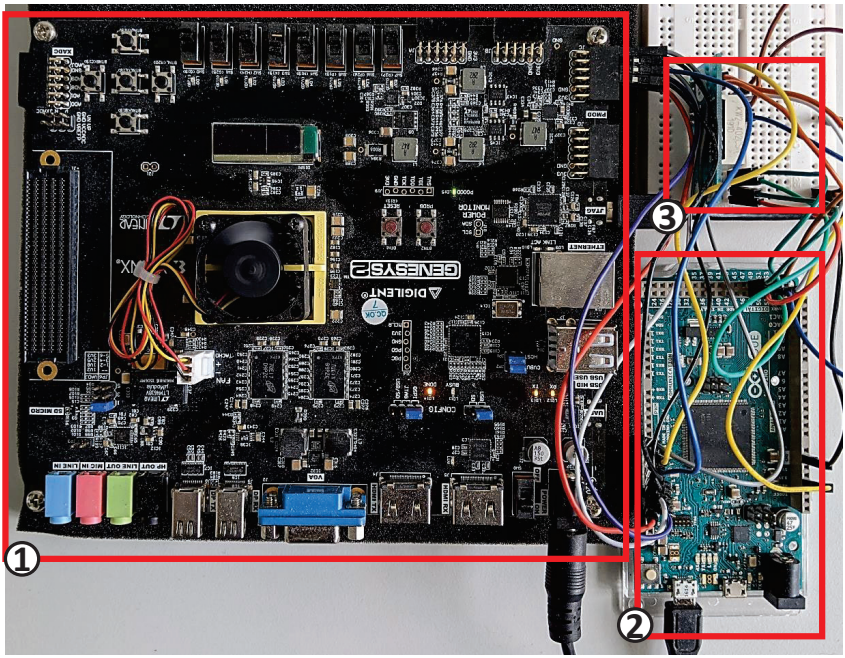


FIGURE 7.8: The figure shows different components of our prototype: ① Digilent Genesys 2 FPGA board, ② Arduino Due as the peripheral simulator, and ③ a seven-segment display unit as an example peripheral.

7.8 IMPLEMENTATION AND EVALUATION

In this section, we describe our prototype of PIE and its evaluation.

7.8.1 *Implementation*

FPGA PROTOTYPE. We implemented an end-to-end prototype of PIE that is based on the Keystone enclave framework [24]. Figure 7.8 shows a case study on a platform that consists of an FPGA emulating the central processor connected to several Arduino boards that emulate peripheral.

FPGA PLATFORM. We base our system on the Ariane core [25], an open-source RISC-V 64-bit core that supports commodity OS such as Linux. It is an RC64GC 6-stage application class core that has been taped out multiple

times and can operate up to 1.5 GHz. We run this core on a Digilent Genesys 2 FPGA board (① in Figure 7.8).

We added PMP capability to the core that originally does not support PMP using 160 lines of SystemVerilog. The PMP unit is formally verified against a handwritten specification with yosys [212]. Two of these units are inserted into the memory management unit (MMU) and are responsible for checking data accesses, and instruction fetches. An additional unit is placed in the hardware page table walker to check page table accesses. Our implementation has a configurable number of PMP entries up to the maximum number of 16 mandated by the standard [55]. Our modifications have been contributed to the Ariane project and are open source [213]. Note that PMP is part of the RISC-V privilege standard and, as such, is already available on many other cores [51, 214].

PROCESSOR-LOCAL ENCLAVES. We modified the SM to be able to connect two enclaves or an enclave and peripheral. Specifically, we added three new interfaces to the SM called `connect`, `sync_disconnect`, and `async_disconnect`. These interfaces can be used to set up shared regions between two enclaves or peripheral specified by their identifier. We also modified Keystone's attestation procedure to include a list of identifiers for all connected enclaves. Our modifications only amount to 390 additional or modified lines of code. The SM consists of around 2000 lines of code excluding SHA3 and ed25519 implementations that contribute around 4000 additional lines of code.

In Keystone, every enclave runs on top of a minimal runtime that handles syscalls and manages virtual memory. Hence, its code is critical and part of the TCB. For our prototype, we added support to dynamically map shared memory regions into the virtual address space of an enclave. We modified 213 LoC out of a total of 3600 LoC for Keystone's runtime.

On the untrusted OS side, there are many components to make it easier to create and run enclaves, such as an SDK and a kernel driver. These components also required numerous changes. However, they are not trusted and, as such, do not increase the TCB.

7.8.2 *Two Use Case Scenarios*

IO PERIPHERALS AND SENSORS. In our prototype, we emulated a number of simple peripheral (e.g., keyboard, mice, simple sensors, etc.) on the Arduino Due microcontroller prototyping board (② in Figure 7.8) using

Arduino HID library. The Due's GPIO pins are connected to the FPGA's PMOD pins over two pairs of 8 wires for bi-directional data. We modify the I^2C protocol to communicate data between the Due and the FPGA. The physical limitations of the PMOD pins restrict the channel's frequency to 8 MHz, yielding 1 MB/s bandwidth. In the real world, the physical interfaces between the peripheral and the platform could be diverse such as USB, PCI-E, etc. As a concrete example, we implemented a keyboard with the Arduino board and wrote a simple keyboard driver that interprets the GPIO signal from the Arduino. Additionally, we use a PMOD interface-based seven-segment display unit as an output peripheral (③ in Figure 7.8). The driver contains around 50 LoC and is incorporated into our example controller enclave. Additionally, we use the `USBHost` library that can emulate a number of USB peripheral devices on the Arduino. We use the Arduino cryptographic library for signing the challenge messages from the controller enclave during the local attestation. The Due uses 128-bit AES (CTR mode) for encryption, HMAC_SHA256 for message authentication, Curve25519 for key exchange, and SHA3 for the hash function. We use `DueFlashStorage` library to implement the NVM flash that contains the key materials for the peripheral attestation. Our prototype implementation is approximately 2.5K lines of code.

ACCELERATOR. We conduct another case study to show how complex peripheral such as a GPU-scale accelerator [198] can be extended to support PIE. The accelerator is a 4096-core RISC-V platform that has comparable performance to current state-of-the-art machine learning accelerators. It is organized in clusters, each with 8 individual single-stage RISC-V cores [215], each of which is accompanied by a double-precision floating-point unit capable of two double precision and four single-precision flops per cycle. To hide memory latency, all clusters have access to a scratchpad memory and a large L2 data cache.

To provide multi-tenant isolation on the accelerator, we introduce a shared PMP unit with 4 entries into every cluster. The PMP entries can only be configured by one out of eight cores, but the access policies will be enforced on all of them. With this additional hardware support, we were able to implement a small firmware that configures the PMP entries according to the specifications from the host and then runs a task in user mode. Upon a context switch, the scratchpad memory that was in use by the previous task must be flushed, and the PMP entries must be reconfigured.

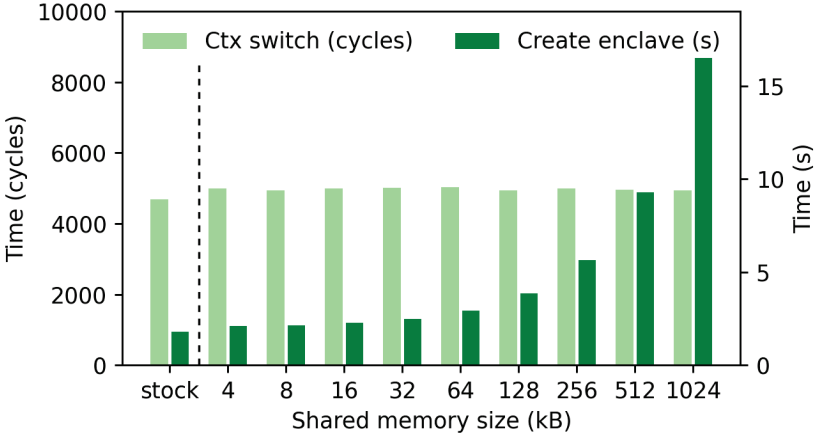


FIGURE 7.9: Context switch performance for varying sizes of a shared memory region compared to stock Keystone performance on the left (equivalent to no shared memory).

The firmware consists of 143 lines of assembly and 73 lines of C code. This implementation and verification take around 3 weeks.

7.8.3 *Performance*

We list PIE's performance in the following categories:

1. *Performance of enclave communication* Since PIE supports shared memory to communicate, its communication speed is the same as what the memory bus provides. This is much faster compared to traditional TEEs, where enclaves communicate through the OS, requiring extra encryption steps. Concurrent work also demonstrates the performance gains that can be extracted from enclave to enclave communication using shared memory [199].

2. *Context switches* Context switches are critical for any system and determine its responsiveness and a part of its performance. We performed experiments for various sizes of shared memory regions and gathered various context switch latencies in Figure 7.9. We also measured the time of enclave creation which is mostly dominated by copying all the

PMP entries	logic	caches	total
0	472k GE	686k GE	1141k GE
8	497k GE	686k GE	1164k GE
16	531k GE	686k GE	1197k GE

TABLE 7.1: **Hardware size over of PIE over the Ariane core**Size of the default configuration of the Ariane core in gate equivalents (GE), synthesized in 22nm at 1GHz with varying number of PMP entries.

enclave data from the untrusted OS to the protected memory region and thus is expected to be linear in terms of shared memory size.

These measurements highlight that the context switches are independent of the shared memory size. The absolute context switch time increases from 4730 for stock Keystone to 4950 for PIE.

3. *PMP overhead* We measure the hardware overhead of PMP units in terms of the logic, the caches, and the total amount in NAND2 gate equivalents within the processor pipeline for 0, 8, and 16 PMP entries and present them in Table 7.1. We instantiate the Ariane core [25] with the default configuration: including the floating-point unit, 32KiB L1 data cache, 16KiB L1 instruction cache, branch history table of size 64, and a 16-entry branch target buffer. We synthesized this instantiation of the core in a 22nm technology at 1GHz.

4. *IO peripherals and sensors* The communication overhead between the platform and the peripheral device emulated by the Arduino due is very small. At the time of initialization, the peripheral and the platform exchanges handshake messages to perform local attestation. The initial handshake message is 60 bytes. Every message size of our modified I^2C protocol is 32 bytes. The combined latency introduced by signing averages around 60 μs.

5. *Accelerator* Our modification of the accelerator cores increases size by around 15% and slows down from 750MHz to 666MHz due to the impact of the PMP access checks on the critical path. Note that this may not reflect the general case but rather reflects an upper bound. The change in the area of a single core complex (core, FPU, and an integer subsystem) can be found in Table 7.2. In total, the area of the

Area [μm^2]	#PMP entries		Overhead
	0	4	
Core	5.7	6.7	15.5%
FPU	39.2	37.9	-3.3%
IPU	8.6	8.5	-1.4%
Total	53.5	53.2	-0.7%

TABLE 7.2: **The area rundown of our modifications to the accelerator.** The area rundown of our modifications to the accelerator with 4 and 0 PMP entries respectively. Synthesized in 22nm with 750MHz clock for 0 entries and 666MHz with 4 entries. Area given in μm^2.

entire accelerator only increased by around 0.6%, with most of the area being occupied by the floating-point units.

7.9 DISCUSSION

7.9.1 *Limitation of the Number of PMP Entries*

The number of PMP entries in the RISC-V privilege specification is limited to 16 (proposals for 64 entries are in discussion) due to the overhead of such entries on the CPU size. However, this limited the number of enclaves and shared memory regions that may coexist on a system. With one shared memory region per enclave, at most $(N-2)/2$ enclaves can exist at a time (16 entries support 7 enclaves). However, we stress that the isolation of enclaves can also be achieved using the memory management unit (MMU) in a similar fashion as Intel SGX [16] or Sanctum [18][4]. MMU-based isolation can also easily be extended to shared memory ranges and remove any limitation on the maximum number of enclaves.

4 There are efforts towards the hypervisor extension in RISC-V that would allow MMU based isolation without non-standard modifications, but as these are not ratified, they are hard to evaluate.

7.9.2 *Enhanced Privacy Mode*

Using an end-to-end secure channel between application enclave and peripheral, we can enable an enhanced privacy mode into a platform-wide enclave. After the remote attestation of the platform-wide enclave, the remote verifier receives the attestation report of the individual components, including their public keys. Using these keys, the application enclave and the peripheral can establish a TLS session using the controller enclave as an untrusted transport layer. The developers need to enable this feature in the peripheral firmware. Moreover, cryptographic operations executed in software may result in lower performance. This enhanced privacy mode can work alongside the regular operation (trusted controller enclave).

7.10 RELATED WORK

In this section, we discuss some of the most relevant works to PIE. We also discuss the main differences with PIE to these works.

7.10.1 *TEE-based solutions*

There exist several solutions for integrating external devices to widely deployed TEEs: Intel SGX, and ARM TrustZone.

SGXIO. SGXIO is a proposal by Weiser et al. [36] that builds on top of Intel SGX, intending to allow SGX to interact with input-output devices. They achieve that by introducing a trusted hypervisor that allows enclaves to access virtualized peripherals. Similar to our approach, SGXIO also considers a remote attacker. However, SGXIO is rather static in nature, i.e., all the peripherals have to be set up at boot time. After the system setup phase, no changes are allowed (connect new peripherals, etc.). It is not clear how enclaves are created and get access to a peripheral while preserving the confidentiality of previous enclaves that used said peripheral. Essentially, SGXIO allows for a static extension of the hardware TCB.

GRAVITON AND SYSTEMS BASED ON IT. Graviton [216] is a TEE that runs on an accelerator such as a graphics card. It can provide isolation between the data of multiple stakeholders that run tasks on the GPU concurrently. It also provides remote attestation of an enclave on the accelerator. Graviton was evaluated on a modern graphics card and shows that the

predominant overhead stems from encryption of the communication to the processor. However, they also demonstrate that the overhead of around 20% is tolerable. Graviton would fit very well within a PIE as it is an excellent example of an enclave on a peripheral. It provides isolation for secret data and attestation reports. In addition, it shows that even some of the most powerful accelerators can be extended with a local TEE. Visor [211] is a system built upon Graviton [216] that proposes a hybrid TEE that spans over both CPU and GPU. Visor is aimed towards privacy-preserving video analytics where the computation pipeline is shared between the CPU (non-CNN workloads) and the GPU (CNN workloads) to increase efficiency. Visor addresses micro-architectural-based side-channel attacks where a local physical attacker can use the data-dependent memory access patterns (e.g., branch-prediction, cache-timing, or controlled page fault attacks) to reveal elements on the video analysis (e.g., leaks pixel patterns). The communication between the CPU and GPU enclaves is encrypted. Additionally, Visor ensures that the traffic pattern between the CPU and GPU enclaves is independent of the video content.

HETEE. HETEE [62] is another proposal to extend TEEs to accelerators (specifically GPUs) without requiring changes to existing CPUs/GPUs. HETEE focuses on data center applications and proposes an extra hardware box per rack that is protected from physical attacks. This box also contains all accelerators, which it then connects to compute servers in the same rack. Each enclave then runs on a dedicated compute server and a connected accelerator. In essence, the HETEE box provides secure routing of accelerators to dedicated compute servers. In contrast to HETEE, we aim to be able to execute multiple composite enclaves on the same compute server.

ARM TRUSTZONE AND SYSTEMS BASED ON IT. TrustZone is a system TEE provided by ARM for their system-on-chips (SoC) [17]. TrustZone applications run on top of a secure OS that is trusted and isolated from the standard operating system (also known as the rich OS). Isolation between the two worlds is achieved by an extra bit on the bus. However, there is no isolation between different TrustZone applications. Due to this limitation, mobile phone manufacturers usually only allow TrustZone applications that are signed by them. TrustZone only provides the lower level isolation property between the rich OS and the secure OS. Everything else, i.e., isolation between TrustZone applications, remote attestation, etc., has to be added to the secure OS [217]. There have been many proposals that try to

improve on the capabilities of TrustZone [218, 219]. Sanctuary [218] enables user-space TrustZone enclaves. Sanctuary achieves isolation by running enclaves in their own address space in the normal world. However, Sanctuary is very similar to Intel SGX, and thus, it does not extend to peripherals. There exist proposals that enable additional security properties such as a trusted path by enabling direct pairing of peripherals (e.g., the touchscreen) to the TrustZone application. Such proposals include TruZ-Droid [140], TrustUI [167], SeCloak [220], VButton [139]. All of these solutions do not provide any form of peripheral attestation similar to PIE. They also do not consider other peripherals or how to dynamically allow an enclave to access a peripheral previously used by another enclave. Moreover, as mentioned earlier, the absence of isolation between the TrustZone applications makes the proposal mentioned above weak in inter enclave isolation guarantee.

7.10.2 *Other isolation methods*

Minimal hypervisors or operating systems [127, 221] can also achieve isolation, and some are even formally verified [127]. Usually, such hypervisors do not include attestation, but the cost of adding that should be very low. A PIE could also be based on a microkernel such as seL4. One would have to add an interface for the malicious OS running in a virtual machine to interact with enclaves that run directly on top of seL4, similar to other pure hypervisor-based isolation systems. It might even be possible to formally prove such modifications to provide an even stronger assurance of isolation. Similar to other pure hypervisor-based isolation system include Virtual Ghost [129], Overshadow [128], InkTag [130], TrustVisor [131], Splitting Interface [132], Terra [169] etc. The hypervisor is also in charge of the scheduling, resulting in a significantly bigger TCB than PIE. Moreover, in none of the hypervisor-based proposals, platform awareness and platform-wide attestation are considered. Isolation is the sole objective of these proposals.

7.10.3 *Bump in the wire-based solutions*

Fidelius [14], ProtectIOn (refer to Chapter 5), IntegriScreen (refer to Chapter 4), FPGA-based overlays [32], IntegriKey (refer to Chapter 3) are some of the trusted path solutions that use external trusted hardware devices as intermediaries between the platform and IO devices. These external devices create a trusted path between a remote user and the peripheral and

enable the user to exchange sensitive data securely with the peripheral in the presence of an attacker-controlled OS. Such solutions provide a loose notion of platform awareness and are focused on IO devices. Platform-wide attestation and strong isolation guarantee are out-of-scope of such proposals.

7.11 CONCLUSION

We introduce PIE, a secure platform design with a configurable hardware and software TCB. PIE allows to integrate peripheral into TEEs, something that before was not possible without violating TEEs' attacker model. PIE provides two new security properties: platform-wide attestation and platform awareness. The former expands on the traditional notion of the attestation to provide a complete view of the platform's state, and platform awareness provides mechanisms to the enclave to cope with the platform's change, such as the disconnection of a peripheral. We present a prototype based on RISC-V Keystone, which shows that PIE is feasible and only adds 600 LoC to the TCB.

Part VI

CONCLUSIONS

8

CLOSING REMARKS

I dream my painting and I paint my dream.
— Vincent Van Gogh

This chapter summarizes the work presented in this thesis and highlights the main findings and results. Besides, we remark on the lessons learned and provide directions for future work.

8.1 SUMMARY OF THE CONTRIBUTIONS

This thesis expands our current understanding of the trusted path problem in modern computing systems and how it could be strengthened with moderns trusted execution environments or TEEs. We begin this thesis by illustrating the problem of the trusted path in modern (x86) platforms that run complex software and hardware. These entities span over millions of lines of code which results in a large attack surface. Attackers can exploit the vulnerabilities in the software and hardware to break the integrity and confidentiality property of sensitive IO data. TEEs have the potential to be a candidate for trusted path realization as TEEs minimize the software (in some scenario hardware, e.g., Intel SGX assumes DRM to be untrusted) attack surfaces. TEEs do this by excluding the OS and hypervisors from the software TCB. However, issues such as reliance on the OS to connect with external devices and trust on first use (ToFU) property of the remote attestation make TEEs not ideal for the trusted path. Moreover, the TEEs' reliance on the OS or hypervisor to reach an external device result in static hardware and software TCB. This issue is undesirable, specifically in the context of modern data centers that host multiple VMs accessing different devices for their computations. Such issues amplify when we consider the trusted path applications can span over multiple special-purpose hardware devices outside the CPU cores, i.e., the modern disaggregated computing architecture in the data centers.

Given this problem space, in summary, we make progress on two fundamental aspects of the trusted path and trusted computing in two following ways. *First*, we understand the fundamental security properties that are critical for the integrity and confidentiality of a trust path. Along this line, we

investigate how trusted small-TCB external devices can work in conjunction with modern computing platforms (with or without TEEs) to strengthen the trusted path. And *second*, how to make small changes in the hardware and software architecture of modern TEEs to extend the security guarantees of TEEs from CPU cores to the external hardware devices.

8.1.1 *External Trusted Devices*

First, this thesis shows two systems IntegriKey (Chapter 3) and IntegriScreen (Chapter 4) based on two new mechanisms: input signing and visual supervision, respectively, to provide user input integrity (and output integrity in the later work). Both of the proposals rely on a trusted device to ensure the integrity of the IO data. IntegriKey uses a low-TCB embedded device where IntegriScreen uses an off-the-shelf smartphone. In IntegriKey, we also show a new form of IO manipulation attack that we call label swapping attack that swaps the label of input fields that may take overlapping input data to mislead the user. Both of these works show potential for using an external verifier device to work in conjunction with an attacker-controlled host to achieve trusted path properties. However, over time, we found that, like many other existing trusted path solutions, IntegriKey and IntegriScreen are not fully secure.

To further investigate these security issues with not only IntegriKey and IntegriScreen, but also existing proposals in trusted path, we look for fundamental security properties that are critical to ensure the integrity and confidentiality of use IO data. In ProtectIOn (Chapter 5), we propose these fundamental security properties, including a semi-formal proof. In ProtectIOn, we also provide a prototype based on a trusted embedded device that we call IOHub that works as the intermediary between the user's IO peripherals and the attacker-controlled host. ProtectIOn provides a sandboxed UI that interacts with the user that is rendered and operated by the IOHub. ProtectIOn shows how a trusted path can be enforced by an external hardware device that significantly reduces the TCB of a host platform running commodity software stack - OS, hypervisor, etc.

The thesis then continues with the idea of the external trusted devices and presents ProximiTEE (Chapter 6) that uses a small USB connected device that we call ProximiKey to identify relay attacks. In ProximiTEE, we address relay attack, a long-standing problem in TEE where a malicious OS can divert all the traffic to a compromised platform that the attacker physically owns. The rerouting problem is well-known in the context of TPM. However,

in this thesis, we are the first ones to understand the implication of the attack on Intel SGX remote attestation. We identify that almost all flavors of remote attestation rely on the trust on first-use (ToFU) property that makes them vulnerable to the relay attack. ProximiKey uses the distance bounding protocol to distinguish a physically connected platform (over USB) from a relayed (over a network) platform. Our rigorous experimental result shows the feasibility of such an attack where the attacker can execute the relay attack with an average round trip time of only 186 μs. ProximiTEE distance bounding mechanism can distinguish such an attack with a very high probability. In ProximiTEE, we also propose a second mechanism based on boot-time initialization and boot-time attestation to protect against an attacker with access to at least one leaked Intel SGX attestation key. We implement a prototype that shows the boot-time initialization and attestation are feasible.

8.1.2 *Modification in Software/Hardware Architecture of the Platform*

Next, this thesis presents PIE (Chapter 7), which looks into the trusted path problem from the perspective of disaggregated computing architecture. PIE proposes a new software architecture that is based on the traditional software architecture of application-driver-device. This new software architecture is based on an idea that we call a platform-wide enclave where enclaves could be CPU-bound enclaves or firmware enclaves running on the devices. CPU bounds enclaves are divided into application enclave and controller enclave based on their functionality (application logic or driver logic). The enclaves inside a platform-wide enclave communicate with each other over shared memory regions isolated from the OS and other enclaves using RISC-V's physical memory isolation or PMP. The platform-wide enclaves are fully attestable, and the attestation report includes the individual enclave's measurement, including the access control policy of the device and the setup configuration. Platform-wide enclaves have platform-awareness, i.e., they can detect changes in their components (connect or disconnect) and take appropriate action. We implement a PIE prototype that is based on a RISC-V processor with keystone TEE running on an FPGA platform and an Arduino microcontroller that emulates peripherals. We introduce small changes to the RISC-V firmware (known as the security monitor or SM), and we show that PIE is feasible.

8.2 FUTURE WORK

In this section, we provide future work in the field of the trusted path and trusted computing.

8.2.1 *Trusted Path in Smart Manufacturing Systems*

The smart manufacturing system is one of the cornerstones of the fourth industrial revolution (Industry 4.0). Unlike traditional manufacturing systems, where the entire process is geared towards producing a single product, the smart manufacturing system is demand-driven and flexible. This is possible due to the adoption of 3-D printers, injection molding, industrial robots, and re-configurable manufacturing templates. Additionally, compared to traditional manufacturing, smart manufacturing systems are often highly automated. But despite such high automation, humans are involved intricately in the process. Such a high level of automation makes the smart manufacturing system an obvious target for the attacker to introduce fault to the manufactured items, damage equipment, harm human operators, impact the downtime of the systems, and many more. Such distributed system is one natural use-case scenario of PIE.

8.2.2 *Industrial HMI*

Industrial human-machine interfaces (HMIs) act as the bridge between the human operator and the smart manufacturing plants. Typical HMIs are either commodity PC hardware or dedicated touch interfaces that provide rich UIs. An attacker-controlled OS or the host can easily represent wrong information on the HMI to confuse the human operators. Such attacks have already been demonstrated in StruxNet malware attacks. Moreover, the UI on the industrial HMIs is very different from the standard web-based UI we see. Such HMIs rely on color codes to relay information about a process or a safety hazard. The UI sequence is also critical as, during an emergency, the operators often need to execute a specific series of actions. There hasn't been sufficient research on the security of such HMI devices. Moreover, despite having some recommended standards on the HMIs, most well-deployed HMI devices are proprietary and do not adhere to those standards, making them ideal for exploits.

8.2.3 *Authentication devices*

Authentication is another important aspect of a trusted path. Recently, we have seen a sharp rise in second-factor authentication devices such as FIDO and other hardware-backed password managers. A preliminary investigation shows that many of such password managers emulate themselves as a keyboard to enter a password to the host system, in plain text, despite claiming to provide security against a malicious OS. Few of the managers require the installation of the browser plugin to provide auto-fill functionality in the browser. However, this requires trusting the browser that significantly increases the TCB of such approaches. Through study and security analysis of such devices are necessary to understand the security properties that they provide.

8.3 FINAL REMARKS

This thesis looks at the increasing software and hardware complexity of modern computing platforms and how it affects one of the most fundamental aspects of computation - human-machine interaction. Despite a large number of existing research proposals and deployed systems, the problem of trusted paths largely remains unsolved. As shown in this thesis, a system-building way of designing the trusted path often leads to insecure systems. This we learn not only from our own system but the existing systems. This thesis provides an understanding of the trusted path problem by breaking it down into a set of fundamental security properties. Moreover, this thesis explores the idea of hardware-assisted security where a small trusted embedded device can reinforce trusted path properties. The thesis also shows that the trivial way of integrating the trusted path into TEEs such as Intel SGX does not guarantee security as an underlying assumption (trust on the first use (ToFU) of remote attestation) can introduce relay attack. This thesis also explores the idea of introducing small changes into software hardware architecture to distribute the TEE properties from the CPU cores to heterogeneous peripherals. Hence, we can conclude that there is an increasing need to redesigning many of the existing platforms to reduce the trusted computing base significantly. This thesis also opens up avenues for future research directions of how to design & evaluate trusted path systems and how to enable new trusted path features in TEEs.

BIBLIOGRAPHY

[1] Brad A. Myers. "A Brief History of Human-Computer Interaction Technology". In: *Interactions* 5.2 (1998), 4454.

[2] Usabilla. *A Short History of Computer User Interface Design*. https://medium.theuxblog.com/a-short-history-of-computer-user-interface-design-29a916e5c2f5. 2017.

[3] *X-600M | Web Enabled I/O Controller*. https://www.controlbyweb.com/x600m. 2021.

[4] *InPen Smart Insulin Delivery System | by Companion Medical*. https://www.companionmedical.com/.

[5] Alex Berry. *WannaCry Malware Profile*. https://www.fireeye.com/blog/threat-research/2017/05/wannacry-malware-profile.html. 2017.

[6] Matthew Field. "WannaCry cyber attack cost the NHS £92m as 19,000 appointments cancelled". en-GB. In: *The Telegraph* (2018).

[7] Microsoft. *Microsoft XP - Microsoft Lifecycle | Microsoft Docs*. https://docs.microsoft.com/en-us/lifecycle/products/windows-xp. 2021.

[8] Kim Parker, Juliana Menasce Horowitz, and Rachel Minkin. *How the Coronavirus Outbreak Has - and Hasnt - Changed the Way Americans Work*. https://www.pewresearch.org/social-trends/2020/12/09/how-the-coronavirus-outbreak-has-and-hasnt-changed-the-way-americans-work/. 2020.

[9] Zongwei Zhou, Virgil D Gligor, James Newsome, and Jonathan M McCune. "Building verifiable trusted path on commodity x86 computers". In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012.

[10] Atanas Filyanov, Jonathan M McCuney, Ahmad-Reza Sadeghiz, and Marcel Winandy. "Uni-directional trusted path: Transaction confirmation on just one device". In: *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. IEEE. 2011.

[11] Thomas Weigold and Alain Hiltgen. "Secure confirmation of sensitive transaction data in modern Internet banking services". In: *2011 World Congress on Internet Security (WorldCIS-2011)*. IEEE. 2011.

[12] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. "Bump in the Ether: A Framework for Securing Sensitive User Input". In: *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*. 2006.

[13] Mohammad Mannan and Paul C Van Oorschot. "Using a personal device to strengthen password authentication from an untrusted computer". In: *FC*. 2007.

[14] Saba Eskandarian, Jonathan Cogan, Sawyer Birnbaum, Peh Chang Wei Brandon, Dillon Franke, Forest Fraser, Gaspar Garcia, Eric Gong, Hung T Nguyen, Taresh K Sethi, et al. "Fidelius: Protecting user secrets from compromised browsers". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, 264.

[15] Bonnie Brinton Anderson, Anthony Vance, C Brock Kirwan, Jeffrey L Jenkins, and David Eargle. "From warning to wallpaper: Why the brain habituates to security warnings and what can be done about it". In: *Journal of Management Information Systems* (2016).

[16] Victor Costan and Srinivas Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* (2016).

[17] Johannes Winter. "Trusted computing building blocks for embedded linux-based ARM trustzone platforms". In: *Proceedings of the 3rd ACM workshop on Scalable trusted computing*. 2008, 21.

[18] Victor Costan, Ilia Lebedev, and Srinivas Devadas. "Sanctum: Minimal hardware extensions for strong software isolation". In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, 857.

[19] Stephen Checkoway and Hovav Shacham. "Iago attacks: why the system call API is a bad untrusted RPC interface". In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), 253.

[20] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. "Memory deduplication as a threat to the guest OS". In: *Proceedings of the Fourth European Workshop on System Security*. 2011, 1.

[21] Bryan Parno. "Bootstrapping Trust in a Trusted Platform". In: *HotSec'08*. 2008.

[22] Amazon. *What is Amazon EC2 - Amazon Elastic Compute Cloud*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. 2021.

[23] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. "Rack-scale disaggregated cloud data centers: The dReDBox project vision". In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2016, 690.

[24] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. "Keystone: An open framework for architecting trusted execution environments". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, 1.

[25] Florian Zaruba and Luca Benini. "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), 2629.

[26] United States Government Department of Defense. *Trusted Computer System Evaluation Criteria ["Orange Book"]*. https://csrc.nist.gov/csrc/media/publications/conference-paper/1998/10/08/proceedings-of-the-21st-nissc-1998/documents/early-cs-papers/dod85.pdf.

[27] Jeremy Epstein and Jeffrey Picciotto. "Trusting X: Issues in building Trusted X window systems-or-what's not trusted about X". In: *Proceedings of the 14th Annual National Computer Security Conference*. 1991.

[28] Thomas Weigold and Alain Hiltgen. "Secure confirmation of sensitive transaction data in modern Internet banking services". In: *WorldCIS 2011*.

[29] Stuart E Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer. "The emperor's new security indicators". In: *S&P'07*.

[30] Adrienne Porter Felt, Robert W. Reeder, Alex Ainslie, Helen Harris, Max Walker, Christopher Thompson, Mustafa Embre Acer, Elisabeth Morant, and Sunny Consolvo. "Rethinking Connection Security Indicators". In: *SOUPS 2016)*.

[31] Adrienne Porter Felt, Robert W. Reeder, Hazim Almuhimedi, and Sunny Consolvo. "Experimenting At Scale With Google Chrome's SSL Warning". In: *CHI*. 2014.

[32] Anthony Brandon and Michael Trimarchi. "Trusted display and input using screen overlays". In: *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*. IEEE. 2017.

[33] Yeongjin Jang, Simon P Chung, Bryan D Payne, and Wenke Lee. "Gyrus: A Framework for User-Intent Monitoring of Text-based Networked Applications." In: *NDSS*. 2014.

[34] Ramakrishna Gummadi, Hari Balakrishnan, Petros Maniatis, and Sylvia Ratnasamy. "Not-a-Bot: Improving Service Availability in the Face of Botnet Attacks". In: *NSDI 2009*.

[35] Hongliang Liang, Mingyu Li, Yixiu Chen, Lin Jiang, Zhuosi Xie, and Tianqi Yang. "Establishing trusted i/o paths for sgx client systems with aurora". In: *IEEE Transactions on Information Forensics and Security* 15 (2019), 1589.

[36] Samuel Weiser and Mario Werner. "SGXIO: generic trusted I/O path for Intel SGX". In: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM. 2017.

[37] Intel. *Intel SGX homepage*. https://software.intel.com/en-us/sgx.

[38] Travis Peters, Reshma Lal, Srikanth Varadarajan, Pradeep Pappachan, and David Kotz. "BASTION-SGX: Bluetooth and Architectural Support for Trusted I/O on SGX". In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '18. ACM, 2018.

[39] SSLab SGX 101. *Attestation Primitives*. http://www.sgx101.com/portfolio/attestation_primitives/.

[40] Intel. *Innovative technology for CPU based Attestation and Sealing*. https://software.intel.com/content/www/us/en/develop/articles/innovative-technology-for-cpu-based-attestation-and-sealing.html. 2013.

[41] Simon Johnson and Intel. *Intel SGX: EPID Provisioning and Attestation Services*. https://software.intel.com/content/www/us/en/develop/download/intel-sgx-intel-epid-provisioning-and-attestation-services.html. 2017.

[42] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. "Supporting third party attestation for Intel® SGX with Intel® data center attestation primitives". In: *White paper* (2018).

[43] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software grand exposure: SGX cache attacks are practical". In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. 2017.

[44] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. "Cache attacks on Intel SGX". In: *Proceedings of the 10th European Workshop on Systems Security*. 2017, 1.

[45] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "Cachezoom: How SGX amplifies the power of cache attacks". In: *CHES'17*. 2017.

[46] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. "Inferring fine-grained control flow inside SGX enclaves with branch shadowing". In: *26th USENIX security symposium (USENIX security 17)*. 2017, 557.

[47] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-channel attacks: Deterministic side channels for untrusted operating systems". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, 640.

[48] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. "Spectre attacks: Exploiting speculative execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, 1.

[49] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. "Meltdown: Reading kernel memory from user space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, 973.

[50] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, 991.

[51] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. *The rocket chip generator*. Tech. rep. University of California, Berkeley, 2016.

[52] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini. "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), 2700.

[53] Krste Asanovic, David A Patterson, and Christopher Celio. *The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor*. Tech. rep. University of California at Berkeley Berkeley United States, 2015.

[54] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. "TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V." In: *NDSS*. 2019.

[55] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanović. "The RISC-V instruction set manual volume II: Privileged architecture". In: *EECS Department, University of California, Berkeley* (2019).

[56] Linux Torvalds et al. "Linux kernel source tree". In: *Git Respository* (2020).

[57] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the art of virtualization". In: *ACM SIGOPS operating systems review* 37.5 (2003), 164.

[58] Ilia Lebedev, Kyle Hogan, Jules Drean, David Kohlbrenner, Dayeol Lee, Krste Asanović, Dawn Song, and Srinivas Devadas. "Sanctorum: A lightweight security monitor for secure enclaves". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, 1142.

[59] Nikolaos Alachiotis, Andreas Andronikakis, Orion Papadakis, Dimitris Theodoropoulos, Dionisios Pnevmatikatos, Dimitris Syrivelis, Andrea Reale, Kostas Katrinis, George Zervas, Vaibhawa Mishra, et al. "dReDBox: A Disaggregated Architectural Perspective for Data Centers". In: *Hardware Accelerators in Data Centers*. Springer, 2019, 35.

[60] DELL. *PoweEdge rack server | Dell USA*. https://www.dell.com/en-us/work/shop/servers-storage-networking/sf/poweredge-rack-servers. 2021.

[61] Lenovo. *Flex system compute node for blade server | lenovo US*. https://www.lenovo.com/us/en/data-center/servers/flex-blade-servers/compute-nodes/c/compute-nodes. 2021.

[62] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, et al. "Enabling Rack-scale Confidential Computing using Heterogeneous Trusted Execution Environment". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, 1450.

[63] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. "Disaggregated memory for expansion and sharing in blade servers". In: *ACM SIGARCH computer architecture news* 37.3 (2009), 267.

[64] Hugo Meyer, Jose Carlos Sancho, Josue V Quiroga, Ferad Zyulkyarov, Damian Roca, and Mario Nemirovsky. "Disaggregated computing. an evaluation of current trends for datacentres". In: *Procedia Computer Science* 108 (2017), 685.

[65] Microsoft. *High Performance Compouting - HPC | Microsoft Azure*. https://azure.microsoft.com/en-us/solutions/high-performance-computing/. 2021.

[66] NetApp. *Disaggregated HCI for Hybrid Cloud | NetApp*. https://www.netapp.com/virtual-desktop-infrastructure/netapp-hci/. 2021.

[67] Nvidia. *What is DPU? | Nvidia Blog*. https://blogs.nvidia.com/blog/2020/05/20/whats-a-dpu-data-processing-unit/. 2020.

[68] Nvidia. *Nvidia BlueField 2 dataset*. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf. 2021.

[69] Fungible. *DPU Platform - Fungible*. https://www.fungible.com/product/dpu-platform/. 2021.

[70] B. Mahato, T. Maity, and J. Antony. "Embedded Web PLC: A New Advances in Industrial Control and Automation". In: *2015 Second International Conference on Advances in Computing and Communication Engineering*.

[71] *Sitraffic smartGuard*. https://www.siemens.com/global/en/home/products/mobility/road-solutions/traffic-management/strategic-management-and-coordination/centrals/smartguard.html.

[72] Siemens. *SIMATIC S7-1200 | SIMATIC Controllers | Siemens Global Website*. https://new.siemens.com/us/en/products/automation/systems/industrial/plc/s7-1200.html.

[73] Schneider Electric. *Modicon Momentum | Schneider Electric USA*. https://www.schneider-electric.us/en/product-range/535-modicon-momentum.

[74] BitGo. *Making Digital Currencies Usable for Business*. https://www.bitgo.com/.

[75] Tim Dierks. "The transport layer security (TLS) protocol version 1.2". In: (2008).

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

[77] Z. Zhou, M. Yu, and V. D. Gligor. "Dancing with Giants: Wimpy Kernels for On-Demand Isolated I/O". In: *S&P 2014*.

[78] *BitcoinWallet.com*. https://bitcoinwallet.com/. 2020.

[79] *Coinbase - Buy/Sell Digital Currency*. https://www.coinbase.com/. 2020.

[80] Coin. *Crypto Wallet - Highly Secured Wallet, Buy Bitcoin | CoinSpace*. https://coin.space/. 2020.

[81] *Bitcoin Wallet - Store and invest in crypto*. https://www.blockchain.com/wallet. 2020.

[82] Claude Fachkha, Elias Bou-Harb, Anastasis Keliris, Nasir Memon, and Mustaque Ahamad. "Internet-scale probing of CPS: Inference, characterization and orchestration analysis". In: *NDSS*. 2017.

[83] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, Nagendra Modadugu, et al. "The Ghost in the Browser: Analysis of Web-based Malware." In: *HotBots* (2007).

[84] Timothy Dougan and Kevin Curran. "Man in the browser attacks". In: *International Journal of Ambient Computing and Intelligence (IJACI)* (2012).

[85] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne Burleson. "Trojan Side-Channels: Lightweight Hardware Trojans through Side-Channel Engineering". In: *CHES 2009*. Ed. by Christophe Clavier and Kris Gaj.

[86] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester. "A2: Analog Malicious Hardware". In: *S& P 2016*.

[87] Keiko Hashizume, David G Rosado, Eduardo Fernández-Medina, and Eduardo B Fernandez. "An analysis of security issues for cloud computing". In: *Journal of internet services and applications* (2013).

[88] Diego Perez-Botero, Jakub Szefer, and Ruby B Lee. "Characterizing hypervisor vulnerabilities in cloud computing servers". In: *Proceedings of the 2013 international workshop on Security in cloud computing*.

[89] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. "Flicker: An execution infrastructure for TCB minimization". In: *ACM SIGOPS Operating Systems Review*. 2008.

[90] *WebUSB API*. https://wicg.github.io/webusb/.

[91] *Access USB Devices on the Web | Web | Google Developers*. https://developers.google.com/web/updates/2016/03/access-usb-devices-on-the-web.

[92] *I2C*. https://learn.sparkfun.com/tutorials/i2c. Accessed on 27.04.2018. 2018.

[93] *dk.brics.automaton - finite-state automata and regular expression for Java*. http://www.brics.dk/automaton/.

[94] *Home Assistant Demo*. https://home-assistant.io/demo/.

[95] Kai Salomaa and Sheng Yu. "NFA to DFA transformation for finite languages". In: *Automata Implementation: First International Workshop on Implementing Automata, WIA '96 London, Ontario, Canada, August 29–31, 1996 Revised Papers*. Ed. by Darrell Raymond, Derick Wood, and Sheng Yu. 1997.

[96] *Web Bluetooth*. https://webbluetoothcg.github.io/web-bluetooth/.

[97] S. Kiljan, H. Vranken, and M. Van Eekelen. "What You Enter Is What You Sign: Input Integrity in an Online Banking Environment". In: *2014 Workshop on Socio-Technical Aspects in Security and Trust*.

[98] Weidong Cui, R. H. Katz, and Wai tian Tan. "Design and implementation of an extrusion-based break-in detector for personal computers". In: *ACSAC'05*.

[99] DavidPhelan. *Tim Cook on the importance of coding and Augmented Reality*. Accessed August 2018. 2017. URL: https://www.independent.co.uk/life-style/gadgets-and-tech/features/apple-tim-cook-boss-brexit-uk-theresa-may-number-10-interview-ustwo-a7574086.html.

[100] Microsoft. *HoloLens 2-Mixed reality is ready for business*. Accessed June 2019. 2019. URL: https://www.microsoft.com/en-us/hololens.

[101] Sven Fleck and Wolfgang Straßer. "Smart camera based monitoring system and its application to assisted living". In: *Proceedings of the IEEE* 96.10 (2008), 1698.

[102] Lenovo. *Lenovo Smart Assistant with Google Home*. Accessed September 2019. 2018. URL: https://www.lenovo.com/gb/en/smart-display/.

[103] Johanna Wald, Keisuke Tateno, Jürgen Sturm, Nassir Navab, and Federico Tombari. "Real-Time Fully Incremental Scene Understanding on Mobile Platforms". In: *IEEE Robotics and Automation Letters* 3.4 (2018), 3402.

[104] Alex Perala. *Computer Vision Takes Spotlight in Latest Google Smartphones*. Accessed August 2019. 2018. URL: https://mobileidworld.com/computer-vision-google-smartphones-910111/.

[105] Konrad Rieck, Tammo Krueger, and Andreas Dewald. "Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks". In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC '10. Austin, Texas, USA, 2010.

[106] Sruthi Bandhakavi, Samuel T King, Parthasarathy Madhusudan, and Marianne Winslett. "VEX: Vetting Browser Extensions for Security Vulnerabilities." In: *USENIX Security Symposium*. Vol. 10. 2010, 339.

[107] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. "Protecting Browsers from Extension Vulnerabilities". In: *Network and Distributed System Security Symposium*. NDSS '10. 2010.

[108] Forbes. *Over A Million Coders Targeted By Chrome Extension Hack*. https://www.forbes.com/sites/leemathews/2017/08/03/over-a-million-coders-targeted-by-chrome-extension-hack. Accessed June 2018.

[109] Microsoft. *Security Update Guide*. https://msrc.microsoft.com/update-guide/. Online; accessed December 2020.

[110] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. "Digtool: A virtualization-based framework for detecting kernel vulnerabilities". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, 149.

[111] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. "Understanding linux malware". In: *2018 IEEE Symposium on Security and Privacy (SP)*. SP '18. 2018, 161.

[112] Digital Defense Inc. *Zeus Trojan - What It Is and How to Prevent it*. https://www.digitaldefense.com/blog/zeus-trojan-what-it-is-how-to-prevent-it-digital-defense. Online; accessed December 2020.

[113] The Washington Post. *NSA officials worried about the day its potent hacking tool would get loose. Then it did.* https://cyber-peace.org/wp-content/uploads/2017/05/NSA-officials-worried-about-the-day-its-potent-hacking-tool-would-get-loose.-Then-it-did.pdf. Online; accessed December 2020.

[114] Aritra Dhar, Der-Yeuan Yu, Kari Kostiainen, and Srdjan Capkun. *IntegriKey: End-to-End Integrity Protection of User Input*. Cryptology ePrint Archive, Report 2017/1245. https://eprint.iacr.org/2017/1245. 2017.

[115] Mohammed Korayem, Robert Templeman, Dennis Chen, David Crandall, and Apu Kapadia. "Enhancing Lifelogging Privacy by Detecting Screens". In: *ACM CHI*. 2016.

[116] Luyang Liu, Hongyu Li, and Marco Gruteser. "Edge Assisted Real-Time Object Detection for Mobile Augmented Reality". In: *The 25th Annual International Conference on Mobile Computing and Networking*. MobiCom '19. Los Cabos, Mexico, 2019.

[117] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'15. Montreal, Canada, 2015, 91–99.

[118] R. Huang, J. Pedoeem, and C. Chen. "YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers". In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, 2503.

[119] Xiang Zhang, Stephan Fronz, and Nassir Navab. "Visual marker detection and decoding in AR systems: A comparative study". In: *IEEE ISMAR*. 2002.

[120] *Android Protected Confirmation: Taking transaction security to the next level*. 2018.

[121] Google. *Text Recognition API Overview*. https://developers.google.com/vision/android/text-overview. Accessed June 2018. 2017.

[122] OpenCV. *Open Source Computer Vision Library*. https://opencv.org/. Accessed June 2018. 2018.

[123] Apache. *Apache Tomcat*. https://tomcat.apache.org/. Accessed July 2018. 2019.

[124] Selenium. *Selenium WebDriver: A browser automation framework and ecosystem*. https://github.com/SeleniumHQ/selenium. Accessed July 2018. 2018.

[125] Anna Pereira, David L Lee, Harini Sadeeshkumar, Charles Laroche, Dan Odell, and David Rempel. "The effect of keyboard key spacing on typing speed, error, usability, and biomechanics: Part 1". In: *Human factors* 55.3 (2013), 557.

[126] Kyle Krafka, Aditya Khosla, Petr Kellnhofer, Harini Kannan, Suchendra Bhandarkar, Wojciech Matusik, and Antonio Torralba. "Eye tracking for everyone". In: *IEEE CVPR*. 2016.

[127] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. "seL4: Formal verification of an OS kernel". In: *ACM SOSP*. 2009.

[128] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems". In: *ACM SIGOPS Operating Systems Review* 42.2 (2008), 2.

[129] John Criswell, Nathan Dautenhahn, and Vikram Adve. "Virtual ghost: Protecting applications from hostile operating systems". In: *ACM SIGARCH Computer Architecture News* (2014).

[130] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. "Inktag: Secure applications on an untrusted operating system". In: *ACM SIGARCH Computer Architecture News*. ACM. 2013.

[131] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. "TrustVisor: Efficient TCB reduction and attestation". In: *IEEE S&P*. 2010.

[132] Richard Ta-Min, Lionel Litty, and David Lie. "Splitting interfaces: Making trust between applications and operating systems configurable". In: *USENIX OSDI*. 2006.

[133] Jisoo Yang and Kang G Shin. "Using hypervisor to provide data secrecy for user applications on a per-page basis". In: *ACM SIGPLAN/SIGOPS VEE*. 2008.

[134] *Innovative CoDeSys automation - I/O modules - Overdigit.com*. https://web-plc.com/. 2017.

[135] *DirectLOGIC 205 Series | KOYO ELECTRONICS INDUSTRIES CO., LTD.* https://www.koyoele.co.jp/en/product/plc/directlogic-205.

[136] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution". In: *27th USENIX Security Symposium USENIX Security 18)*. 2018.

[137] Aritra Dhar, Ivan Puddu, Kari Kostianen, and Srdjan Čapkun. "Proximitee: Hardened SGX Attestation by Proximity Verification". In: *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy (CODASPY '20)*. 2020.

[138] *Android Protected Confirmation: Taking transaction security to the next level*. https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html. 2018.

[139] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. "VButton: Practical Attestation of User-driven Operations in Mobile Apps". In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM. 2018.

[140] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. "TruZ-Droid: Integrating TrustZone with mobile operating system". In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM. 2018, 14.

[141] Miao Yu, Virgil D Gligor, and Zongwei Zhou. "Trusted display on untrusted commodity platforms". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, 989.

[142] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. "You've been warned: an empirical study of the effectiveness of web browser phishing warnings". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2008, 1065.

[143] Jennifer Sobey, Robert Biddle, Paul C Van Oorschot, and Andrew S Patrick. "Exploring user reactions to new browser cues for extended validation certificates". In: *European Symposium on Research in Computer Security*. Springer. 2008.

[144] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. "Robust physical-world attacks on deep learning models". In: *arXiv preprint arXiv:1707.08945* (2017).

[145] Chawin Sitawarin, Arjun Nitin Bhagoji, Arsalan Mosenia, Prateek Mittal, and Mung Chiang. "Rogue signs: Deceiving traffic sign recognition with malicious ads and logos". In: *arXiv preprint arXiv:1801.02780* (2018).

[146] *User Interface Security and the Visibility API*. https://www.w3.org/TR/UISecurity/.

[147] Lin-Shung Huang, Alexander Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. "Clickjacking: Attacks and Defenses." In: *USENIX security symposium*. 2012.

[148] J. Lee, L. Bauer, and M. L. Mazurek. "The Effectiveness of Security Images in Internet Banking". In: *IEEE Internet Computing* (2015).

[149] Claudio Marforio, Ramya Jayaram Masti, Claudio Soriente, Kari Kostiainen, and Srdjan Čapkun. "Evaluation of Personalized Security Indicators As an Anti-Phishing Mechanism for Smartphone Applications". In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. CHI '16. New York, NY, USA: ACM, 2016.

[150] *BadUSB - On accessories that turn evil*. https://srlabs.de/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf. 2014.

[151] Admin. *B101 HDMI to CSI-2 Bridge (15 pin FPC)*. https://auvidea.eu/b101-hdmi-to-csi-2-bridge-15-pin-fpc/. 2016.

[152] *HTML: The Markup Language (an HTML language reference)*. https://www.w3.org/TR/2012/WD-html-markup-20121025/. 2012.

[153] *OpenCV:Template Matching*. https://docs.opencv.org/master/d4/dc6/tutorial_py_template_matching.html.

[154] *picamera- Picamea 1.13 documentation*. https://picamera.readthedocs.io/en/release-1.13/.

[155] Simon Blake-Wilson and Alfred Menezes. "Authenticated Diffe-Hellman key agreement protocols". In: *International Workshop on Selected Areas in Cryptography*. Springer. 1998, 339.

[156] Chromium. *chromium/chromium*. https://github.com/chromium/chromium. 2019.

[157] Mozilla. *mozilla/gecko-dev*. https://github.com/mozilla/gecko-dev. 2019.

[158] *V8 JavaScript Engine*. https://chromium.googlesource.com/v8/v8.git.

[159] *Getting SpiderMonkey source code*. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Getting_SpiderMonkey_source_code.

[160] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. "Shadowcrypt: Encrypted web applications for everyone". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, 1028.

[161] Zishuang Eileen Ye, Sean Smith, and Denise Anthony. "Trusted paths for browsers". In: *ACM Transactions on Information and System Security (TISSEC)* (2005).

[162] Ardalan Amiri Sani. "SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications." In: *MobiSys*. 2017.

[163] Ahmed M Azab, Peng Ning, and Xiaolan Zhang. "Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms". In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011.

[164] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. "Trustotp: Transforming smartphones into secure one-time password tokens". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015.

[165] Dongtao Liu and Landon P Cox. "VeriUI: Attested login for mobile devices". In: *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. ACM. 2014, 7.

[166] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. "Adattester: Secure online mobile advertisement attestation using trustzone". In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM. 2015.

[167] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. "Building trusted path on untrusted device drivers for mobile devices". In: *Proceedings of 5th Asia-Pacific Workshop on Systems*. ACM. 2014, 8.

[168] Amit Vasudevan, Jonathan McCune, James Newsome, Adrian Perrig, and Leendert Van Doorn. "CARMA: A hardware tamper-resistant isolated execution environment on commodity x86 platforms". In: *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM. 2012.

[169] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. "Terra: A virtual machine-based platform for trusted computing". In: *ACM SIGOPS Operating Systems Review*. ACM. 2003.

[170] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Adi Shamir, and Eran Tromer. "Physical key extraction attacks on PCs". In: *Communications of the ACM* 59.6 (2016), 70.

[171] Zhenghong Wang and Ruby B Lee. "Covert and side channels due to processor architecture". In: *ACSAC'06*. 2006.

[172] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. "Electromagnetic analysis: Concrete results". In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2001, 251.

[173] Adi Shamir and Eran Tromer. "Acoustic cryptanalysis". In: *presentation available from http://www. wisdom. weizmann. ac. il/ tromer* (2004).

[174] Stefan Brands and David Chaum. "Distance-Bounding Protocols". In: *EUROCRYPT '93*. 1993.

[175] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. "Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks". In: (2018).

[176] Ashay Rane, Calvin Lin, and Mohit Tiwari. "Raccoon: Closing digital side-channels through obfuscated execution". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, 431.

[177] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. "ZeroTrace: Oblivious Memory Primitives from Intel SGX." In: *NDSS*. 2018.

[178] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. "Obfuscuro: A commodity obfuscation engine on intel sgx". In: *Network and Distributed System Security Symposium*. 2019.

[179] Dan Wendlandt, David G Andersen, and Adrian Perrig. "Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing." In: *USENIX Annual Technical Conference*. Vol. 8. 2008, 321.

[180] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. "ROTE: Rollback protection for trusted execution". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, 1289.

[181] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. "VC3: Trustworthy data analytics in the cloud using SGX". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, 38.

[182] Cypress. *EZ-USB FX3^{TM} SuperSpeed USB 3.0 peripheral controller*. https://www.cypress.com/products/ez-usb-fx3-superspeed-usb-30-peripheral-controller.

[183] ARM Limited. *SSL Library mbed TLS / PolarSSL*. https://tls.mbed.org/.

[184] rweather. *GitHub - rweather/arduinolibs: Arduino Cryptography Library*. https://github.com/rweather/arduinolibs. 2020.

[185] Ofir Weisse, Valeria Bertacco, and Todd Austin. "Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves". In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), 81.

[186] Sachin Agarwal. *Public Cloud Inter-region Network Latency as Heatmaps*. https://medium.com/@sachinkagarwal/public-cloud-inter-region-network-latency-as-heat-maps-134e22a5ff19. 2018.

[187] Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Peter Wyckoff, and D K Panda. "Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics". In: *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. 2003, 58.

[188] Inc. Algo-Logic Systems. *Low Latency PCIe Solutions for FPGA*. https://www.algo-logic.com/sites/default/files/PCIe.pdf. 2019.

[189] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-privilege-boundary data sampling". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, 753.

[190] *Tiny Core Linux, Micro Core Linux, 12MB Linux GUI Desktop, Live, Frugal, Extendable*. https://distro.ibiblio.org/tinycorelinux/. 2018.

[191] Arvind Seshadri, Mark Luk, and Adrian Perrig. "SAKE: Software attestation for key establishment in sensor networks". In: *International Conference on Distributed Computing in Sensor Systems*. Springer. 2008, 372.

[192] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. "SCUBA: Secure code update by attestation in sensor networks". In: *Proceedings of the 5th ACM workshop on Wireless security*. 2006, 85.

[193] Ahmad-Reza Sadeghi and Christian Stüble. "Property-based attestation for computing platforms: caring about properties, not mechanisms". In: *Proceedings of the 2004 workshop on New security paradigms*. 2004, 67.

[194] Zhangkai Zhang, Xuhua Ding, Gene Tsudik, Jinhua Cui, and Zhoujun Li. "Presence attestation: The missing link in dynamic trust bootstrapping". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, 89.

[195] Russell A Fink, Alan T Sherman, Alexander O Mitchell, and David C Challener. "Catching the cuckoo: Verifying tpm proximity using a quote timing side-channel". In: *International Conference on Trust and Trustworthy Computing*. Springer. 2011, 294.

[196] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John Stratton, Alexey Titov, Ke Wang, Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. "SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance". In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Ed. by Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond. Springer International Publishing, 2015, 46.

[197] Arm Ltd. *Learn the Architecture: TrustZone for AArch64*. https://developer.arm.com/architectures/learn-the-architecture/trustzone-for-aarch64/trustzone-in-the-processor. 2021.

[198] Florian Zaruba, Fabian Schuiki, and Luca Benini. "A 4096-core RISC-V Chiplet Architecture for Ultra-efficient Floating-point Computing". In: *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society. 2020, 1.

[199] Zhijingcheng Yu, Shweta Shinde, Trevor E Carlson, and Prateek Saxena. "Elasticlave: An Efficient Memory Model for Enclaves". In: *arXiv preprint arXiv:2010.08440* (2020).

[200] AMD. *AMD I/O Virtualization Technology(IOMMU) Specification*. https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf. 2007.

[201] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. "Intel Virtualization Technology for Directed I/O." In: *Intel technology journal* 10.3 (2006), 179.

[202] ARM Holdings. *ARM system memory management unit architecture specification—SMMU architecture version 2.0*. 2013.

[203] sifive. *RISC-V Security Architecture Introduction*. https://sifive-china.oss-cn-zhangjiakou.aliyuncs.com/%E8%A5%BF%E5%AE%89%E7%8F%A0%E6%B5%B7%E6%9D%AD%E5%B7%9E%E5%90%88%E8%82%A5ppt/04%20hujin%20RISC-V%20Security%20Architecture%20Introduction_4%20City.pdf. 2019.

[204] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. "DR.SGX: Automated and Adjustable Side-Channel Protection for SGX Using Data Location Randomization". In: *Proceedings of the 35th Annual Computer Security Applications Conference*. ACSAC '19. San Juan, Puerto Rico: Association for Computing Machinery, 2019, 788–800.

[205] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, 217.

[206] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. "Towards practical page coloring-based multicore cache management". In: *Proceedings of the 4th ACM European conference on Computer systems*. 2009, 89.

[207] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. "Sonicboom: The 3rd generation berkeley out-of-order machine". In: *Fourth Workshop on Computer Architecture Research with RISC-V*. 2020.

[208] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. "Scattercache: Thwarting cache attacks via cache set randomization". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, 675.

[209] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. "Rendered insecure: GPU side channel attacks are practical". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, 2139.

[210] Chethan Ramesh, Shivukumar B Patil, Siva Nishok Dhanuskodi, George Provelengios, Sébastien Pillement, Daniel Holcomb, and Russell Tessier. "FPGA side channel attacks without physical access". In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2018, 45.

[211] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. "Visor: Privacy-Preserving Video Analytics as a Cloud Service". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[212] Clifford Wolf. *Yosys open synthesis suite*. http://www.clifford.at/yosys/. 2016.

[213] Florian Zaruba. *Ariane RISC-V CPU*. https://github.com/openhwgroup/cva6. 2020.

[214] Lowrisc. *Ibex RISC-V Core*. https://github.com/lowRISC/ibex. 2020.

[215] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini. "Snitch: A tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads". In: *IEEE Transactions on Computers* (2020), 1.

[216] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. "Graviton: Trusted execution environments on GPUs". In: *13th USENIX Symposium on Operating Systems Design and Implementation (osdi 18)*. 2018, 681.

[217] Peng Ning. "Samsung knox and enterprise mobile security". In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. 2014, 1.

[218] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. "SANCTUARY: ARMing TrustZone with User-space Enclaves." In: *NDSS*. NDSS, 2019.

[219] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. "vTZ: Virtualizing ARM TrustZone". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, 541.

[220] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. "SeCloak: ARM Trustzone-Based Mobile Peripheral Control". In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys, 18. Munich, Germany: Association for Computing Machinery, 2018, 1–13.

[221] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. "MINIX 3: A highly reliable, self-repairing operating system". In: *ACM SIGOPS Operating Systems Review* 40.3 (2006), 80.

CURRICULUM VITAE

PERSONAL DATA

Name	Aritra Dhar
Date of Birth	March 3, 1990
Place of Birth	Hooghly, West Bengal, India
Citizen of	India

EDUCATION

2016 – present	**Doctor of Science** ETH Zurich
2012 – 2014	**Master in Computer Science** IIIT Delhi, India
2008 – 2012	**Bachelor in Computer Science & Engineering** West Bengal University of Technology, India

WORK EXPERIENCE

Dec 2014 – Mar 2016	**Research Engineer** Xerox Research Center India, India
Jan 2013 – June 2013	**Research Intern** Accenture Technology Lab, India

COLOPHON

This document was typeset in LaTeX using the typographical look-and-feel `classicthesis` by André Miede[1]. The organization and source code template is heavily influenced from the PhD theses of Tino Wagner[2] and Patrick Pletscher[3]. Most of the graphics in this thesis are generated using `Microsoft PowerPoint`, `R Project`, and `tikz`. The bibliography is typeset using `biblatex`.

1 https://miede.de
2 https://github.com/tuxu/ethz-thesis
3 https://github.com/ppletscher/phd-thesis-sample