UNIVERSITY OF CALIFORNIA,
IRVINE

Enhancing System Security and Privacy with Trusted Hardware Components

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Networked Systems

by

Yoshimichi Nakatsuka

Dissertation Committee:
Professor Gene Tsudik, Chair
Professor Ardalan Amiri Sani
Doctor Andrew Paverd

2023

DEDICATION

To my mother Nakako and my sister Yuriko Nakatsuka, and my grandfather, Hirotsugu Yahashi.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to start by thanking my Ph.D. advisor, Professor Gene Tsudik, for providing me with the chance to pursue research at UCI in the first place. His guidance and feedback helped me become a better researcher and I am grateful for the support I received from him. I hope that these 5 and half years have proven that his decision of hiring me was not wrong.

I would also like to thank my committee members, Professor Ardalan Amiri Sani and Doctor Andrew Paverd. Their constructive feedback towards this thesis was very much appreciated and it was a great pleasure to have such established researchers on my Ph.D. committee.

In addition to my Ph.D. mentors, I would like to express my gratitude towards my early mentors, especially Professor Hiroaki Nishi, who was the very first to introduce me to the world of scientific research.

Throughout my Ph.D., I was extremely fortunate enough to have collaborated with a group of bright and fantastic people, both from within and from outside of UCI. Through their constructive criticisms and insights, I learned how to do better research. I must also mention that my experience in collaborating with my many excellent co-authors inspired me to aim for a research career.

I cannot thank enough my fellow labmates who went through the highs and lows with me during my Ph.D.: Tyler Kaczmarek, Norrathep Rattanavipanon, Ercan Ozturk, Seo Yeon Hwang, Sashidhar Jakkamsetti, Andrew Searles, Youngil Kim, Renascence Tarafder Prapty, Elina Van Kempen, Benjamin Terner, and of course, Professor Gene Tsudik. Thank you all for the fun times we spent together and I look forward to spending more in the future.

I am thankful for all my friends both in the US and in Japan and my family members who supported me throughout my time here.

I cannot start without mentioning my mother – Nakako – and my sister – Yuriko Nakatsuka. You were both my strongest supporters and knowing that you will always be there for me (even if we live far apart) helped me go through the toughest times.

I would also like to thank all my family members, especially my late grandparents and my many uncles and aunts. Even though we could not see each other often, I always appreciated your support.

Finally, I am extremely grateful to all my friends, especially everyone from my university and high school. It warmed my heart to see you all greeting me when I went back to Japan no different from when I was living there. Thank you all for rooting for me and for keeping in touch no matter the distance.

Last but not least, I would like to express my deepest gratitude towards The Nakajima Foundation, which provided me with financial support during the 5 years of my Ph.D. Their generous support alleviated many financial strains, which allowed me to focus on research.

—

Portion of Chapter 3 is a reprint of the material as it appears in the Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC 2019) [177] and in Digital Threats: Research and Practice, Volume 2, Issue 1 (DTRAP 2021) [178], used with permission from the Association for Computing Machinery. The co-authors listed in this publication are Doctor Andrew Paverd, and Professor Gene Tsudik.

Portion of Chapter 4 is a reprint of the material as it appears in the Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021) [175], used with permission from the USENIX Association. The co-authors listed in this publication are Doctor Ercan Ozturk, Doctor Andrew Paverd, and Professor Gene Tsudik.

Portion of Chapter 5 is a reprint of the material as it appears in the Proceedings of the 30th Network and Distributed System Security Symposium (NDSS 2023) [135], used with permission from the Internet Society. The co-authors listed in this publication are Professor Scott Jordan, Doctor Ercan Ozturk, Doctor Andrew Paverd, and Professor Gene Tsudik.

Financial support was provided by the University of California, Irvine, UCI School of ICS and Vice Chancellor of Research Seed Funding Awards, NSF Awards SATC-1956393 and CICI-1840197, and The Nakajima Foundation.

VITA

Yoshimichi Nakatsuka

EDUCATION

Doctor of Philosophy in Networked Systems **2023**
University of California, Irvine *Irvine, California*

Master of Science in Networked Systems **2020**
University of California, Irvine *Irvine, California*

Bachelor of Science in System Design Engineering **2017**
Keio University *Yokohama, Japan*

RESEARCH EXPERIENCE

Graduate Research Assistant **2017–2023**
University of California, Irvine *Irvine, California*

Research Intern **Summer 2022**
Cloudflare Research *Remote, US*

Research Intern **Winter & Spring 2021**
Microsoft Research *Cambridge, UK*

PROFESSIONAL EXPERIENCE

Product Security Intern **Summer 2019**
Edwards Lifesciences *Irvine, California*

Software Engineer Intern **Spring & Summer 2017**
Plaid, Inc *Tokyo, Japan*

TEACHING EXPERIENCE

TA for Computer and Network Security (CS134) **Fall 2019**
TA for Computer Networks (CS132) **Winter 2019**
University of California, Irvine *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

VICEROY: GDPR-/CCPA-compliant Enforcement of Verifiable Accountless Consumer Requests **Feb 2023**
Network and Distributed System Security Symposium (NDSS)

Vronicle: Verifiable Provenance for Videos from Mobile Devices **Jun 2022**
ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)

CACTI: Captcha Avoidance via Client-side TEE Integration **Aug 2021**
USENIX Security Symposium

PDoT: Private DNS-over-TLS with TEE Support **Dec 2019**
Annual Computer Security Applications Conference (ACSAC)

FROG: A Packet Hop Count based DDoS Countermeasure in NDN **Jun 2018**
IEEE Symposium on Computers and Communications (ISCC)

REFEREED JOURNAL PUBLICATIONS

PDoT: Private DNS-over-TLS with TEE Support **2021**
Digital Threats: Research and Practice (DTRAP)

PAPERS IN SUBMISSION OR UNDER REVIEW

An Empirical Study & Evaluation of Modern CAPTCHAs **2023**
USENIX Security Symposium

COMIT: Retrofitting Cooperative Migration to In-Process Trusted Execution Environments **2023**
TBD

Puppy: A Publicly Verifiable Watermarking Protocol **2023**
TBD

Scrappy: SeCure Rate Assuring Protocol with PrivacY **2023**
TBD

SOFTWARE

Daphne `https://github.com/cloudflare/daphne`
Implementation of the Distributed Aggregation Protocol (DAP) written in Rust

VICEROY `https://github.com/sprout-uci/VICEROY`
Implementation of VICEROY server and client written in C and JavaScript

Vronicle `https://github.com/trusslab/vronicle`
Implementation of Vronicle server and client written in C

PDoT `https://github.com/sprout-uci/PDoT`
Implementation of PDoT recursive resolver and client written in C

ABSTRACT OF THE DISSERTATION

Enhancing System Security and Privacy with Trusted Hardware Components

By

Yoshimichi Nakatsuka

Doctor of Philosophy in Networked Systems

University of California, Irvine, 2023

Professor Gene Tsudik, Chair

Trusted hardware components are essential when protecting the security of our devices and privacy of our online activities. Several kinds of trusted hardware components are widely available, most notably Trusted Execution Environments (TEEs) and Secure Hardware Tokens. Increasing availability of such hardware prompts a natural question: How can systems benefit from these trusted hardware components?

In this dissertation, we design four systems (COMIT, PDoT, CACTI, and VICEROY) that have enhanced security and privacy properties due to the integration of trusted hardware components. We identify and address the key challenges and issues that arise during the integration process. By evaluating proof-of-concept implementations of the four systems, we show that they meet necessary security, privacy, latency, throughput, and deployment requirements.

Chapter 1

Introduction

Research into protecting software and data using trusted hardware components dates back to the 1980s. Kent [137] introduced the important notion of *tamper-resistant modules*, which prevent adversaries from obtaining information contained within the module by erasing it when an attack is detected. Around the same time, Best [31] proposed using a crypto-microprosessor which holds a secret key that can be used to decrypt instructions of valuable programs, preventing unauthorized re-distribution or manipulation of proprietary software. The two studies were motivated by the increase in the need for software protection as computers were projected to be widely available to individuals. This sparked a wide range of research efforts (e.g., [209, 172, 90, 198, 117, 229, 230]) that investigate how software protection can be achieved by relying on *hardware* cryptographic co-processors.

In the late 1990s, attacks utilizing sophisticated malware increased at an alarming rate [183], which led to the formation of the Trusted Computing Platform Alliance (TCPA) in 1999 [201]. In early 2003, a nonprofit industry standards organization called the Trusted Computing Group (TCG) replaced TCPA. The main goal of TCPA and TCG was to provide hardware and software interface specifications, in order to increase interoperability between different

hardware and software vendors. This ultimately allowed the hardware to be adopted at a large scale, providing hardware-based security to the mass [30].

TCG announced the specification for its first Trusted Platform Module (TPM) version 1.1b [110] in 2003, version 1.2 in 2009 [111], and 2.0 in 2014 [112]. TPMs are used in a wide variety of applications, including OS secure boot [49] (a.k.a., static root-of-trust) and disk encryption software (e.g., Microsoft Windows BitLocker [167]).

Around the same time the TCG announced the TPM 1.1b specification, ARM started to develop a security architecture that provides hardware-assisted isolated execution. This later became ARM TrustZone [156], a Trusted Execution Environment (TEE) available to many devices that use ARM CPUs, especially smartphones. A TEE is a hardware-assisted security primitive capable of running arbitrary code in a secure environment isolated from all other software on the platform. An important feature of many TEEs is the ability to prove to a remote party that a specific code is running in the TEE, thus establishing a dynamic root-of-trust. In recent years, CPU manufacturers, such as Intel and AMD, introduced TEEs for desktop computers and servers, e.g., Intel Software Guard Extensions (SGX) and AMD Secure Encrypted Virtualization (SEV) in addition to the mobile-focused ARM TrustZone.

Apart from TEEs, Secure Hardware Tokens (e.g., Yubikey [235]) emerged as a new type of trusted hardware component. Similar to TPMs, these tokens are capable of securely storing secrets and performing cryptographic operations by utilizing an on-board Secure Element (SE). Secure Hardware Tokens (SHTs) commonly implement protocols standardized by the FIDO alliance [5], an industry-led protocol standard that aims to provide a means for authentication without the need for passwords [6]. The protocol also allows tokens to attest their authenticity to a remote party through a standard challenge-response protocol. Furthermore, SHTs include mechanisms to prove that a human is using the device through physical inputs such as buttons, touch sensors, and biometric sensors.

Both TEEs and SHTs have become widely available in client and server machines, prompting the following important question:

How can we utilize trusted hardware components to improve security and/or privacy of systems?

This dissertation primarily focuses on Web systems, as they pose unique security and privacy challenges. In particular, I focus on the following four challenges:

- Cloud computing platforms are increasingly adopting TEE-enabled services, including Azure Confidential Computing [162], GCP Confidential Computing [101], and AWS Nitro [9]. Migrating VMs and processes running in the cloud is essential for keeping services (e.g., web servers) available to users. However, how is this possible if a VM or process is protected using TEEs?

- Protecting DNS queries from network eavesdroppers is essential for user privacy. However, is it possible to protect them from adversarial DNS recursive resolver operators?

- CAPTCHAs have been used to thwart and mitigate both bot and excessive human activity However, as their effectiveness decline, solving CAPTCHAs have become frustrating and some raise concerns regarding user privacy. Can users bypass solving CAPTCHAs while maintaining the same security guarantees and benefiting from improved privacy?

- Data regulations provide consumers with legal rights to access/modify/delete data collected by websites. These same rights apply to those who do not hold an account on that website. Since collected data contain personal information, websites must authenticate the consumer. How can this be achieved while protecting consumer privacy and meeting scalability requirements?

In this dissertation, I explore an exciting new line of research on the integration of trusted

hardware components into the aforementioned systems to provide novel security and privacy properties and highlight the challenges and issues that arise in the process.

1.1 Contributions

The contributions of this dissertation are:

- Four novel architectures that offer improved security and privacy functionalities due to trusted hardware component integration: COMIT, *PDoT*, CACTI, and VICEROY.
- A focus on client-side trusted hardware in addition to server-side integration and the novel security/privacy guarantees they provide.
- An overview of challenges faced during the integration and their mitigation.
- Proof-of-concept implementations of the four systems.
- Comprehensive security, latency, throughput, and deployability evaluation of each implementation.

1.2 Dissertation Structure

Following the Introduction, Chapter 2 provides background information on TEEs. The following two chapters focus on *availability* and *privacy* issues of *server-side* systems and show how TEEs can help mitigate them. Chapter 3 focuses on availability issues for TEE-enabled processes in cloud environments via COMIT, a software-only design that retrofits migration functionalities into existing in-process TEE architectures. Chapter 4 presents *PDoT*, a Private DNS-over-TLS architecture that protects privacy of DNS requests even from adversarial DNS recursive resolver operators. The next two chapters shift toward *privacy* and *authenticity* issues on the *client side* and explore how trusted hardware components

can help. Chapter 5 points out privacy issues in current CAPTCHA systems and presents CACTI, a protocol that allows clients to avoid solving CAPTCHAs while providing the same security guarantees and improved user privacy. Chapter 6 introduces VICEROY, a protocol that allows accountless consumers to securely and privately authenticate themselves when accessing their data. Finally, Chapter 7 discusses directions for future work.

Note that background information specific to each project is described within each chapter.

Chapter 2

Background

2.1 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a security primitive that protects confidentiality and integrity of security-sensitive code and data from untrusted code through isolation (either virtually or physically). In this section, we provide an overview of basic functionalities offered by TEEs and summarize different instances of TEEs.

2.1.1 Overview of TEEs

A typical TEE provides the following features:

Isolated execution. The principal function of a TEE is to provide an execution environment that is isolated from all other software on the platform, including privileged system software, such as the OS, hypervisor, or BIOS. Specifically, data inside the TEE can only be accessed by the code running inside the TEE. The code inside the TEE provides well-defined entry points (e.g., call gates), which are enforced by the TEE.

Remote attestation. Remote attestation provides a remote party with strong assurances

about the TEE and the code running therein. Specifically, the TEE (i.e., the *prover*) creates a cryptographic assertion that: (1) demonstrates that it is a genuine TEE, and (2) unambiguously describes the code running in the TEE. The remote party (i.e., the *verifier*) can use this to decide whether to trust the TEE and then bootstrap a secure communication channel with the TEE. This is typically implemented by using an *attestation key*.

Data sealing. Data sealing allows the code running inside the TEE to encrypt data such that it can be securely stored outside the TEE. This is typically implemented by providing the TEE with a symmetric *sealing key*, which can be used to encrypt/decrypt the data. In current TEEs, sealing keys are platform-specific, meaning that data can only be unsealed on the same platform on which it was sealed.

Hardware monotonic counters. A well-known attack against sealed data is a *rollback attack*, where the attacker replaces the sealed data with an older version. Mitigating this requires at least some amount of rollback-protected storage, typically realized as a hardware monotonic counter. When sealing, the counter can be incremented and the latest value is included in the sealed data. When unsealing, the TEE checks that the included value matches the current hardware counter value. Since hardware counters themselves require rollback-protected storage, TEEs typically only have a small number of counters.

2.1.2 Instances of TEEs

This section describes several recent or current types of TEEs. We mainly focus on Intel Software Guard Extensions (SGX), as it is used to implement three out of the four systems proposed in the later chapters of this dissertation.

Intel Software Guard Extensions

One prominent example of a TEE is *Intel Software Guard Extensions (SGX)* [15, 118, 158]. SGX is a hardware-enforced TEE available on Intel CPUs from the Sky Lake microarchitec-

ture onwards.

SGX allows applications to create isolated environments, called *enclaves*, running in the application's virtual address space. This is enabled via a special region in physical memory reserved for enclaves, called the Enclave Page Cache (EPC), shared between all running enclaves. When enclave data leaves the CPU boundary, it is transparently encrypted and integrity-protected by the CPU's Memory Encryption Engine (MEE), defending against physical bus snooping/tampering attacks. The EPC of CPUs up to the 9th generation can hold up to 128 MB of code and data, while the 10th generation CPUs have been upgraded to hold up to 1 TB. Since enclaves run in the application's virtual address space, enclave code can access all memory of its host application, even that outside the enclave.

A thread can only invoke enclave code via predefined function calls, called `ECALL`s. Any attempt to jump into the enclave without calling an `ECALL` is prevented by the CPU.

Every enclave has an enclave identity (`MRENCLAVE`), which is a cryptographic hash of the code that has been loaded into the enclave during initialization, and various other configuration details. Each enclave binary must be signed by the developer, and the hash of the developer's public key is stored as the enclave's signer identity (`MRSIGNER`).

SGX provides two types of attestation: local and remote.

Local attestation is used by an enclave (A) to convince another enclave (B) that A is a genuine Intel enclave and that both A and B are running on the same physical machine. This is enabled through a shared symmetric key only known to the processor. The attestation procedure is as follows:

1. Enclave A receives the `MRENCLAVE` value of enclave B.

2. A asks the hardware to generate a *report* destined for B using the `MRENCLAVE` value of B.

3. The report is generated by the hardware and forwarded to B. It is important to note that at this step A can pass in data to the report. This data then can be used to create a secure channel.

4. B receives the report, verifies it, and concludes that A is running on the same platform.

Remote attestation allows remote clients to cryptographically check what code is exactly running inside the TEE. This is to assure that the client is talking to a genuine TEE or that the data received from a TEE was generated by expected code. Remote attestation process includes local attestation with a special enclave called *quoting enclave*, an enclave created by Intel. The quoting enclave holds a group private key (SGX uses the EPID [41] group signature scheme for this purpose) called the *attestation key* which is obtained by the provisioning enclave during the provisioning phase. The provisioning is done during the initial setup (or later if any critical components are updated due to vulnerabilities). Once the quoting enclave verifies the content of the report it received from the enclave via local attestation, it signs the report using its attestation key thus creating a *quote*. The quoting enclave gives the quote back to the enclave and the enclave can hand the quote to the remote party. The remote party then contacts the Intel Attestation Service or an equivalent attestation service (e.g., Microsoft Azure Attestation [163]) to verify that this quote is genuine. If verification succeeds, the remote party is convinced that the enclave is running the expected code. Moreover, using group keys as remote attestation keys prevents malicious remote parties to identify or link platforms running the enclaves.

In SGX, data can be sealed in one of two modes, based on: (1) the enclave's identity (i.e., `MRENCLAVE`), such that only the same type of enclave can unseal it, or (2) the signer identity (i.e., `MRSIGNER`), such that any enclave signed by the same developer (running on the same platform) can unseal it. SGX provides hardware monotonic counters and allows each enclave to use up to 256 counters at a time.

Other TEE instances

Commercially available TEEs include AMD Secure Encrypted Virtualization (SEV) [12], SEV Encrypted State (ES) [14], SEV Secure Nested Paging (SNP) [13], ARM TrustZone [23], and AWS Nitro Enclave [11]. Commercially planned TEEs include Intel Trusted Domain Extensions (TDX) [126] and ARM Confidential Compute Architecture (CCA) [21]. Academic projects include Keystone [146], Sanctum [65], and Penglai [86].

A note on attacks on TEEs. Ever since their introduction, there have been numerous attacks on TEEs, including cache side-channel [151, 38, 69, 109, 170], control channel side-channel [232], memory side-channel [227], fault injection [174, 53], and attacks exploiting speculative execution [140, 150, 44]. In response, many mechanisms have been proposed to mitigate such attacks [207, 216, 37]. We refer to [199] for a comprehensive discussion of attack vectors and countermeasures.

We emphasize that defending against such attacks is *orthogonal* to the work in this dissertation and that all proposed systems assume that the TEE architecture as well as all algorithms and cryptographic primitives within the TEE are implemented correctly.

Chapter 3

COMIT: Retrofitting Cooperative Migration to In-Process Trusted Execution Environments

Abstract

Hardware-based Trusted Execution Environments (TEEs) are becoming increasingly prevalent in cloud computing, forming the basis for *confidential computing*. However, the security goals of TEEs sometimes conflict with cloud functionality, such as VM or process migration, because TEE memory cannot be read by the hypervisor, OS, or other software on the platform. While some newer TEE architectures support migration of entire protected VMs, there is currently no practical mechanism for migrating individual processes containing in-process TEEs. The inability to migrate such processes leads to operational inefficiencies or even data loss if the host platform must be urgently restarted.

In this chapter, we present *COMIT*, a software-only design to *retrofit* migration functionality into existing TEE architectures, while maintaining their expected security guarantees. Our design allows TEEs to be interrupted and migrated at arbitrary points in their execution, thus maintaining compatibility with existing VM and process migration techniques. By cooperatively involving the TEE in the migration process, our design also allows application developers to specify stateful migration-related policies, such as limiting the number of times a particular TEE may be migrated. Our prototype implementation for Intel SGX demonstrates that migration latency increases linearly with the size of the TEE memory and is dominated by TEE system operations.

3.1 Introduction

Confidential computing is an emerging model in cloud computing, which is already offered in some form by each of the three largest cloud providers [162, 101, 9]. The primary aim of confidential computing is to protect data in use, e.g., against insider threats or compromise of the underlying cloud infrastructure. Currently, the leading approach for achieving this is to use hardware-enforced Trusted Execution Environments (TEEs).

Although there are multiple TEE technologies, the overarching idea is the same — to create a strong security boundary between the TEE and other software components. Misbehavior by any component outside the TEE cannot violate confidentiality or integrity of the TEE. Specifically, data within the TEE can only be read or modified by code within the same TEE. TEEs often also provide *remote attestation* functionality, through which a remote party can ascertain what code runs within the TEE, and use this information to make security decisions.

Current TEE technologies can be divided into two groups: those that enable the creation of one or more *in-process* TEEs within an application process, and those that protect larger structures such as containers or entire VMs.

However, the hardware-based nature of most modern TEEs conflicts with VM or process migration. By design, TEE memory cannot be read by the hypervisor, OS, or other software on the platform, preventing existing migration techniques to be used directly on systems containing TEEs. While some newer TEE architectures support migration, this is not universally available, and there are no in-process TEE architectures that support migration.

There are several reasons why migration is important in cloud computing. From an operational perspective, the cloud provider may want to move VMs to different physical machines to reduce the number of active machines. From a security perspective, the cloud provider may need to restart specific physical machines to apply firmware security updates, e.g., to defend

against newly-identified side-channel attacks against the TEE [140, 150, 44, 233, 174, 221]. Finally, the tenants (customers) of the cloud provider may want to move their workloads to a different provider. In these scenarios, the inability to migrate a process that uses a TEE could lead to operational inefficiencies or, in the worst case, data loss if the physical machine must be restarted.

In light of this, several TEE architectures have announced native support for migrating TEEs, namely AMD SEV [12], SEV-SNP [13], and Intel TDX [126]. Some prior work [186, 185, 113] has proposed approaches to support migration of VMs with Intel SGX enclaves. However, these TEEs are *VM-based*, not in-process.

Compared to VM migration, in-process TEEs are harder to migrate because they are not designed with migration in mind. There have been several proposals to *retrofit* migration functionality into existing in-process TEE architectures. Guerreiro et al. [114] use Hardware Security Modules (HSMs) to manage the cryptographic keys needed to securely migrate the data; Alder et al. [4] describe how to migrate the persistent state (e.g., hardware-based monotonic counter values) associated with a TEE. However, all of these require either change to the hardware (which is likely to be impractical given the large deployed base), or the assumption that the TEE will reach a quiescent state before it is migrated, which limits the applicability of the technique.

In this chapter, we present COMIT, a software-only design to *retrofit* migration functionality into existing TEE architectures, while maintaining their expected security guarantees. The core idea is to add a minimal set of extra functionality to the TEE and then *enlighten* the migration tool to make use of this functionality. COMIT makes the following contributions:

(1) It enables migration of *existing* in-process TEE architectures, without requiring modifications to the hardware or placing constraints on the software running within the TEE. This is challenging because it requires a software-only mechanism that can operate within the

constraints of existing TEE architectures (e.g., in Intel SGX, some critical data structures are inaccessible even from software within the enclave). Furthermore, these architectures may not have been designed with migration in mind.

(2) It allows TEEs to be interrupted, migrated, and resumed at arbitrary points in execution, thus matching the paradigm of existing process migration tools, such as CRIU [182]. This allows us to integrate COMIT with current tools with minimal changes. Our method for *enlightening* these tools is itself extensible and could be used to enable new types of process migration behavior.

(3) It involves the TEE in the migration operation, resulting in a type of *cooperative migration*. This gives TEE application developers the ability to specify flexible stateful policies to govern migration. Examples of such policies may be to limit the number of times a particular TEE is migrated or to migrate only a subset of the TEE's memory.

As a proof of concept, we implemented COMIT for Intel SGX. Through micro and macro benchmarks, we show that migration latency increases linearly with the size of the TEE and that the overhead is dominated by TEE system operations, mainly creation and termination of TEEs.

3.2 Migration is the Cloud

For a large number of cloud services, liveness relies on the availability of the machines on which the service is deployed [165, 96, 8]. This architecture runs counter to the deployment philosophy of many cloud providers. Cloud providers consider a single machine to be expendable and instead define large availability zones where machines within a single availability zone may become simultaneously unavailable while machines in different availability zones will be available [161, 97, 10]. In this situation, migrating the application from a machine that the cloud provider plans to soon shut down provides a high level of availability.

Therefore, migrating TEEs is important, especially as confidential computing is gaining popularity amongst cloud providers. In this section, we describe *VM-based* TEEs that support migration and then discuss the state-of-the-art system for *in-process* TEEs.

TEE VM migration. AMD enabled its TEE extensions SEV and SEV-SNP with hardware support for live migration [12, 13]. The hardware extensions provided by AMD SEV are called AMD Secure Processor (SP) which manages the encryption keys required for live migration. During TEE VM migration, the destination SP attests itself to the source SP and once the attestation is verified, the source SP sends the key securely to the destination SP. After verification is completed successfully the source SP sends encrypted TEE VM pages to the destination SP, which uses the key sent by the source SP to decrypt the pages and copy them into the destination TEE VM.

AMD SEV-SNP improves upon the AMD SEV model by introducing *migration agents*. A migration agent oversees the enforcement of migration policies removing the requirement for the VM to maintain its migration policy.

Intel announced live migration support for their unreleased TEE extension, TDX [126]. TDX creates TEE VMs, called *trust domains* (TDs). Similar to SEV-SNP's migration agent, TDX utilizes an entity called *migration TD*. The source and migration TDs conduct mutual remote attestation and negotiate a key to encrypt the contents of the migrating TD.

Although Intel's current TEE architecture, SGX, does not natively support migration, several attempts had been made. The first was by Park et al., which proposed a design that supports live migration of SGX-enabled VMs [186]. It discussed several problems of migrating such VMs, including secure migration of enclave memory. The idea was to introduce a new set of CPU instructions to enable live migration. A follow-up work [185] implemented the new set of instructions using OpenSGX [132], a fully functional Intel SGX emulator based on QEMU.

Gu et al. [113] utilize a control thread to securely copy an enclave's state from within the TEE. The system uses a scheme called "two-phase checkpointing", which requires an enclave to reach a checkpoint before migrating, to ensure data consistency of migrated enclaves. The system was implemented using a variant of the KVM that provides support for Intel SGX in applications, guest OSs, and KVM itself.

Unfortunately, these results cannot migrate a process running within an operating system without the coordination of the application being aware and adding logic that coordinates with the migration coordinator. This results in TEEs losing a considerable amount of utility when run on a machine within a cloud data center. Additionally, aforementioned systems cannot migrate a process running within the TEE without migrating the entire VM. This increases latency and adds strain to the cloud provider's network bandwidth, as a larger amount of data must be migrated. This motivates migrating *in-process* TEEs without the need of migrating the entire underlying VM.

In-process TEE migration. To the best of our knowledge, TEEnder [114] is the only work that aims to enable migration for in-process TEEs, specifically Intel SGX. TEEnder uses Hardware Security Modules (HSMs) to encrypt and decrypt SGX enclave data during migration. The main motivation behind using HSMs is the recent findings of security vulnerabilities surrounding SGX remote attestation [147, 206, 215, 221]. TEEnder realizes this by integrating enclave applications with HSMs and implementing an infrastructure that utilizes HSMs to provide enclaves with migration capabilities. However, the usage of HSMs, although motivated clearly, will decrease the deployability of the system as well as the performance of migration.

Given the challenges above, a design for migrating TEE-containing processes must fulfill the following requirements:

R1 The migration functionality must be *retrofitted* into existing TEE architectures.

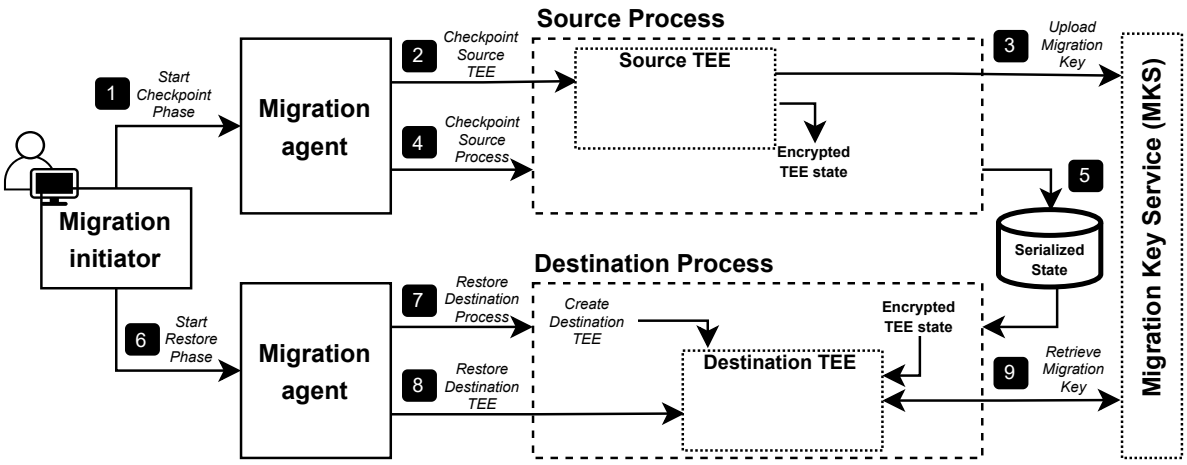


Figure 3.1: Overview of the main components COMIT and the interactions between them.

R2 The design must maintain the existing security guarantees provided by the TEEs.

R3 The TEE developer must be able to define stateful policies that govern migration.

We present an overview of our design in the next section and show how it meets these requirements in Section 3.4.

3.3 System Overview

Figure 3.1 shows an overview of the main entities in COMIT, and the interactions between them.

The migration begins when the *migration initiator* decides that a process should be migrated from the node on which it is currently running (the *source*) to another node (the *destination*). The source process contains at least one TEE that must be included in the migration. The migration can be broadly divided into two phases: *checkpoint* (§3.3.1) and *restore* (§3.3.2).

3.3.1 Checkpoint phase

The first step in the checkpoint phase is to pause the execution of the source process by stopping all currently executing threads, including those within the TEE. With the execution paused, the state of the source process can be serialized and saved. There are several existing tools, such as CRIU [182], that can be used to serialize and save the state of the source process. We refer to this type of tool as the *migration agent* on the source (and destination) node.

However, since the migration agent is running outside the TEE, it cannot directly read the TEE's memory, as would be required to perform the migration. Using an unmodified migration agent on a process containing a TEE typically results in an access violation. COMIT, therefore, requires a small amount of additional functionality to the TEE, through the inclusion of a software library. This functionality can be used to request the TEE to serialize its state, encrypt it, and write the output to the memory of the source process. The cryptographic key used to encrypt (and subsequently decrypt) the TEE's state cannot be revealed to the source process, so it is securely transferred to a trusted *migration key service*.

The existing migration agent can be *enlightened* to i) recognize that the source process contains a TEE, and ii) use the additional functionality added to the TEE to perform the migration. The output of the checkpoint phase is the serialized state of the source process, including the encrypted state of one or more TEEs. This state is then transferred to the destination host.

3.3.2 Restore phase

The restore phase begins when the destination node receives the state from the checkpoint phase. This state is used to recreate the saved process (now referred to as the *destination process*), using a migration agent on the destination node (e.g., CRIU). Using information from the saved process state, the migration agent also creates one or more fresh TEEs in the

destination process, corresponding to the TEEs that were paused in the source process.

However, the migration agent does not have the decryption keys for the encrypted TEE state and cannot write directly to the memory of the destination TEE. Similar to the checkpoint phase, COMIT delegates the task of restoring the TEEs' internal state to the TEEs themselves. Specifically, the additional functionality added to the TEE can also be used to restore a previously-saved TEE state onto a newly-initialized TEE.

The destination TEE securely retrieves the decryption keys from the migration node, using remote attestation to demonstrate that it is the correct type of TEE running the expected code. Once the keys have been retrieved, the destination TEE decrypts the saved state and transforms itself by overwriting its own heap, stack, and other data structures with those from the restored state. Once the migration has been completed, this restored TEE can continue operating from the same point at which the source TEE was paused.

3.4 Design Challenges

This section discusses challenges arising from requirements in Section 3.2 and approaches for addressing them in COMIT.

3.4.1 Retrofitting migration to existing TEEs

The requirement to support existing TEE architectures, which may already be widely deployed, precludes modifying TEE hardware (e.g., [186]) and necessitates a software-only approach. Since only the software running within the TEE can read and write TEE memory, the checkpoint and restore functionality must be provided from within the TEE. As described in Section 3.3, COMIT adds these functionalities by including an additional software library within the TEE. In the future, this library could be included by default in the TEE development frameworks (e.g., Open Enclave [164]). Performing the migration from

within the TEE raises specific challenges in both checkpoint and restore phases.

Arbitrarily pausing TEEs. Halting a TEE without modifying TEE hardware is a challenge. One way of doing this is to wait until the TEE reaches a certain point in its execution and exit (e.g., [113]). However, this requires the TEE to be aware of the migration and thus may not fit the requirements of certain migration agents (e.g., CRIU requires processes to be arbitrarily paused). This means that COMIT must be able to halt a TEE at any point in time. COMIT realizes this by using *interrupts*, which are widely supported across different TEE architectures.

Operating within the TEE. The checkpoint functionality is invoked once the source TEE has been paused, and therefore must carry out its operations without affecting the state of the source TEE. In addition, the working state of this operation must not be included in the saved state of the TEE. Techniques used to overcome these challenges will vary by TEE technology (e.g., we describe our implementation for Intel SGX in Section 3.5). However, in general, this requires the checkpoint operation to use its own stack and heap memory. The restore functionality faces similar challenges in that it must decrypt and process saved state within the fresh destination TEE and then overwrite the state of that TEE (stack and heap) with restored state. This means that the restore operation must use its reserved memory within the TEE.

Understanding TEE memory. The next challenge is that the checkpoint and restore operations within the TEE must understand the TEE's memory map. For example, the TEE's memory might contain control structures that cannot be read or written even by software running within the TEE (e.g., the Thread Control Structures in Intel SGX). The migration operations must take care to avoid these memory regions. Additionally, COMIT may need to use architecture-specific techniques to infer the values held in these control structures in the source TEE and to correctly set these values in the destination TEE.

3.4.2 Maintaining TEE security

When migrating TEEs, COMIT needs to ensure that existing TEE security guarantees still hold. Specifically, the only change to the security model is that the TEE can be migrated. As described in Section 3.3, the checkpoint operation encrypts the TEE's memory before writing it outside the TEE. Specifically, an authenticated encryption scheme, such as AES-GCM, must be used so that the integrity of the saved state can be verified by the destination TEE. We refer to the symmetric encryption/decryption key as the *migration key*.

Secure key transfer. The migration key cannot be directly exported with the saved state — it must instead be securely transferred to the destination TEE. If the source and destination TEEs were running concurrently, they could use existing remote attestation functionality to mutually attest each other, establish a secure channel, and securely transfer the migration key. However, requiring both TEEs to be running concurrently would severely limit the applicability of the approach. For example, existing process migration tools such as CRIU [182] operate strictly sequentially: the checkpoint phase is completed before the restore phase begins. Requiring concurrently running source and destination TEEs would also preclude the possibility of *self-migration*, where the source and destination are the same physical node. This would be used to allow the node to be restarted e.g., to install security firmware updates.

To overcome this challenge, COMIT makes use of a new *migration key service* (MKS), which serves as a trusted intermediary and key escrow service between the source and destination TEEs. Specifically, once the source TEE has generated the migration key, it establishes a secure channel with the MKS and sends the migration key. The restore operation in the destination TEE retrieves the migration key from the MKS and uses it to decrypt the TEE state.

The MKS is a relatively simple store-and-forward helper service, for which there are various

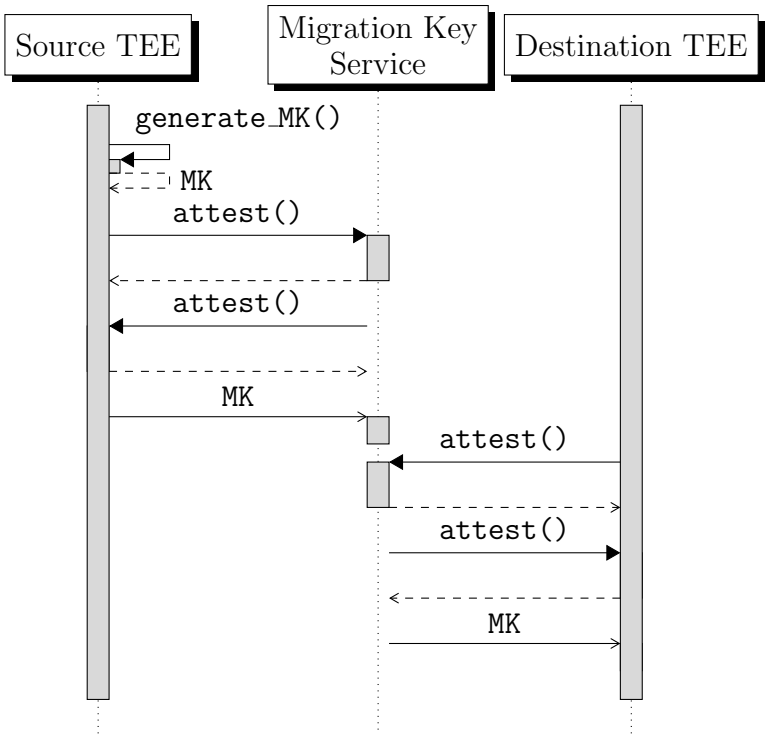


Figure 3.2: Secure transfer of the migration key (`MK`) from source to destination TEE via the Migration Key Service. All communication takes place via secure channels.

possible implementations. One possibility is to implement the MKS using another TEE, either on the source, destination, or another node. As shown in fig. 3.2, the source TEE attests the MKS to ensure that the migration key is only transferred to a trustworthy MKS, and the MKS attests the source TEE to verify the provenance of this key. Subsequently, the MKS attests the destination TEE to ascertain that it is the correct type of TEE (e.g., the same as the source TEE), and the destination TEE attests the MKS to again verify the provenance of this key. This provides the same security guarantee as the direct key transfer between source and destination TEE.

Fork and roll-back attacks. In addition to secure key transfer, the design must also ensure that the migration functionality itself cannot be used to mount attacks such as a fork or roll-back attack [4]. Specifically, we need a mechanism to ensure that the source TEE cannot continue running after the checkpoint operation has been completed, as this could lead to a fork attack with multiple copies of the same TEE running. To prevent this, COMIT

blocks the source TEE from being resumed before the output from the checkpoint operation is released. Although the precise implementation is architecture-dependent, it can always be achieved through minor modifications of the TEE's software. We also require a mechanism to prevent one saved state from being restored to multiple destination TEEs (another type of fork attack) or being restored more than once (a roll-back attack). This cannot be prevented by changes within the TEE, so COMIT requires the MKS to only release the migration key to a single destination TEE.

3.4.3 Supporting policy-governed migration

The design must provide mechanisms that TEE developers can use to define and enforce policies to govern the migration of the TEE. COMIT does not define any specific policies, rather it aims to provide as much flexibility as possible to TEE developers. Specifically, COMIT enforces policies by calling a developer-defined function (which calls additional functions) during the restore phase before allowing the TEE to resume operation. Since the TEE state has already been decrypted and put into place, this function can be stateful and can inspect the full state of the TEE. The return value of this function indicates whether the TEE should be allowed to resume operation. This ensures that every restore operation is visible to the TEE. We sketch two example policies to illustrate the use of this mechanism.

Limited number of migrations. One example policy could be to limit the number of times a specific TEE can be migrated. This may be useful in cases where the TEE developer is concerned that an excessive number of migrations might leak information from the TEE (e.g., through side-channel attacks) and limit the number of times the cloud provider migrates a TEE. To implement this, the developer would define a counter variable in the TEE's memory and decrement this on each successful restore operation. When the counter reaches zero, the function would indicate that the TEE should not be permitted to resume.

Clearing caches upon migration. As another example, the policy might not need to gov-

ern whether the TEE can be resumed, rather define a set of actions that must be performed after each migration. For example, the TEE might contain a node-specific state (e.g., some type of cache of local sessions) that should be invalidated if the TEE is migrated. Again this can be achieved by calling a developer-defined function that clears/resets this part of the state whenever the TEE is restored.

3.5 Implementation

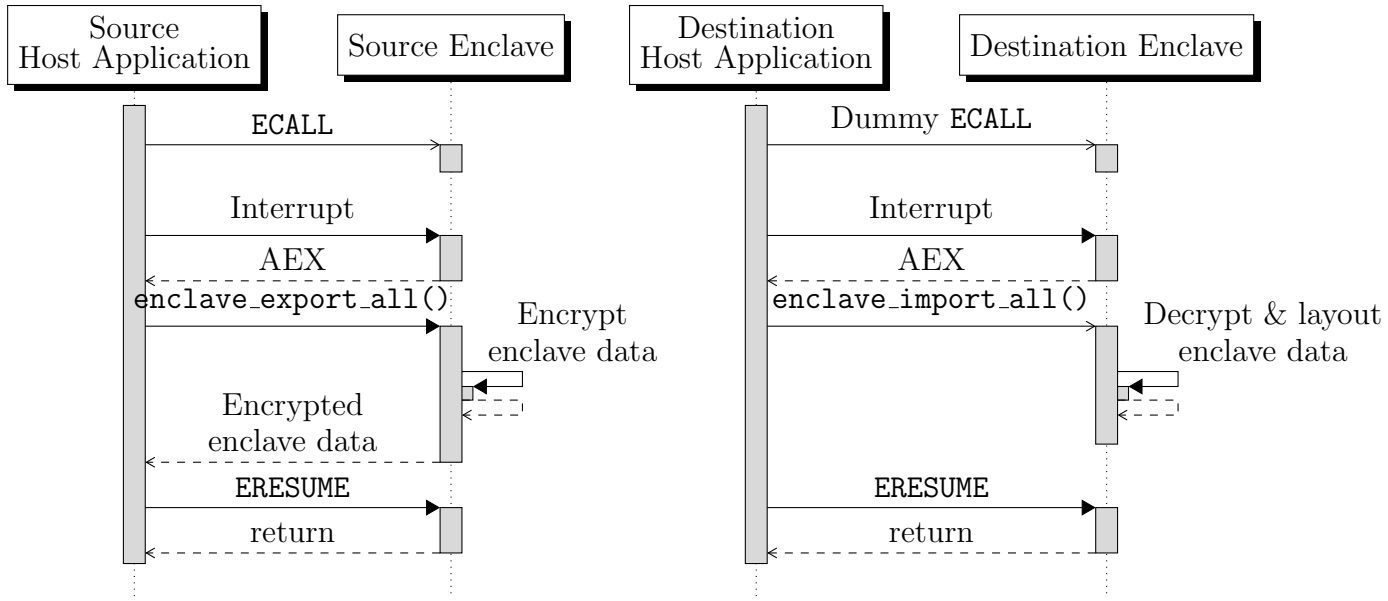
This section describes COMIT-SGX, an implementation of COMIT for Intel Software Guard Extensions. Although our implementation is based on the Open Enclave SDK [164] v0.16.1, it could be applied to any other SGX SDK. We first describe the specific steps required for migrating an Intel SGX enclave (§3.5.1) and then discuss how we integrated these into the CRIU [182] process migration tool (§3.5.2).

3.5.1 Enclave Migration

In practice, migrating an SGX enclave requires some additional preparation before the checkpoint phase and some additional cleanup after the restore phase. We, therefore, describe this process in terms of the following four phases:

1. **Preparation Phase**: Initializes variables used for migration.
2. **Checkpoint Phase**: Collects, encrypts, and exports all necessary data from the source enclave.
3. **Restore Phase**: Imports, decrypts, and restores all data to the destination enclave.
4. **Cleanup Phase**: Cleans up data structures (e.g., buffers) used for migration.

Below, we describe each phase in detail.



(a) Checkpoint phase (b) Restore phase

Figure 3.3: Enclave migration flow

Preparation Phase

The preparation phase is run asynchronously, any time before the enclave is to be migrated. This can either be integrated into the enclave's initialization sequence or called via a separate `ECALL`. In this phase, the enclave generates a migration key and initializes the encryption context (i.e., the data structures used by the cryptographic library). It also retrieves the memory addresses and sizes of the enclave's stack, heap, and data sections, as well as the address and size of the SGX-specific State Save Area (SSA). These values are stored within the enclave in preparation for migration.

Checkpoint Phase

To initiate the checkpoint phase, the host should interrupt all enclave threads and call a newly-added `ECALL` (called `enclave_export_all`). In this `ECALL`, the host provides a pointer to a memory buffer outside the enclave, into which the encrypted state should be written. The enclave then performs several steps to serialize, encrypt, and save its state.

Halting enclave threads. COMIT-SGX uses interrupts to pause TEEs. In Intel SGX, interrupting a thread that is within the enclave causes an event called an Asynchronous Exit (AEX). When it occurs, the CPU saves the thread's state in the enclave's State Save Area (SSA). This saved state includes the Thread Local Storage (TLS), current instruction pointer, stack and base pointers, and other register values (within a data structure called GPRSGX). The CPU then clears the registers and jumps to a predefined code location outside the enclave.

The host process (i.e., the source process) can send a signal to each thread within the enclave to cause an AEX. However, the default behavior of most SGX frameworks is for the thread to immediately re-enter and resume executing within the enclave after the interrupt has been handled. The host process, therefore, needs to take additional steps to prevent the thread from re-entering the enclave. A simple way would be to cause the thread to sleep or spin in an infinite loop. However, all enclave threads jump to the same code address outside the enclave when they are interrupted. At least one thread is required to perform the checkpoint operations within the enclave, and this thread might also be periodically interrupted. If this thread is interrupted, it should not be held outside the enclave.

The host process can overcome the above issues using three flags: `IS_HOST_MIGRATING`, `HAS_ENTERED_ENCLAVE`, and `ALLOW_ERESUME`. All three start in the unset state. The host process first sets the `IS_HOST_MIGRATING` flag to indicate that it is going to halt the enclave thread. This flag causes the interrupted enclave threads to be kept in an infinite loop, which is conditioned to break when the `ALLOW_ERESUME` flag is set. Finally, the `HAS_ENTERED_ENCLAVE` flag is set immediately before the migration thread enters the enclave. This flag prevents any new threads from being caught in the infinite loop, while still holding the previously-captured threads.

Once the migration thread has entered the enclave (via the `enclave_export_all ECALL`), it could set an in-enclave flag to prevent any other `ECALL`s from being made. It could

also prevent any threads from being resumed while the migration is in progress by saving and overwriting the saved instruction pointer values in the SSA. If any thread did resume operation after this point, it would cause the enclave to crash.

Exporting enclave data. With the enclave paused, the next step is to encrypt and export the enclave's data. This includes (1) data section, (2) heap section, (3) stack, and (4) SSA. These data are serialized by concatenating all the above in the order shown. We do not export code data because we assume both the source and destination enclaves are running the same code.

Additionally, every SGX enclave includes one or more Thread Control Structures (TCS), which cannot be read or written even by code running within the enclave. Each TCS contains a Current State Save Area (CSSA) value, which indicates how many threads have been interrupted and are now outside of the enclave. Since we are using interrupts to halt enclave threads, we need to infer the CSSA value. We could employ the same method proposed in [113], which introduces a software monitor to keep track of how many threads have entered and exited the enclave.

Recall that enclave data must be encrypted before it can be written outside the enclave (Section 3.4.2). We use AES in GCM mode (i.e., authenticated encryption) with a 256-bit key for this purpose. We implemented this using both the cryptographic libraries supported by Open Enclave: mbedTLS and OpenSSL. Once encrypted, that data is written to the specified buffer outside the enclave.

Restore Phase

The restore phase also consists of several steps. First, the destination enclave must be created and initialized. Second, COMIT-SGX creates one or more placeholder threads within the destination enclave, corresponding to the threads that were interrupted from the source enclave.

Third, the host process initiates the restore process by calling a new `ECALL` (`enclave_import_all`). This `ECALL` takes the pointer to the encrypted enclave data received from the source process. Finally, the restored enclave threads resume execution from the point at which they were interrupted.

Creating placeholder threads. Placeholder threads are essentially enclave threads with an infinite loop. There are several reasons for creating placeholder threads in the destination enclave. First, we need a pre-allocated memory region to lay out the exported enclave data back to its original location. Second, this causes the CPU to increase the CSSA value in the TCS. This technique overcomes the limitation of not being able to write to the TCS from software, and can thus be used to set the correct CSSA values. COMIT-SGX, therefore, creates a placeholder thread in the destination enclave for each thread that was interrupted from the source enclave.

Restoring enclave data. The main challenge during the restoration phase is to ensure that the enclave data is restored to the correct location. For security reasons, the saved state must be decrypted within the destination enclave. The data structure used by the cryptographic library (i.e., the decryption context) is typically allocated on the heap. This creates a problem when we restore the heap section, because we may potentially overwrite the decryption context when we are decrypting and restoring the subsequent section. To overcome this issue, COMIT-SGX places the decryption context in the global data section and intentionally avoids that area during the restoration process. This is possible because the same code runs in the source and destination enclaves, allowing encryption and decryption contexts to be allocated in the same memory location.

Cleanup Phase

Both source and destination processes carry out a cleanup phase after they have completed their respective roles in the migration. The source node tears down the enclave. Alter-

natively, if allowed by the enclave developer, the host process may resume the enclave by setting the `ALLOW_ERESUME` flag. This is essentially "forking" the enclave, which may be desirable in some circumstances. On the destination node, the buffer that was used to store the encrypted enclave data can be freed.

3.5.2 Integrating COMIT-SGX into CRIU

In this section, we describe how enclave migration can be integrated into existing process migration tools. Specifically, we integrate COMIT-SGX into CRIU [182]. Below we describe this integration in each of the four phases discussed above. Although we describe this in the context of a source process that contains a single enclave, it applies to the case of multiple enclaves.

Preparation Phase

In addition to the steps described in Section 3.5.1, the source process must perform several steps. First, it must close file descriptors to `stdin`, `stdout`, `stderr`, and `/dev/sgx`, as they cannot be migrated by CRIU. Second, it must allocate a memory buffer to store the encrypted enclave memory, the address of which is passed as a parameter when calling `enclave_export_all`. Finally, it creates a file containing code pointers to several functions within the source process, which will be used in both the checkpoint and restoration phases.

Checkpoint Phase

This phase begins when the migration initiator instructs the migration agent (in this case, CRIU) to migrate a specific process (in this case, the source process). CRIU process transitions the source process into a "seized" state and begins to determine what must be migrated. To do this, CRIU injects a piece of code (called the *parasite code*) into the source process. The parasite code allows CRIU to gain access to resources held by the source process, such as file descriptors and threads. However, by design, even this parasite code cannot read the

memory of the SGX enclave.

Extending CRIU. COMIT-SGX, therefore, extends the parasite code to work with the source enclave in order to achieve the migration. Specifically, we extend the parasite code to call the `enclave_export_all ECALL`. Although it may be possible to make this `ECALL` from outside the source process (e.g., from the migration agent), this would involve several address translations. A more natural approach is for the injected parasite code to make this `ECALL`, since it: i) has direct access to variables in the source process, such as the enclave context, and ii) can directly invoke functions from the source process's address space, such as the `ECALL`.

Determining function addresses. However, the addresses of the `enclave_export_all ECALL` and other enclave functions are not deterministic, due to memory address randomization. To overcome this issue, our enlightened version of CRIU looks up the necessary function pointers in the file created in the preparation phase (§3.5.2). Specifically, the parasite code opens the file from a predefined location, retrieves the function pointers, and invokes each of the functions sequentially.

Passing parameters. Since this approach does not allow CRIU to pass arguments to any of the functions, the function pointers in the file should point to wrapper functions that do not take any parameters. These wrapper functions could in turn call other functions for which the parameters have been determined in advance. A specific example is the `enclave_export_all ECALL`, which must include the address of the memory buffer outside the enclave into which the encrypted state will be written. This parameter can be determined when the buffer is allocated (i.e., in the preparation phase). CRIU calls a wrapper function with no parameters, which in turn calls this `ECALL` with the pre-prepared parameter value.

Further extensibility. This approach is extensible in that the application developer can provide additional functions for CRIU to invoke before starting the migration. Apart from

enabling cooperative migration of TEEs, this could also be used to support other types of extended functionality e.g., if the source process needs to perform some non-standard clean-up operations or close some resources before being migrated.

Overall checkpoint process. The interaction between the source process and the source enclave is shown in Figure 3.3a. Putting it all together, the checkpoint process proceeds as follows: When CRIU seizes the source process, this will interrupt and pause all of the source process's threads. Specifically, this will cause an AEX for any threads within the enclave, as required. Once the process is seized, CRIU injects the parasite code into the source process and invokes it. The parasite code then calls the `enclave_export_all` `ECALL`, which as described above, causes the enclave to serialize, encrypt, and write its state to the memory of the source process. This memory will then be included in the saved process image output by CRIU, alongside the rest of the source process's memory. Finally, after all the data has been exported, the parasite code destroys the enclave. The migration agent can then transfer the saved process image to the destination node.

Restore Phase

The restore phase begins once the migration initiator has transferred the process image to the destination node and invoked the migration agent (CRIU) on the destination node. The CRIU process on the destination node uses this process image and essentially *transforms* itself into the restored destination process. To restore the process memory, CRIU uses another separate piece of code called the *restorer blob*. As with the parasite code in the checkpoint phase, COMIT-SGX extends the CRIU restorer blob to handle restoration of the enclave. The interaction between the destination process and the destination enclave is shown in Figure 3.3b.

Restoring a process with CRIU. CRIU first restores resources that do not require the restorer blob, such as threads. This includes the enclave threads that were exported during

the checkpoint phase. CRIU then injects the restorer blob into its process memory and uses this blob to restore the memory from the saved process image.

Restoring an enclave with CRIU. Once the destination process's memory has been restored, the restorer blob needs to create and restore the destination enclave. COMIT-SGX uses a similar mechanism as for the parasite code in the checkpoint phase in that the restorer blob accesses a file at a predefined location and reads a list of function pointers to invoke. As described above, this file of function pointers was created during the preparation phase. Since the destination process does not yet contain an enclave, the restorer blob first invokes the function to create an enclave and the `ECALL` to initialize important variables (e.g., encryption context) within the enclave. It then creates one or more placeholder enclave threads, corresponding to the number of enclave threads that were interrupted. Each of these threads is then interrupted and destroyed once outside the enclave. The restorer blob then invokes the `enclave_import_all` `ECALL` (via the corresponding wrapper function). As discussed in Section 3.5.1, this `ECALL` restores the exported enclave data to the correct location.

Cleanup Phase

In addition to the steps described in Section 3.5.1, the injected restorer blob must be removed from the destination process's memory. Finally, the restored process is released from its seized state and starts resuming its execution. Once the restored process is resumed, the restored enclave thread can re-enter the enclave and continue execution.

3.6 Evaluation

We evaluated COMIT-SGX to understand the overheads and performance impact of migrating in-process TEEs. In §3.6.1 we first describe the settings of the environment and benchmarks used for the evaluation. Next, we report the throughput of an application run-

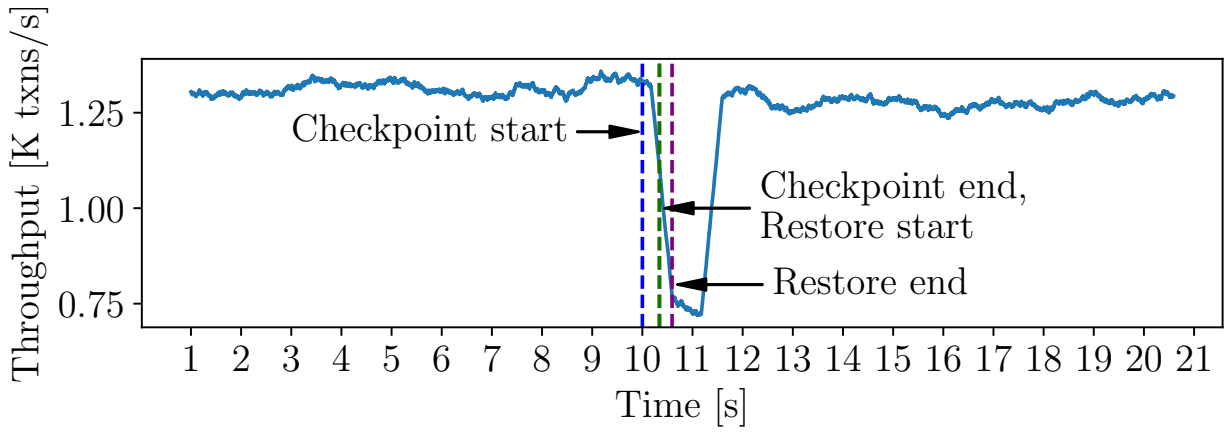
ning in an enclave and the performance impact of when it is migrated (§3.6.2). Then, we show the overall latency of migrating both the host application and enclave (§3.6.3). Next, we report the latency of migrating just the enclave (§3.6.4) and the host application (§3.6.5). Finally, we consider the cryptographic operations used in COMIT-SGX (§3.6.6).

3.6.1 Experimental Setup

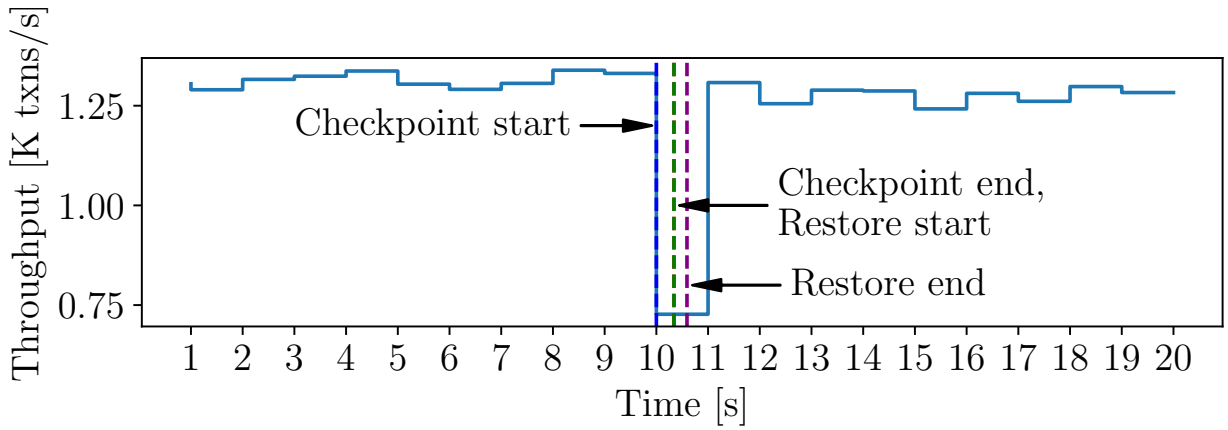
We conducted our experiments using a Microsoft Azure Confidential Computing DC4ds v3 series VM. The VM had 4 Intel Xeon Platinum 8370C vCPUs (Icelake, 2.80 GHz) with 32 GB of assigned memory of which 16 GB could be used by the SGX TEE's enclave page cache (EPC). We ran Ubuntu 18.04 and installed the Intel SGX DCAP Driver version 1.33.2. To remove disk I/O latency from the critical path, we used a RAM disk to store images created by COMIT-SGX's checkpoint phase.

Implementation. We built our implementation of COMIT-SGX in 913 lines of C/C++ code of which 96 lines were additions or modifications to the Open Enclave SDK and 204 lines were additions or modifications to CRIU. COMIT-SGX was integrated into CRIU version 3.15 and Open Enclave version 0.16.1. All cryptography was performed using either mbedTLS version 2.16.10 or OpenSSL version 1.1.1k and COMIT-SGX used OpenSSL unless otherwise specified.

Benchmarks. The throughput evaluation used the *SmallBank* benchmark [7] which models a bank with $1,000$ accounts. The clients perform one of five randomly selected transactions that either: deposit funds; transfer funds; withdraw funds; check an account's balance; or amalgamate two accounts. We implemented the SmallBank benchmark with SQLite version 3.34.1. We ran the benchmark for 10 seconds allowing it to reach a steady state before migrating it to a different process and then ran it for another 10 seconds ensuring the migrated process also reached a steady state.



(a) The blue line is the average number of transactions in the previous second. The vertical lines show when certain migration operations occurred.



(b) The blue plot shows the actual number of transactions that occurred per second. Vertical lines show the point of time where certain migration operations occur.

Figure 3.4: Migration affecting throughput of application with a 64 MB enclave running SmallBank benchmark.

3.6.2 Throughput

We first explored how the throughput of an application is affected when it is migrated. We created an in-memory SQLite database inside a 64 MB size enclave and manipulated the data within the database by executing the SmallBank benchmark [7].

Figure 3.4a shows the result. The blue plot shows the number of transactions executed per second on a one-second rolling average, which calculates the average of transactions in the previous second. We plot our results starting after the first second and advance

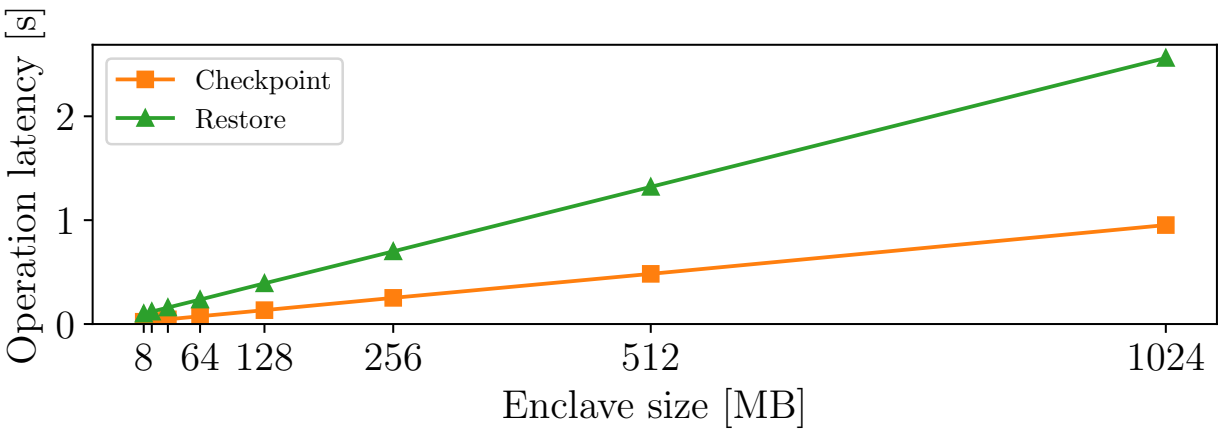


Figure 3.5: Latency of checkpointing/restoring host application and enclave when varying enclave size. Enclave runs a simple counter application.

in 1-millisecond increments. The vertical blue, green, and purple dashed line shows the time points where the checkpoint starts, checkpoint ends and restore starts, and restore ends, respectively. We can see that the throughput of nearly $1,300$ transactions per second (txns/s) dips to 750 txns/s as the migration happens, and recovers after around $1,400$ ms. In this benchmark, when we consider the number of transaction executed per second, the rolling average of executed SmallBank transactions never dips to zero. This shows that most applications – which typically consider their throughput at a per-second granularity – will never experience an instance when their application is unavailable.

We further consider a more course-gained concept of availability in Figure 3.4b. This figure shows the case where we plot the sum of the number of transactions recorded per second. We see a similar trend as Figure 3.4a, where the number of executed transactions in a window is approximately $1,300$ txns/s, dips to appropriately 750 txns/s, and recovers back the next second. Thus, we observe that a window of unavailability – where throughput is 0 txns/s – does not exist and the window of reduced throughput is only one second.

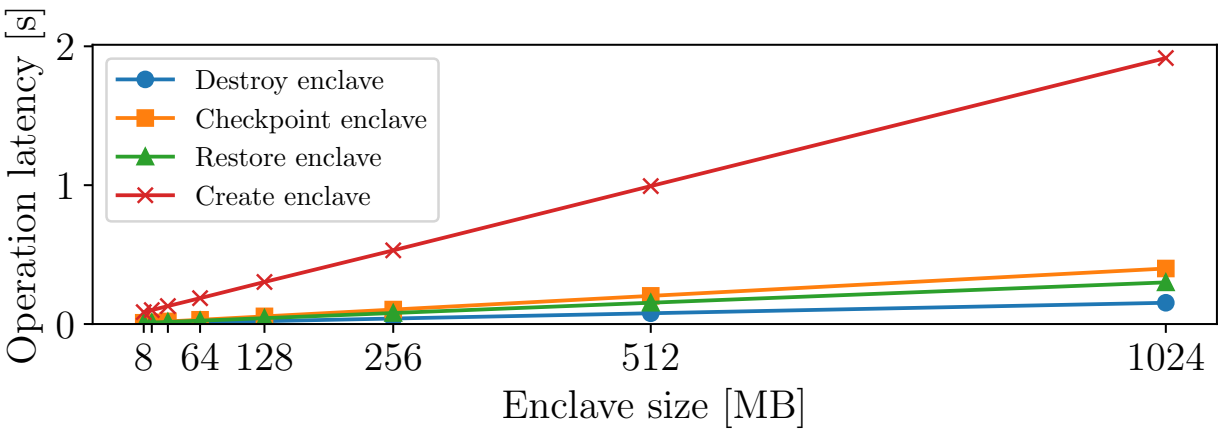


Figure 3.6: Latency of enclave migration operations when varying enclave size. OpenSSL is used to encrypt/decrypt enclave data. Checkpoint and Destroy enclave operations occur during checkpoint phase, while Create and Restore enclave operations occur during restore.

3.6.3 Migration Latency

Now, we look at the impact of migrating enclaves with an increasing amount of memory allocated to the enclave and report all results as the average of 10 measurements. We first show the overall latency of migrating both the host application and enclave.

We measured the change in migration latency when varying the enclave size from 8 MB up to 1024 MB, increasing in two folds. The enclave sizes were chosen based on a survey conducted by Guerreiro et al. [114] on various projects that include enclaves, in which they concluded that enclave sizes between 1 MB and 1 GB represent many development scenarios. The enclave ran an application that incremented a counter from 0 to $1,000,000,000$. For this evaluation, we used OpenSSL (see § 3.6.6 for a comparison of cryptographic libraries). We use this benchmark to understand the overall latency of COMIT-SGX with the side-effects that a more complex workload such as SmallBank may introduce.

Figure 3.5 shows the time required for COMIT-SGX to checkpoint or restore the host application and the enclave when the enclave size changes. We can see that the restore phase takes the most time (2.55 sec for a 1 GB enclave) while checkpoint takes less time (0.95 sec). We also observed that the latency for both phases increases linearly with the enclave

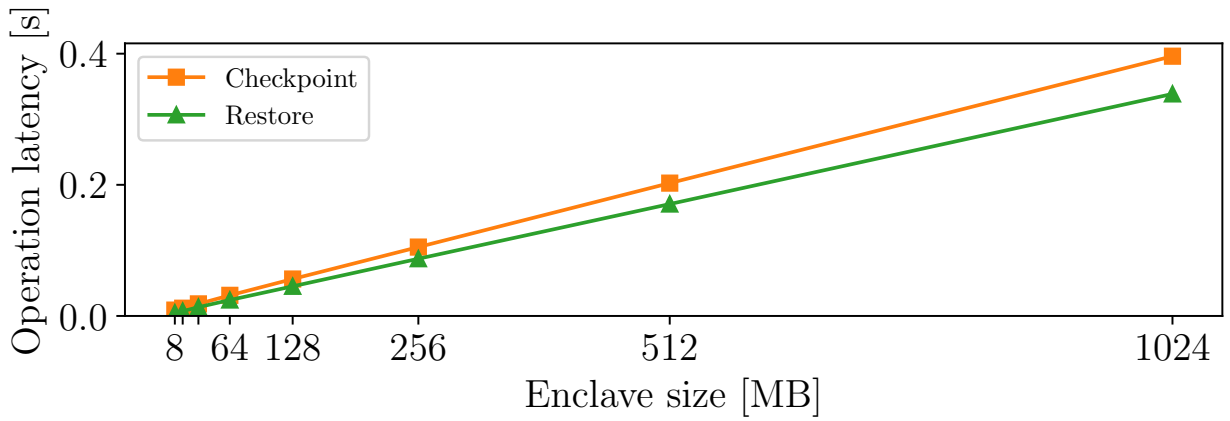


Figure 3.7: Latency of checkpointing/restoring only the host application. The same amount of memory needed to store encrypted enclave data is allocated by the host application and is migrated with the application.

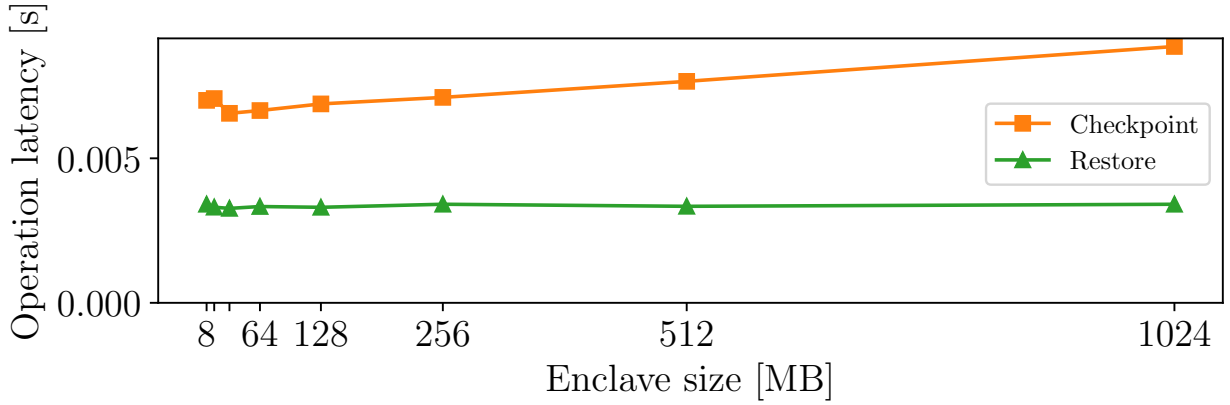


Figure 3.8: Latency of checkpointing/restoring only the host application. No memory buffer is allocated.

size. Since modern Icelake CPUs have an EPC that is half the available system memory, we expect this linear relationship between the enclave's memory and migration time to be consistent until memory is exhausted. This allows COMIT-SGX users to estimate the time required to migrate their enclave.

3.6.4 Enclave Migration Latency

Next, we consider the overhead introduced when migrating just the enclave when varying enclave size. We measure key enclave operations, namely checkpoint enclave, destroy enclave,

create enclave, and restore enclave. Note that (1) and (2) occur during the checkpoint phase and (3) and (4) occur during the restore phase. These operations were chosen based on preliminary observations conducted before this evaluation.

The results are shown in Figure 3.6. We can observe that creating an enclave takes the most time (1.91 sec for 1 GB enclave), followed by checkpoint (0.40 sec), restore (0.30 sec), and destroy enclave (0.15 sec). The reason that creating an enclave takes the longest is because of the allocation and initialization of enclave pages. We can see that the restore phase is the largest contributor to the latency we observed in Figure 3.5. The other essential enclave migration operations (checkpoint and restore enclave) introduce minimal latency.

Another point of consideration is that the create enclave latency is included in the overall latency strictly because the destination enclave is created during the migration operation. We emphasize that this would change according to migration policies, e.g., if the policy requires the destination enclave to be created before or during migration, this latency would not occur in the critical path. We could further reduce the latency in Figure 3.6 by employing multiple threads when checkpointing/restoring enclave data. Therefore, the numbers reported here represent the *upper bound*, where all migration operations are done sequentially.

3.6.5 Host Migration Latency

Next, we look at the overheads introduced when migrating only the host application. In this evaluation, we migrated only the host application and we did not create an enclave in the application. Since the host application allocates and initializes a buffer required to store encrypted enclave data in COMIT-SGX, we also wanted to observe whether this has an impact on latency. Therefore, although the application does not create an enclave during this evaluation, the application still allocated and initialized the buffer.

Figure 3.7 shows the results. We can see that the checkpoint phase takes the longest (0.4 sec

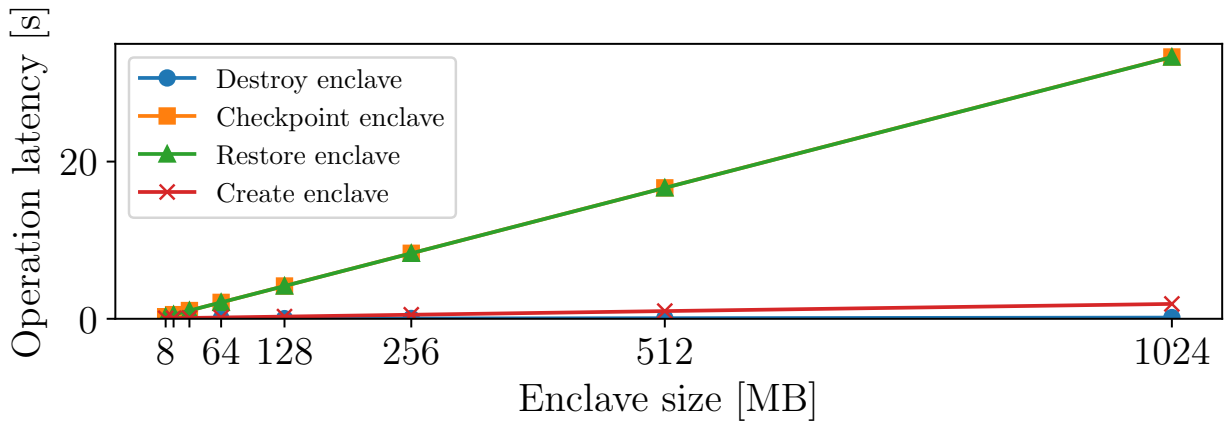


Figure 3.9: Latency of key enclave migration operations when varying enclave size. mbedTLS is used to encrypt/decrypt enclave data. Checkpoint and Destroy enclave operations occur during checkpoint phase, Create and Restore enclave operations occur during restore.

for a 1 GB buffer), followed by restore (0.35 sec). From this we can draw several conclusions: First, the time required to allocate and initialize the buffer required to store encrypted enclave data is significant. This conclusion is further validated by comparing the results from Figure 3.7 to the results from Figure 3.8 where COMIT-SGX does not allocate the buffer. Figure 3.8 shows that the latency of migrating a process without the memory buffer does not grow as the enclave's memory increases. Second, the checkpoint/restore latency of a host application is near-identical to that of an enclave. This shows COMIT-SGX can migrate both a host application and its enclave by doubling the latency of CRIU and that our implementation of in-process TEE migration has the same latency overhead as a state-of-the-art non-TEE process migration utility.

3.6.6 Latency vs. Crypto library

Finally, we investigate how different cryptographic libraries affect enclave migration latency. Here we use two different cryptographic libraries supported by OpenEnclave; mbedTLS and OpenSSL. Additionally, we measure the latency without encrypting/decrypting the enclave data during the checkpoint/restore phase to show the overhead introduced by cryptographic operations. The evaluation followed the same methodology as Section 3.6.3, where we vary

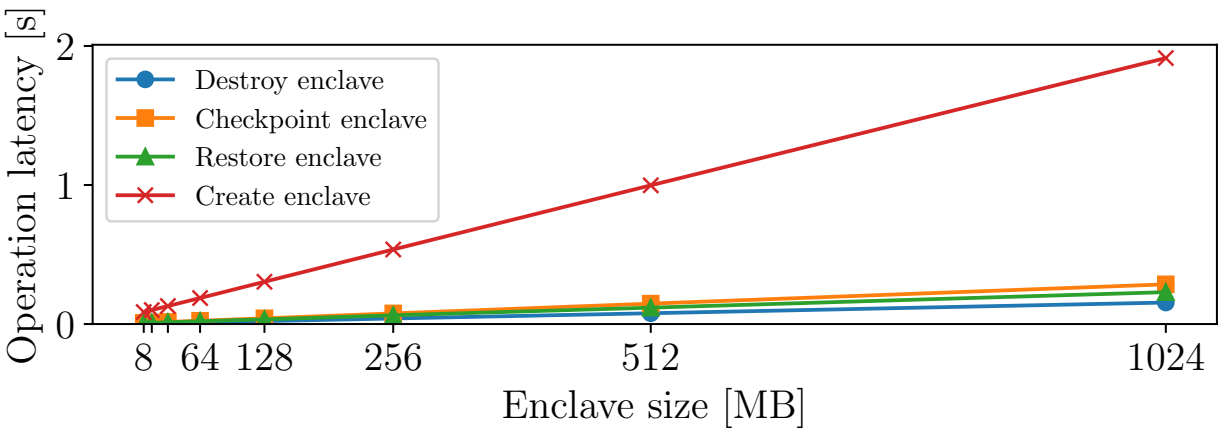


Figure 3.10: Latency of key enclave migration operations when varying enclave size. Enclave data is copied outside without any encryption. Checkpoint and Destroy enclave operations occur during checkpoint phase, Create and Restore enclave operations occur during restore.

the enclave size and measure the time taken to migrate an enclave.

Figures 3.6 and 3.9 show the latency of enclave migration when using OpenSSL and mbedTLS, respectively. We see that OpenSSL outperforms mbedTLS by two orders of magnitude, as mbedTLS takes nearly 33 seconds to checkpoint and restore an enclave, while it takes 0.4 seconds to checkpoint and 0.3 seconds to restore an enclave using OpenSSL. This is validated when comparing Figures 3.6 and 3.10, as the overhead introduced by OpenSSL is in the order of milliseconds, not seconds.

We observed from this evaluation it is preferable to use OpenSSL – which utilizes specialized CPU instruction to perform the encryption/decryption – whenever possible. MbedTLS should only be used if there are no specialized CPU instructions to accelerate cryptographic operations as mbedTLS introduces less code into the enclave's trusted code base. This allowed us to conclude that the performance of enclave migration is dependant on the encryption and decryption performance. Thus, we expect that, as more cryptographic operations are optimized and offloaded to hardware accelerators, the migration latency of COMIT-SGX will decrease.

3.7 Related Work

TEE migration. Currently, there are only two TEE architectures (AMD SEV [12], SEV-SNP [13], and Intel TDX [126]) that have publicly announced native support for migrating their VM-based TEEs. Park et al. and Gu et al. proposed systems that enable TEEs with Intel SGX enclaves to be migrated. The drawback of these systems is that they either require native hardware support or TEEs to be aware and coordinate with the migration operator. In addition, they target VM-based TEEs, not in-process ones.

TEEnder [114] is the only TEE migration system which is aware of that targets in-process TEEs. However, it uses Hardware Security Modules (HSMs) to encrypt and decrypt SGX enclave data during migration. This hardware is not standard on commodity cloud servers and thus this reduces which environments their system can be used in as well as its performance.

TEE persistent state migration. Alder et al. [4] proposed a design for including persistent state (e.g., sealed data, hardware monotonic counter values) when migrating Intel SGX enclaves. Support for migrating such data is important, as migrated enclaves that rely on these persistent state will not be able to continue normal operation at their migration destination. Moreover, not being able to migrate hardware monotonic counter values will undermine the security of the system, as the enclave will be susceptible to roll-back attacks. We consider this as complementary work and envision COMIT working in conjunction with this system.

ReplicaTEE [212] considers another aspect of TEE migration: provisioning of newly instantiated TEEs. Although ReplicaTEE does not support replicating TEEs with their internal state intact, it allows server operators to start up multiple instances of the same enclave and provision them with the same secret without interacting with the application owner. ReplicaTEE is designed so that a limit is imposed on the number TEE instances that can be created. COMIT can utilize ReplicaTEE to limit the number of destination TEEs that are

created while allowing such TEEs to be provisioned with necessary secrets (e.g., migration key).

3.8 Conclusion

In this work, we proposed COMIT, a software-only design to enable migration functionality into current in-process TEE architectures without hardware modifications. COMIT allows TEEs to be migrated at arbitrary points in their execution, allowing our method to be integrated into existing process migration tools. We implemented COMIT for Intel SGX and CRIU, and show that migration latency increases linearly with the size of the TEE and is primarily dependent on the time required to create and initialize the destination TEE.

Chapter 4

PDoT: Private DNS-over-TLS with TEE Support

Abstract

Security and privacy of the Internet Domain Name System (DNS) have been long-standing concerns. Recently, there is a trend to protect DNS traffic using Transport Layer Security (TLS). However, at least two major issues remain: (1) how do clients authenticate DNS-over-TLS endpoints in a scalable and extensible manner; and (2) how can clients trust endpoints to behave as expected? In this chapter, we propose a novel Private DNS-over-TLS (*PDoT*) architecture. *PDoT* includes a DNS Recursive Resolver (RecRes) that operates within a Trusted Execution Environment (TEE). Using *Remote Attestation*, DNS clients can authenticate and receive strong assurance of trustworthiness of *PDoT* RecRes. We provide an open-source proof-of-concept implementation of *PDoT* and experimentally demonstrate that its latency and throughput match that of the popular Unbound DNS-over-TLS resolver.

Research presented in this chapter appeared in the Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC 2019) [177] and in Digital Threats: Research and Practice, Volume 2, Issue 1 (DTRAP 2021) [178].

4.1 Introduction

The Domain Name System (DNS) [169] is a global distributed system that translates human-readable domain names into IP addresses. It has been deployed since 1983 and, throughout the years, DNS privacy has been a major concern.

In 2015, Zhu et al. [241] proposed a DNS design that runs over Transport Layer Security (TLS) connections [76]. DNS-over-TLS protects privacy of DNS queries and prevents man-in-the-middle (MiTM) attacks against DNS responses. [241] also demonstrated the practicality of DNS-over-TLS in real-life applications. Several open-source recursive resolver (RecRes) implementations, including Unbound [144] and Knot Resolver [67], currently support DNS-over-TLS. In addition, commercial support for DNS-over-TLS has been increasing, e.g., Android P devices [105] and Cloudflare's `1.1.1.1` RecRes [61]. However, despite attracting interest in both academia and industry, some problems remain.

The first challenge is that clients need a way to authenticate the RecRes. Certificate-based authentication is natural for websites, since the user (client) knows the URL of the desired website and the certificate securely binds this URL to a public key. However, this approach cannot be used for a DNS RecRes because the RecRes does not have a URL or any other unique long-term user-recognizable identity that can be included in the certificate. One way to address this issue is to provide clients with a white-list of trusted RecRes-s' public keys. However, this is neither scalable nor maintainable, because the white-list would have to include all possible RecRes operators, ranging from large public services (e.g., `1.1.1.1`) to small-scale providers, e.g., a local RecRes provided by a coffee-shop.

Even if the RecRes can be authenticated, the second major issue is the lack of means to determine whether a given RecRes is trustworthy. For example, even if communication between client stub (client) and RecRes, and between RecRes and the name server (NS) is authenticated and encrypted using TLS, the RecRes must decrypt the DNS query in order to

resolve it and contact the relevant NS-s. This allows the RecRes to learn unencrypted DNS queries, which poses privacy risks of a malicious RecRes misusing the data, e.g., profiling users or selling their DNS data. Some RecRes operators go to great lengths to assure users that their data is private. For example, Cloudflare promises *"We will never sell your data or use it to target ads"* and goes on to say *"We've retained KPMG to audit our systems annually to ensure that we're doing what we say"* [61]. On March 31, 2020, Cloudflare released the results of the privacy examination of its `1.1.1.1` DNS resolver [59]. The report is publicly available [57] and it assures that during the time of inspection (conducted from February 1, 2019 to October 31, 2019), `1.1.1.1` was configured in such a way that it supports the commitment given by Cloudflare. While this report gives some guarantee that the `1.1.1.1` resolver is honoring the client's privacy, there are three drawbacks of this method. First, it only provides a guarantee to a particular point in time. We cannot be certain whether the privacy promise was kept before and after the inspection. Second, it takes a long time to conduct the inspection and release the report. `1.1.1.1` was announced on April 1, 2018 [58], *two* years before the privacy report was released. Third, this method requires users to trust the auditor and can only be used by operators who can afford an auditor. Although `1.1.1.1` may be one of the famous public resolvers that support DNS-over-TLS and DNS-over-HTTPS, a recent study shows that there are many smaller organizations that also provide such resolvers [153]. Since these organizations cannot afford to be inspected, it is more difficult for them to convince their customers that they are protecting their privacy.

The work presented in this chapter uses Trusted Execution Environments (TEEs) and *Remote Attestation* (RA) to address these two problems. By using RA, the identity of the RecRes is no longer relevant, since clients can check what software a given RecRes is running and make trust decisions based on how the RecRes behaves. RA is one of the main features of modern hardware-based TEEs, such as Intel Software Guard Extensions (SGX) [159] and ARM TrustZone [23]. Such TEEs are now widely available, with Intel CPUs after the 7th generation supporting SGX, and ARM Cortex-A CPUs supporting TrustZone. TEEs with

RA capability are also available in cloud services, such as Microsoft Azure [166]. Specifically, our contributions are:

- We design a Private DNS-over-TLS (*PDoT*) architecture, the main component of which is a privacy-preserving RecRes that operates within a commodity TEE. Running the RecRes inside a TEE prevents even the RecRes operator from learning clients' DNS queries, thus providing query privacy. Our RecRes design addresses the authentication challenge by enabling clients to trust the RecRes based on how it behaves, and not on who it claims to be. (See Section 4.4).

- We implement a proof-of-concept *PDoT* RecRes using Intel SGX and evaluate its security, deployability, and performance. All source code and evaluation scripts are publicly available [142]. Our results show that *PDoT* handles DNS queries without leaking information while achieving sufficiently low latency and offering acceptable throughput (See Sections 4.5 and 4.7).

- In order to quantify privacy leakage via traffic analysis, we performed an Internet measurement study. It shows that 94.7% of the top $1,000,000$ domain names can be served from a *privacy-preserving* NS that serves at least two distinct domain names, and 65.7% from an NS that serves 100+ domain names. (See Section 4.8).

4.2 Background

4.2.1 Domain Name System (DNS)

DNS is a distributed system that translates host and domain names into IP addresses. DNS includes three types of entities: *Client Stub* (client), *Recursive Resolver* (RecRes), and *Name Server* (NS). Client runs on end-hosts. It receives DNS queries from applications, creates DNS request packets, and sends them to the configured RecRes. Upon receiving a request,

RecRes sends DNS queries to NS-s to resolve the query on client's behalf. When NS receives a DNS query, it responds to RecRes with either the DNS *record* that answers client's query, or the IP address of the next NS to contact. RecRes thus recursively queries NS-s until the record is found or a threshold is reached. The NS that holds the queried record is called: *Authoritative Name Server* (ANS). After receiving the record from ANS, RecRes forwards it to client. It is common for RecRes to cache records so that repeated queries can be handled more efficiently.

4.3 Adversary Model & Requirements

4.3.1 Adversary Model

The adversary's goal is to learn, or infer, information about DNS queries sent by clients. We consider two types of adversaries, based on their capabilities:

The first type is a malicious RecRes operator who has full control over the physical machine, its OS, and all applications, including the RecRes. We assume that the adversary cannot break any cryptographic primitives, assuming that they are correctly implemented. We also assume that it cannot physically attack hardware components, e.g., probe the CPU to learn TEE secrets. This adversary also controls all of the RecRes' communication interfaces, allowing it to drop/delay packets, measure the time required for query processing, and observe all cleartext packet headers.

The second type is a network adversary, which is strictly weaker than the malicious RecRes operator. In the passive case, this adversary can observe any packets that flow into and out of RecRes. In the active case, this adversary can modify and forge network packets. Note that this represents the strongest form of network adversary who can observe and potentially manipulate both the *downstream* (i.e., client – RecRes) and *upstream* (i.e., RecRes– NS) communication. In the common case, the network adversary would only be able to observe

the downstream communication (e.g., the adversary is another user connected to the same wireless network as the victim). DNS-over-TLS alone (without *PDoT*) is sufficient to thwart a passive network adversary. However, since an active adversary could redirect clients to a malicious RecRes, clients need an efficient mechanism to authenticate the RecRes and determine whether it is trustworthy, which is one of the main contributions of *PDoT*.

We do not consider Denial-of-Service (DoS) attacks on RecRes, since these do not help to achieve either adversary's goal of learning clients' DNS queries. Connection-oriented RecRes-s can defend against DoS attacks using cookie-based mechanisms to prevent SYN flooding [241].

4.3.2 System Requirements

We define the following requirements for the overall system:

R1: Query Privacy. Contents of client's query (specifically, domain name to be resolved) should not be learned by the adversary. Ideally, payload of the DNS packets should be encrypted. However, even if packets are encrypted, their headers leaks information, such as source and destination IP addresses. In Section 4.8.1, we quantify the amount of information that can be learned via traffic analysis.

R2: Deployability. Clients using a privacy-preserving RecRes should require no special hardware. Minimal software modifications should be imposed. Also, for transition and compatibility, a privacy-preserving RecRes should be able to interact with legacy clients that only support unmodified DNS-over-TLS.

R3: Response Latency. A privacy-preserving RecRes should achieve similar response latency to that of a regular RecRes.

R4: Scalability. A privacy-preserving RecRes should process a realistic volume of queries

generated by a realistic number of clients.

Note: Query privacy guarantees provided by *PDoT* rely on the forward-looking assumption that communication between RecRes and respective NS-s is also protected by DNS-over-TLS. The DNS Privacy (DPrive) Working Group is working towards a standard for encryption and authentication of DNS resolver-to-ANS communication [36], using essentially the same mechanism as DNS-over-TLS. We expect an increasing number of NS-s to begin supporting this standard in the near future. Once *PDoT* is enabled at the RecRes, it can provide incremental query privacy for queries served from a DNS-over-TLS NS. As discussed in Section 4.5, with small design modifications, *PDoT* can be adapted for use in NS-s. Additionally, until DNS-over-TLS becomes the norm for protecting communications between the RecRes and NS, *PDoT* can be modified to support DNSSEC [20]. Although DNSSEC does not provide DNS query privacy, it provides a means for *PDoT* to authenticate the NS.

4.4 System Model & Design Challenges

4.4.1 PDoT System Model

Figure 4.1 shows an overview of *PDoT*. It includes four types of entities: client, RecRes, TEE, NS-s. We now summarize *PDoT* operation, reflected in the figure: (1) After initial start-up, TEE creates an attestation report. (2) When client initiates a secure TLS connection, the attestation report is sent from RecRes to the client alongside all other information required to setup a secure connection. (3) Client authenticates and attests RecRes by verifying the attestation report. It checks whether RecRes is running inside a genuine TEE and running trusted code. (4) Client proceeds with the rest of the TLS handshake procedure only if verification succeeds. (5) Client sends a DNS query to RecRes through the secure TLS channel it has just set up. (6) RecRes receives a DNS query from client, decrypts it into TEE memory, and learns the domain name that the client wants to resolve. (7) RecRes sets

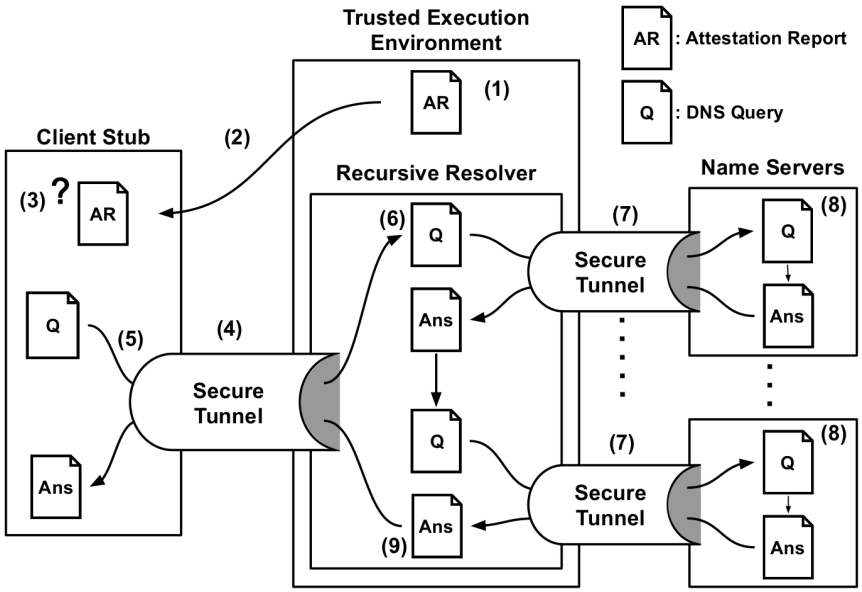


Figure 4.1: Overview of the proposed system.

up a secure TLS channel to the appropriate NS in order to resolve the query. (8) RecRes sends a DNS query to NS over that channel. If NS's reply includes an IP address of the next NS, RecRes sets up another TLS channel to that NS. This is done repeatedly until RecRes successfully resolves the name to an IP address. (9) Once RecRes obtains the final answer, it sends this to client over the secure channel. Client can reuse the TLS channel for future queries.

Note that we assume RecRes is not under the control of the user. In some cases, users could run their RecRes-s, which would side-step the concerns about query privacy. For example, modern home routers are sufficiently powerful to run an in-house RecRes. However, this approach cannot be used in public networks (e.g., airports or coffee shop WiFi networks), which are the target scenarios for *PDoT*.

4.4.2 Design Challenges

The following key challenges were encountered in the process of *PDoT*'s design:

C1: TEE Functionality Limitations. In order to satisfy their security requirements,

current TEEs (e.g., SGX and TrustZone) often limit the functionality available to code that runs within them. One example is the inability to fork within the TEE. Forking a process running inside the TEE forces the child process to run outside the TEE, breaking RecRes security guarantees. Another example is that system calls, such as socket communication, cannot be made from within the TEE.

C2: TEE Memory Limitations. A typical TEE has a relatively small amount of memory. Although an SGX enclave can theoretically have a large amount of in-enclave memory, this will require page swapping of EPC pages. The pages to be swapped must be encrypted and integrity protected in order to meet the security requirements of SGX. Therefore, page swapping places a heavy burden on performance. To avoid page swapping, enclave size should be less than the size of the EPC – typically, 128MB. Since RecRes is a performance-critical application, its size should ideally not exceed 128MB. This limit negatively impacts RecRes throughput, as it bounds the number of threads that can be spawned in a TEE.

C3: TEE Call-in/Call-out Overhead. Applications requiring functionality that is not available within the TEE must switch to the non-TEE side. This introduces additional overhead, both from the switching itself and from the need to flush and reload CPU caches. Identifying and minimizing the number of times RecRes switches back and forth (while keeping RecRes functionality correct) is a substantial challenge.

4.5 Implementation

Figure 4.2 shows an overview of the *PDoT* design. Since our design is architecture-independent, it can be implemented on any TEE architecture that provides the features outlined in Chapter 2.1. We chose the off-the-shelf Intel SGX as the platform for the proof-of-concept *PDoT* implementation in order to conduct an accurate performance evaluation on real hardware

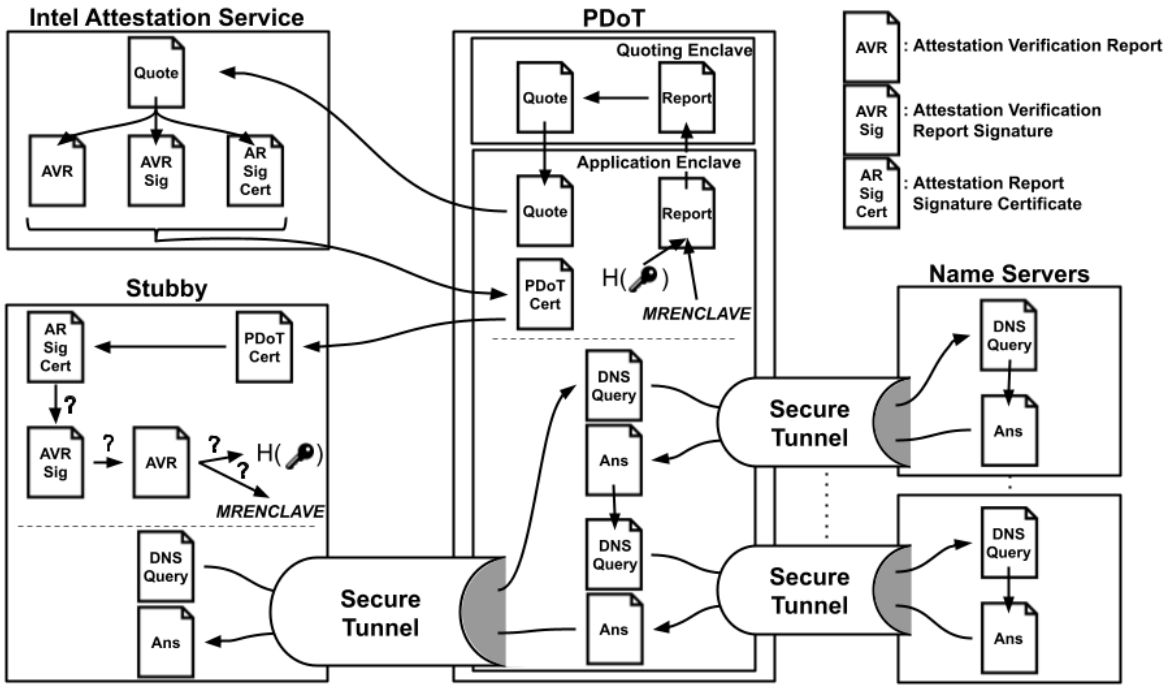


Figure 4.2: Overview of *PDoT* implementation.

(see Section 4.7). Therefore, our implementation is subject to performance and memory constraints in the current version of Intel SGX. It is thus best suited for small-scale networks, e.g., a WiFi hotspot provided by a typical coffee shop. However, as TEE technology advances, we expect that our design will scale to larger networks.

4.5.1 PDoT

PDoT consists of two parts: (1) a trusted part residing in TEE enclaves, and (2) untrusted part that operates elsewhere. The former is responsible for resolving DNS queries, and the latter – for accepting incoming connections, assigning file descriptors to sockets and sending/receiving data received from the trusted part.

Enclave Startup Process. When the application enclave starts, it generates a new public-private key-pair within the enclave. It then creates a *report* that summarizes enclave and platform state. The report includes a SHA256 hash of the entire code that is supposed to run in the enclave (called *MRENCLAVE* value) and other attributes of the target enclave.

PDoT also includes a SHA256 hash of the previously generated public key in the report. The report is then passed on to the SGX quoting enclave to receive a *quote*. The quoting enclave signs the report and thus generates a quote, which cryptographically binds the public key to the application enclave. The quoting enclave sends the quote to the application enclave, which forwards it to the Intel Attestation Service (IAS) to obtain an *attestation verification report*. It can be used in the future by clients to verify the link between the public key and the MRENCLAVE value. After receiving the attestation verification report from IAS, the application enclave prepares a self-signed X.509 certificate required for the TLS handshake. In addition to the public key, the certificate includes (1) attestation verification report, (2) attestation verification report signature, and (3) attestation report signing certificate, extracted from (1). The MRENCLAVE value and hash of public key are enclosed in the attestation verification report.

TLS Handshake Process.[1] Once the application enclave is created, *PDoT* can create TLS connections and accept DNS queries from clients. The client initiates a TLS handshake process by sending a message to *PDoT*. This message is captured by untrusted part of *PDoT* and triggers the following events. [2] First, untrusted part of *PDoT* tells the application enclave to create a new TLS object within the enclave for this incoming connection. This forces the TLS endpoint to reside inside the enclave. The TLS object is then connected to the socket where the client is waiting to be served. The RecRes then exchanges several messages with the client, including the self-signed certificate that was created in the previous section. Having received the certificate from RecRes, the client authenticates RecRes and validates the certificate (see Section 4.5.2). Only if the authentication and validation succeed does the client resume the handshake process.

DNS Query Resolving Process. The client sends a DNS query over the TLS channel

[1]This design is derived from the SGX RA-TLS [139] whitepaper.

[2]Since we consider a malicious RecRes operator, it has an option not to trigger these events. However, clients will notice that their queries are not being answered and can switch to a different RecRes.

established above. Upon receiving the query, RecRes decrypts it within the application enclave and obtains the target domain name. The RecRes begins to resolve the name starting from root NS, by doing the following repeatedly: 1) set up a TLS channel with NS, 2) send DNS queries and receive replies via that channel. Once the RecRes receives the answer from the NS, the RecRes returns it to the client over the original TLS channel.

Figure 4.3 illustrates the threading model of *PDoT*. For each client, we spawn two threads, a *ClientReader* and one or more *QueryHandler* threads.

The *ClientReader* thread is responsible for accepting incoming connections from a client and performing the TLS handshake. Once a TLS connection has been established, this thread reads the incoming DNS queries and stores them in client-specific FIFO queues — the *QueryLists* in Figure 4.3.

The *QueryHandler* threads are responsible for performing the recursive resolution of queries and sending the responses to the clients. Each *QueryList* has one or more *QueryHandler* threads associated with it. Using the thread synchronization primitives in the SGX SDK, the *QueryHandler* threads wait on a condition until they are signaled by the *ClientReader* thread to indicate that there is a pending query in the *QueryList*. Once signaled, a *QueryHandler* thread first checks whether the client is still accepting responses from RecRes, and if so, resolves the query and sends the response via the established TLS connection.

In our implementation, we only use one *QueryHandler* thread per client, in order to maximize the number of concurrent client connections we can support. Due to the limitations on enclave memory, only a limited number of threads can execute within the enclave concurrently. This is controlled by the number of Thread Control Structure (TCS) data structures allocated during enclave compilation. Additionally, by using only a single *QueryHandler* thread per client, this thread can be given exclusive write access to the client's TLS connection, thus avoiding the need for costly thread synchronization.

Having a dedicated *ClientReader* thread to read and buffer the incoming queries from a single client enables PDoT to *receive* concurrent requests. This allows a client to send multiple DNS queries within a short timespan without waiting for the answers to previous queries. For example, when a client loads a webpage that includes images and advertisements from different domains, multiple DNS queries are triggered at the same time. However, because we chose to use one *QueryHandler* thread per client, PDoT does not *resolve* the concurrent requests it received from a single client. In comparison to alternative designs (described below), we observed that this approach provides higher throughput when multiple clients are connected (see Section 4.7.4).

We previously implemented an alternative design using a per-client dedicated *ClientWriter* thread to buffer responses. However, we found that this increased the number of concurrent threads in the enclave without providing a performance benefit. Moreover, using a dedicated *ClientWriter* thread introduced additional queues and therefore required additional thread synchronization variables.

We also previously implemented another alternative design using a single *QueryList* shared between multiple *ClientReader* threads and multiple *QueryHandler* threads. This allowed *ClientReader* threads to share multiple *QueryHandler* threads, thus allowing PDoT to resolve multiple DNS queries from a single client concurrently. However, we observed that this causes contention between the *ClientReader* threads when accessing the single *QueryList*, leading to a large variance in query response times and causing some queries to time-out.

Caching. In order to measure the influence of caching, we implemented a simple in-enclave cache for PDoT. It uses a red-black tree data structure and stores all records associated with the clients' queries, indexed by the queried domain. This results in $O(log_2(N))$ access times with N entries in the cache. We discuss the potential privacy risks of enabling caching and propose possible mitigation strategies in Section 4.6.

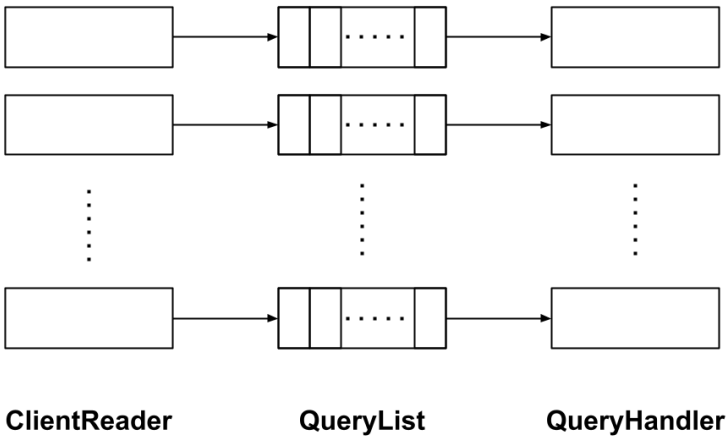


Figure 4.3: Overview of *PDoT* threading model.

PDoT ANS with TEE support. With minor design changes, *PDoT* RecRes design can be modified for use as an ANS. Similar to the caching mechanism described above, an *PDoT* ANS can look up the answers to queries in an internal database, rather than contact external NS-s. The same way that clients authenticate *PDoT* RecRes, the RecRes can authenticate the *PDoT* ANS. Clients can thus establish trust in both RecRes and ANS using *transitive attestation* [3].

4.5.2 Client with PDoT Support

We picked the Stubby client stub from the `getdns` project [143] which offers DNS-over-TLS support and modified it to perform remote attestation during the TLS handshake. We chose to use Stubby as it is the most well-developed, open-sourced DNS-over-TLS client. We now describe how the client verifies its RecRes, decides whether the RecRes is trusted, and emits the DNS request packet.

RecRes Verification. After receiving a DNS request from an application, the client first checks whether there is an existing TLS connection to its RecRes. If so, the client reuses it. If not, it attempts to establish a new connection. During the handshake, the client receives a certificate from RecRes, from which it extracts: 1) attestation verification report,

2) attestation verification report signature, and 3) attestation report signing certificate. This certificate is self-signed by IAS and the client is assumed to trust it. From (3), the client first retrieves the IAS public key and, using it, verifies (2). Then, the client extracts the SHA256 hash of RecRes's public key from (1) and verifies it against (3). This way, the client is assured that RecRes is indeed running in a genuine SGX enclave and uses this public key for the TLS connection.

Trust Decision. The client also extracts the MRENCLAVE value from (1), which it compares against the list of acceptable MRENCLAVE values. If the MRENCLAVE value is not listed or one of the verification steps fails, the client stub aborts the handshake, moves on to the next RecRes, and re-starts the process. Note that the trust decision process is different from the normal TLS trust decision process. Normally, a TLS server-side certificate binds the public key to one or more URLs and organization names. However, by binding the MRENCLAVE value with the public key, the clients can trust RecRes based on its behavior, and not its organization (recall that the MRENCLAVE value is a hash of RecRes code). There are several options for deciding which MRENCLAVE values are trustworthy. For example, vendors could publish lists of expected MRENCLAVE values for their resolvers. For open-source resolvers such as *PDoT*, anyone can re-compute the expected MRENCLAVE value by recompiling the software, assuming a reproducible build process. This would allow trusted third parties (e.g., auditors) to inspect the source code, ascertain that it upholds required privacy guarantees, and publish their list of trusted MRENCLAVE values.

Sending DNS request. Once the TLS connection is established, the client sends the DNS query to RecRes over the TLS tunnel. If it does not receive a response from RecRes within the specified timeout, it assumes that there is a problem with RecRes and sends a DNS reply message to the application with the error code SERVFAIL.

4.5.3 Overcoming Technical Challenges

As discussed in Section 4.4.2, *PDoT* faced three main challenges, which we addressed as follows:

Limited TEE Functionality. The inability to use sockets within the TEE is a challenge because the RecRes cannot communicate with the outside world. We address this issue by having a process running outside the TEE, as described in Section 4.5.1. This process forwards packets from the client to TEE through ECALLs and sends packets received from TEE via OCALLs. However, because it is outside the TEE, this process might redirect the packet to a malicious process or simply drop it. We discuss this issue in Section 4.7.1. Another function unavailable within TEE is forking a process. *PDoT* uses `pthreads` instead of forking to run multiple tasks concurrently in a TEE.

Limited TEE Memory. We use several techniques to address this challenge. First, we ensure no other enclaves (other than the quoting enclave) run on the RecRes SGX machine. This allows *PDoT* to use all available EPC memory. Second, we minimize the number of threads running inside the enclave in order to save space.

OCALL and ECALL Overhead. ECALLs and OCALLs introduce overhead and therefore should be avoided as much as possible. For example, all threads mentioned in the previous section must wait until they receive the following information: for the *ClientReader* thread, the DNS query from the client, and for the *QueryHandler* thread, the query from the QueryList. *PDoT* was implemented so that these threads wait inside the enclave whenever possible to minimize the number of expensive enclave entries and exits.

4.6 Privacy-preserving DNS caching

Some DNS recursive resolvers cache query results and can use these to answer a query directly in the case of a cache hit. Caching is beneficial from the client's perspective because

it reduces query latency. The RecRes also benefits from not having to establish connections to external NS-s. However, if implemented naively, caching at the RecRes may allow adversaries to learn something about the victim's query. In this section, we discuss the potential privacy risks, and show how these can be overcome in *PDoT*.

4.6.1 Attacks exploiting caching

As described in Section 4.3.1, we consider attacks by either a malicious RecRes operator or a network adversary. Both aim to learn information from a naive caching implementation.

Network adversary: As discussed in the previous section, the network adversary cannot see the contents of the victim's query thanks to the TLS connection. However, we assume this adversary can still observe all (encrypted) packets sent and received by the RecRes on both the upstream and downstream communication links. We assume the network adversary can also submit queries to the RecRes and measure the time between queries and responses.

The network adversary's goal is to infer whether a specific query, submitted by either the victim or himself, was answered from the cache. This could be achieved by measuring the time between query and response since a cache hit can be answered relatively quickly. Alternatively, the adversary could observe any (encrypted) recursive queries on the upstream interface, since a cache hit can be answered without consulting upstream NS-s. Knowing whether a query was answered from the cache can leak information to the adversary in two ways:

- The adversary could have primed the cache with specific domains *before* the victim arrives; thus whenever the victim's query is answered from the cache, the adversary can infer that the victim queried one of the primed domains.
- The adversary can probe the cache by submitting queries *after* the victim; thus whenever one of their queries is answered from the cache, the adversary can infer that the

victim had previously queried this domain.

In most cases, these attacks would be complicated by the presence of other users and the inability of the network adversary to clear the RecRes' cache. However, we consider the worst-case scenario, since the network adversary can block packets from other users in order to isolate the victim's traffic.

Malicious RecRes operator: A malicious RecRes operator can perform the same attacks as the network adversary, and can also observe the memory access pattern of the RecRes. In addition to learning whether or not a specific query was answered from the cache, the malicious operator also aims to learn *which* cached response was used. Depending on the type of TEE used to implement *PDoT*, the malicious operator might be able to monitor *PDoT*'s (encrypted) memory accesses. For example, it has been shown that in Intel SGX, an adversary in control of the OS can monitor memory accesses deterministically at page granularity (typically 4 kB) [232], or probabilistically at cache-line granularity [151]. Similarly to the network adversary, this information can be leveraged in two ways:

- The malicious operator could prime the cache with specific domains before the victim arrives, and record the address of each primed entry in the cache. When there is a cache hit, the address of the response reveals exactly which domain the victim queried.

- If there are multiple users, the malicious operator could wait for the users to fill the cache, while recording which users created which cache entries. Afterward, the malicious operator could submit its queries, and if there are any cache hits, the address reveals which user queried for that domain.

4.6.2 Mitigating cache attacks

We propose the following techniques to mitigate the above attacks. In some cases, multiple techniques must be combined. During remote attestation, clients can ascertain which

mitigations the RecRes is using.

A `nocache` bit: Short of disabling caching completely, the impact of the above attacks could be minimized by giving users the choice of whether or not to use the cache on a per-query basis. For *non-sensitive* queries, the cache could be used as normal, while for *sensitive* queries, the users could specify that the query should not be answered from the cache (even if an answer is available) and that the response should not be cached. This could be done by assigning the currently unused `Z` bit in the DNS query header as a `nocache` (`NC`) bit. Historically the `Z` bit was used to indicate that only a response from the primary server for a zone was acceptable [80], which is already very similar to the `nocache` bit. The only addition is that the RecRes would not cache responses to queries with this bit set.

The distinction between sensitive and non-sensitive queries should be made by each user, or their client software (e.g., when browsing in "incognito mode", all DNS queries could be marked as sensitive). Although the adversary would still be able to perform all the above attacks, the impact to the user would be minimal since these would only reveal information about non-sensitive queries. Sensitive queries would be indistinguishable from cache misses. Although less efficient than normal non-privacy-preserving caching, this approach is more efficient than disabling caching completely, while providing similar privacy guarantees.

Delayed responses: In most cases, a realistic network adversary's abilities are weaker than the assumptions we make in Section 4.3.1. Specifically, this adversary would not usually be able to see upstream communication between the RecRes and the NS-s. For example, in the coffee shop scenario, a malicious user on the wireless network may be able to observe the communication between other wireless clients and the RecRes. However, they cannot observe the RecRes' upstream communication, as this would take place via a wired interface. This type of network adversary can thus only infer whether there was a DNS cache hit (either their own or a victim's query) by measuring the time between the query and response.

One possible mitigation is therefore to introduce artificial delay when answering queries from the cache [2]. For example, when there is a cache miss, the RecRes itself can measure the time required to resolve a particular query, and store this time measurement along with the answer in the cache. When this answer is subsequently served from the cache, the RecRes can simply wait for the same amount of time before sending the response to the client. Although this results in higher-latency responses for the clients, it is still beneficial for the RecRes as it reduces the load on the upstream connection. This addresses both types of attacks by the weaker network adversary, although it does not help against the malicious RecRes operator, who can observe whether or not the RecRes makes an upstream query.

Pre-populated cache: On startup and periodically while it is running, the RecRes itself could query for example the top 1,000 most popular domains and store the results in the cache. This mitigates the first of the two possible attacks the network adversary can perform because when there is a cache hit by the victim, the adversary cannot distinguish whether it is for a primed or a pre-populated domain. In other words, this negates the adversary's ability to prime the cache. Note that the network adversary can still probe the cache after the victim's query. The malicious RecRes operator can also bypass this mitigation by observing the RecRes's memory access pattern to distinguish between primed and pre-populated domains.

Oblivious memory accesses: To prevent the malicious RecRes operator from learning which memory address in the cache contained the desired result, we need to make all cache memory accesses oblivious to the operator. This could be used in conjunction with the pre-populated cache mitigation above. Since this is a more general challenge for applications using SGX enclaves, multiple techniques are already available, including general-purpose oblivious RAM (ORAM) schemes (e.g., [205, 64]), and purpose-built oblivious access schemes (e.g., [216]). These techniques could be applied to the caching mechanism described in the previous section.

4.6.3 DNS cache poisoning

Although not an attack on privacy, introduction of a cache also opens *PDoT* to the possibility of DNS cache poisoning attacks, whereby a network adversary causes incorrect answers to be stored in the cache. This could be used as the precursor to a man-in-the-middle attack where the adversary redirects the victim to a malicious IP address, even though the domain appears to be correct.

A network adversary could attempt to poison the cache by pretending to be an upstream NS and responding to upstream queries from the RecRes before the real NS response arrives. Alternatively, the adversary could force the RecRes to query a malicious upstream NS with which the adversary is colluding (e.g., simply by querying the relevant domain). The DNS response from the malicious NS could also include falsified *additional information* records for other domains, which would also poison the cache. These attacks can be mitigated by the RecRes using DNSSEC [20] wherever possible.

4.7 Evaluation

4.7.1 Security Analysis

This section describes how query privacy (Requirement R1) is achieved, concerning the two types of adversaries, per Section 4.3.1.

Malicious RecRes operator. Recall that a malicious RecRes operator controls the machine that runs *PDoT* RecRes. It cannot obtain the query from intercepted packets since they flow over the encrypted TLS channel. Also, because the local TLS endpoint resides inside the RecRes enclave, the malicious operator cannot retrieve the query from the enclave, as it does not have access to the protected memory region.

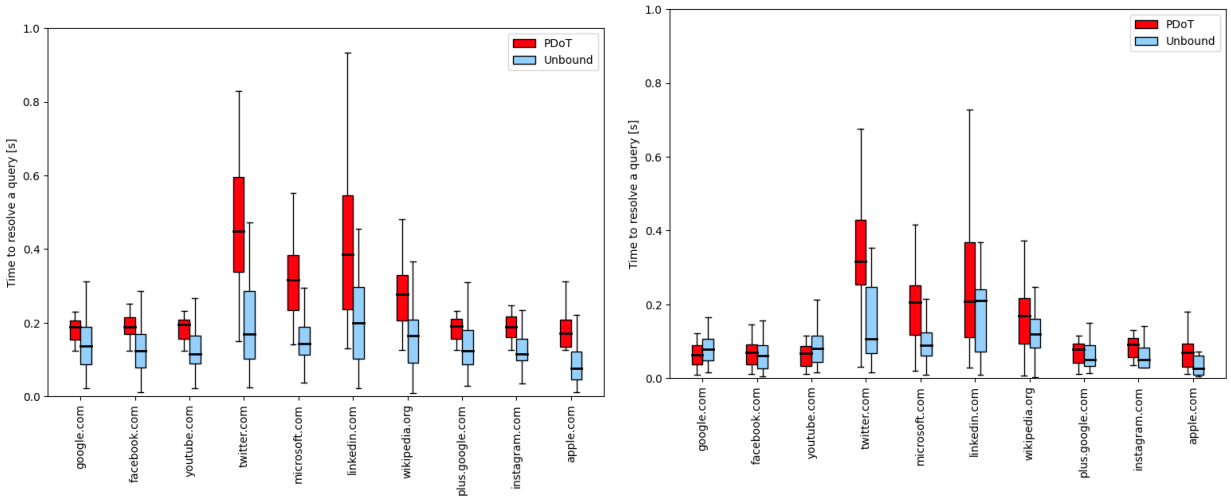
However, a malicious RecRes operator may attempt to connect the socket to a malicious

TLS server that resides in either: 1) an untrusted region or 2) a separate enclave that the operator itself created. If the operator can trick the client into establishing a TLS connection with the malicious TLS server, the adversary can obtain plaintext DNS queries. For case (1), the verification step on the client side fails because the TLS server certificate does not include any attestation information. For case (2), the malicious enclave might receive a legitimate attestation verification report, attestation verification report signature, and attestation report signing certificate from IAS. However, that report would contain a different MRENCLAVE value, which would be rejected by the client. To convince the client to establish a connection with *PDoT* RecRes, the adversary has no choice except to run the code of *PDoT* RecRes. Therefore, in both cases, the adversary cannot trick the client into establishing a TLS connection with a TLS server other than the one running a *PDoT* RecRes.

Network Adversary. Recall that this adversary captures all packets to/from *PDoT*. It cannot obtain plaintext queries since they flow over the TLS tunnel. The only information it can obtain from packets includes cleartext header fields, such as source and destination IP addresses. This information, coupled with a timing attack, might let the adversary correlate a packet sent from the client with a packet sent to an NS. The resulting privacy leakage is discussed in Section 4.8.1

4.7.2 Deployability

Section 4.5 argues that *PDoT* clients do not need special hardware, and require only minor software modifications (Requirement R2). To aid deployability, *PDoT* also provides several configurable parameters, including: the number of QueryHandle threads (to adjust throughput), the amount of memory dedicated to each thread (to serve clients that send a lot of queries at a given time), and the timeout of QueryHandle threads (to adjust the time for a QueryHandle thread to acquire a resource). Another consideration is incremental deployment, where some clients may request DNS-over-TLS without supporting *PDoT*. *PDoT* can



(a) Latency of *PDoT* and Unbound (Cold Start) (b) Latency of *PDoT* and Unbound (Warm Start)

Figure 4.4: Latency comparison of *PDoT* and Unbound

handle this situation by having its TLS certificate **also** be signed by a trusted root CA, since legacy clients will ignore *PDoT*-specific attestation information.

On the client side, an ideal deployment scenario would be for browser or OS vendors to update their client stubs to support *PDoT*. In the same way that browser vendors currently include and maintain a list of trusted root CA certificates in their browsers, they could include and periodically update a list of trustworthy MRENCLAVE values for PDoT resolvers. This could all be done transparently to end-users. As with root CA certificates, expert users can manually add/remove trusted MRENCLAVE values for their systems. In practice, there are only a handful of recursive resolver software implementations. Thus, even allowing for multiple versions of each, the list of trusted MRENCLAVE values would be orders of magnitude smaller than the list of public keys of every trusted resolver, as would be required for standard DNS-over-TLS.

4.7.3 Latency Evaluation

We aim to assess overhead introduced by running RecRes inside an enclave. To do so, we measure the time to resolve a DNS query using *PDoT* and compare this with the correspond-

ing latency incurred by Unbound [144], a popular open source RecRes. To meet requirement R3, *PDoT* should not incur a significant increase in latency compared to Unbound.

Experimental Setup. We ran *PDoT* on a DC4s_v2 series virtual machine (VM) in Microsoft Azure. This VM has 4 virtual CPUs and 16 GB of memory and supports Intel SGX with up to 112 MB of EPC. We used Ubuntu 18.04 for the OS and Intel SGX SDK version 2.9. We configured our *PDoT* to support up to 100 concurrent clients and used Stubby as the DNS-over-TLS client.

We measured latency under two scenarios: cold start and warm start. In the former, the client sets up a new TLS connection every time it sends a query to the RecRes. In the warm start scenario, the client sets up one TLS connection with the RecRes at the beginning and reuses it throughout the experiment. In other words, cold start measurements also include the time to establish the TLS connection. In this experiment, caching mechanisms of both *PDoT* and Unbound were disabled.

We created a Python program to feed DNS queries to the client. It sends 100 queries sequentially (i.e., waits for an answer to the previous query before sending the next query) for each of the top ten domains in the Majestic Million domain list [155].

The Python program measures the time between sending the query and receiving an answer. For the cold start experiment, we spawned a new Stubby client and established a new TLS connection for each query. In the warm start scenario, we first established the TLS connection by sending a query for another domain (not in the top ten), but did not include this in the timing measurement.

Note that the numeric latency values are specific to our experimental setup because they depend on network bandwidth of the RecRes, and latency between the latter and relevant NS-s. The important aspect of this experiment is the ratio between the latencies of *PDoT* and Unbound. Therefore, it is not meaningful to compute average latency over a large set of

domains. Instead, we took multiple measurements for each of a small set of domains (e.g., 100 measurements for each of 10 domains) to analyze the range of response latencies for each domain.

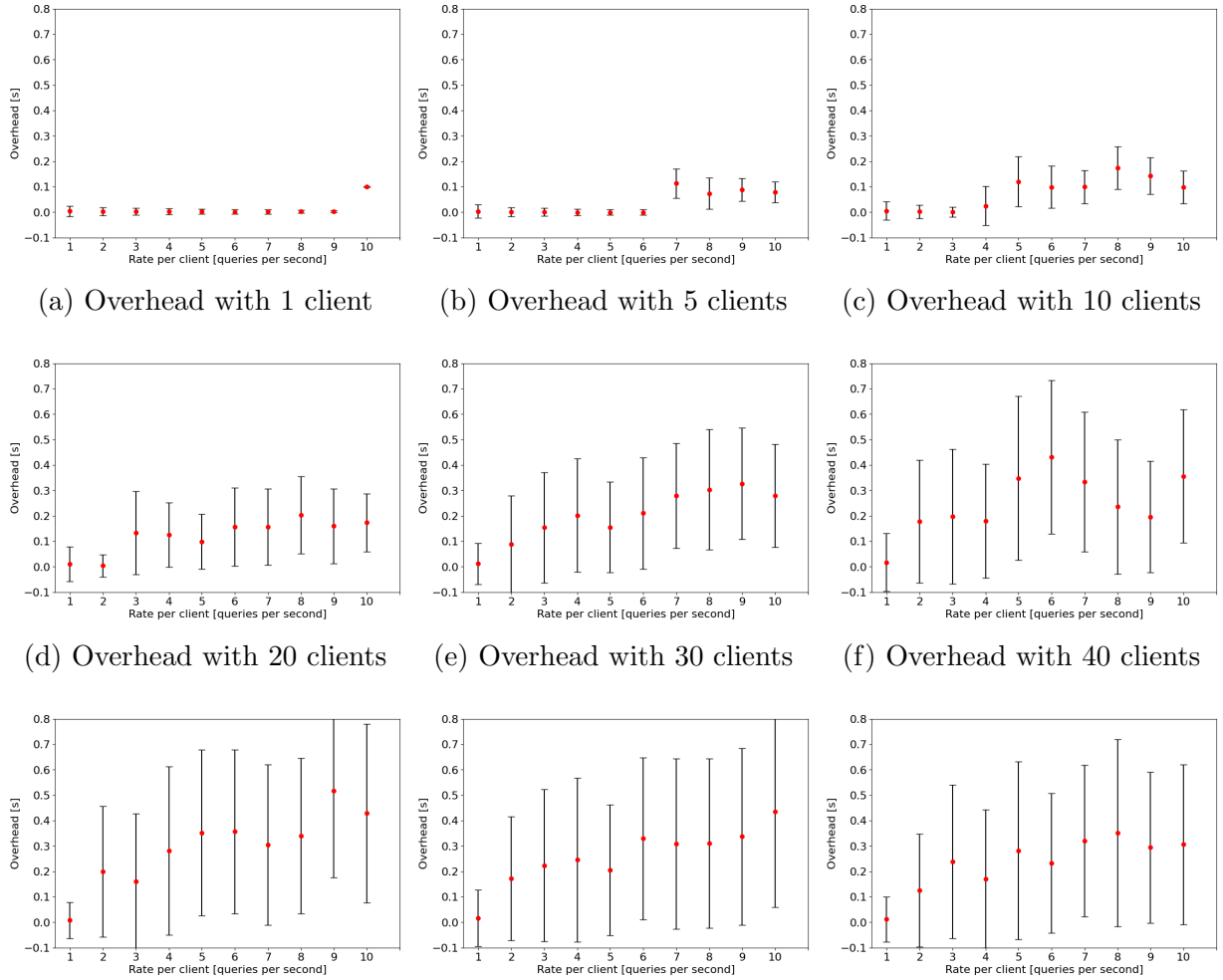
Results and observations. The results are shown in Figure 4.4. Red boxes show latency of *PDoT* and the blue boxes – of Unbound. In these plots, boxes span from the lower to upper quartile values of collected data. Whiskers span from the lowest datum within the 1.5 interquartile range (IQR) of the lower quartile to the highest datum within the 1.5 IQR of the upper quartile. Median values are shown as black horizontal lines inside the boxes.

For the cold-start case in Figure 4.4a, although Unbound is typically faster than our proof-of-concept *PDoT* implementation, the range of latencies is similar. For 5 out of 10 domains, the upper whisker of *PDoT* was lower than that of Unbound. Moreover, we observed that the range of latencies of *PDoT* overlaps with that of Unbound. Across the tested domains, *PDoT* shows an average of 73% overhead compared to Unbound in this setting.

For the warm-start case in Figure 4.4b, the median latency is lower across the board compared to the cold-start setting because the TLS tunnel has already been established. In this setting, *PDoT* shows an average of 44% overhead compared to Unbound. Compared to the cold start setting, the difference in the range of latencies is smaller. In fact, for half of the domains, the range of *PDoT* latencies is smaller than that of Unbound. In practice, once the client has established a connection to RecRes, it will maintain this connection; thus, the vast majority of queries will see only the warm-start latency.

4.7.4 Throughput evaluation

The objective of our throughput evaluation is to measure the rate at which the RecRes can sustainably respond to queries. *PDoT*'s throughput should be close to that of Unbound to satisfy requirement R4.



(a) Overhead with 1 client (b) Overhead with 5 clients (c) Overhead with 10 clients

(d) Overhead with 20 clients (e) Overhead with 30 clients (f) Overhead with 40 clients

(g) Overhead with 50 clients (h) Overhead with 60 clients (i) Overhead with 70 clients

Figure 4.5: Throughput overhead comparison of *PDoT*

Experiment setup. We used the same experimental setup as the latency evaluation (Section 4.7.3). The client and RecRes ran on different VMs located in the same virtual network so that the RecRes could use all available resources of a single VM. This is representative of a local RecRes running in a small network (e.g., a coffee shop WiFi network). We conducted this experiment using Stubby and the same two RecRes-s as in the latency experiment. Stubby was configured to reuse TLS connections where possible and the caching mechanisms of both *PDoT* and Unbound were disabled. To eliminate further variance due to external network latency, we also ran our own authoritative NS in the same virtual network as the RecRes. Running our own authoritative NS also has the benefit of not applying strain on

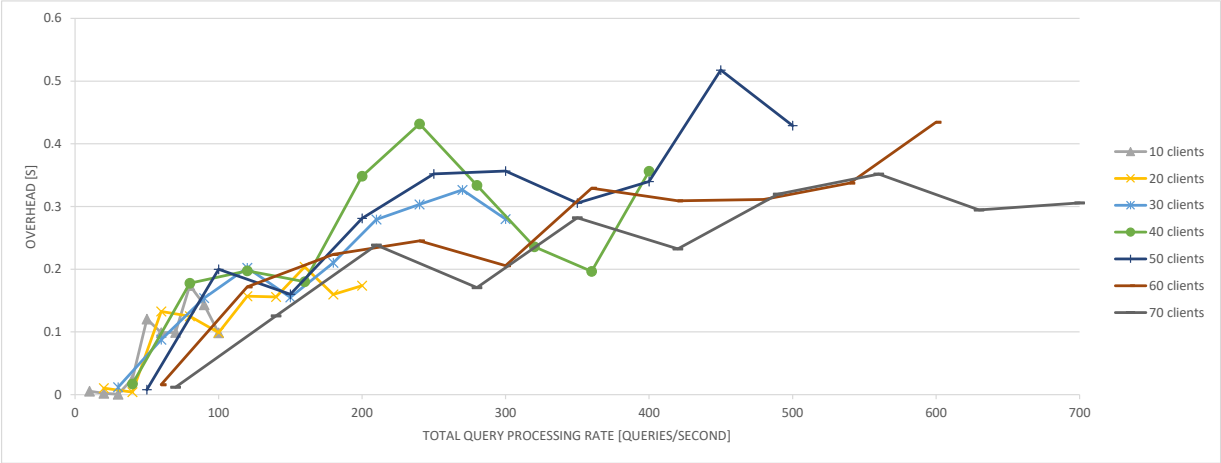


Figure 4.6: Average overhead of *PDoT* for different numbers of clients.

the public NS during our evaluation. All queries sent to the RecRes could be resolved with a single upstream query to our local NS. To simulate a small to medium-scale network, we varied the number of concurrent client connections between 1 and 70 and adjusted the query rate from 1 to 10 queries per second per client. We maintained each constant query rate for half a minute.

Results and observations. The results of our throughput experiments are shown in Figure 4.5. Each graph corresponds to a different number of clients. The horizontal axis shows different per-client query rates and the vertical axis shows the overhead for response latencies compared to the average response latency of Unbound for the same setting. During our preliminary evaluation, we found that the average latency of Unbound is consistently about 0.05 seconds across the evaluated scenarios. The red dots show the average overhead of *PDoT*, and the black error bars represent the standard deviation.

Figure 4.6 is a combination of the individual plots in Figure 4.5, showing the total rate at which *PDoT* is processing queries for different numbers of concurrent client connections on the x axis and the average latency overhead in the y axis. From this figure, we can see that the average latency overhead increases linearly as the total number of queries sent by the clients increases up to 200 queries per second. However, we can observe that the increase

in the overhead is smaller after 200 queries per second. In fact, we can almost see that the overhead is converging to a particular value. For instance, after 500 queries per second, the latency overhead that 70 clients experience is around 0.3 seconds on average.

4.7.5 Caching evaluation

Caching domains

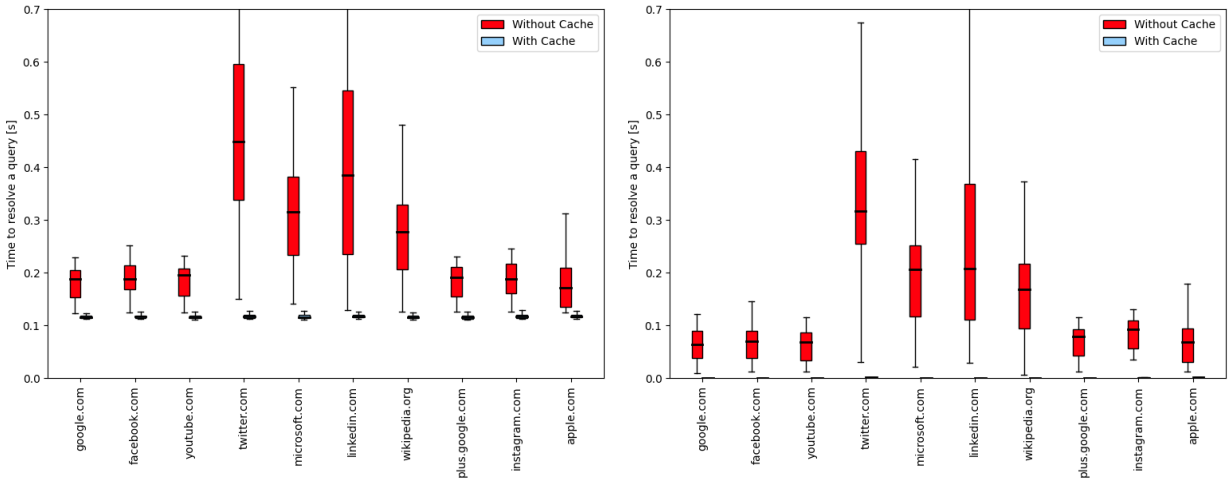
First, we counted how many domains can be handled by the in-enclave cache, as described in Section 4.5. A single client was set up to query domains from the Majestic Million domain list [155]. The client queried domains until *PDoT* experienced a significant increase in latency, indicating that the cache starts to exceed the size of the Enclave Page Cache (EPC). Although the cache could grow beyond this point, this would lead to increased latency because memory pages would have to be swapped in and out of the EPC. The memory setting for each thread was the same as the throughput evaluation. Even though different queries have answers of different sizes, this experiment showed that *PDoT* can cache at least 10,000 domains.

Effect on latency

Next, we quantified the effect of caching on query latency by repeating the latency evaluation with and without caching. The results are shown in Figure 4.7. The average latency of *PDoT* without caching in the cold start setting is 249 ms and with caching is 117 ms. In the warm start setting, the average latency without caching is 123 ms, and with caching is 1 ms. In addition to reducing average latency, caching also reduces the variance of the latency, thus preventing time-outs and improving the stability of the system.

Benefiting from caching

Section 4.7.5 showed how both users and *PDoT* can potentially benefit from caching query answers and Section 4.7.5 showed that our simple proof-of-concept in-enclave cache can



(a) Effect of Cache (Cold Start)

(b) Effect of Cache (Warm Start)

Figure 4.7: Latency comparison of *PDoT* without caching (red) and with caching (blue)

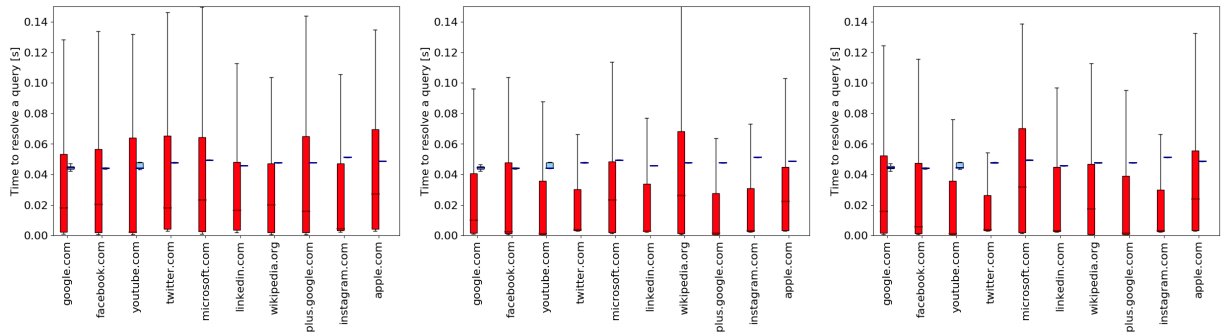
accommodate at least 10,000 domains. The overall performance impact of this type of cache depends heavily on the pattern of DNS queries sent by each user. However, we expect that a significant number of queries would fall within the most popular 10,000 domains, and could thus be answered from the cache. Additionally, *PDoT* operators can also pre-populate the cache with domains that are frequently queried by their users.

Effect of different numbers of domains in cache

Lastly, we evaluated the performance of both resolvers with caching enabled; Unbound with its default caching behavior, and *PDoT* with the proof-of-concept caching mechanism.

Experiment setup. To simulate a realistic small-scale network, we ran *PDoT* on a low-cost Intel NUC consisting of an Intel Pentium Silver J5005 CPU with 128 MB of EPC memory and 4 GB of RAM, using Ubuntu 16.04 and the Intel SGX SDK version 2.2. We pre-populated resolvers' caches with varying numbers of domains and measured response latency for a representative set of 10 popular domains.

Results and observations. Figure 4.8 shows the response latency with caching enabled. The box and whisker plots have the same meaning as in Figure 4.4. Unbound serves responses



(a) 10 domains in cache (b) 100 domains in cache (c) 1000 domains in cache

Figure 4.8: Latency comparison of *PDoT* (red) and Unbound (blue) with different number of domains in cache

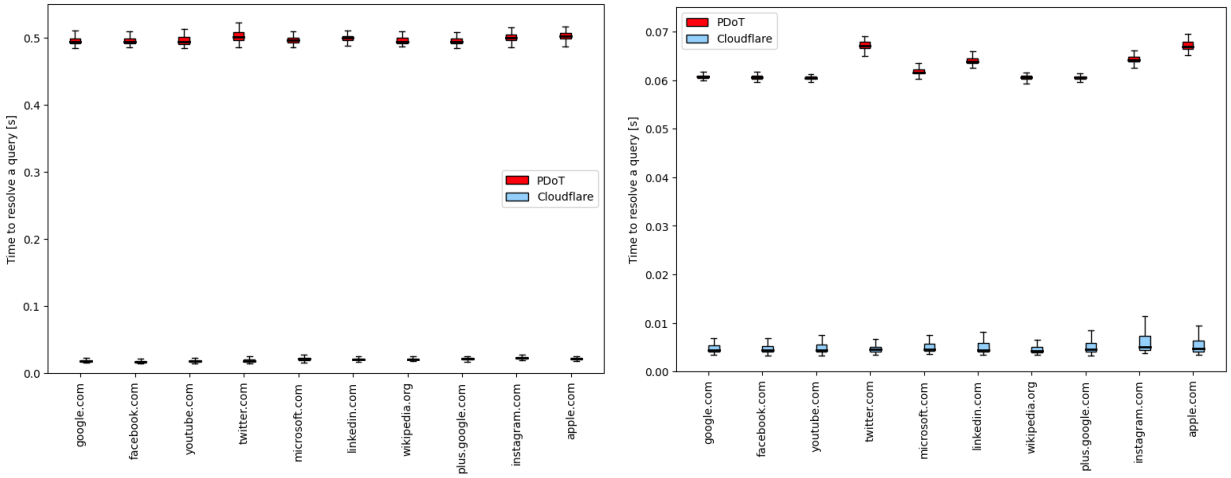
from cache with a consistent latency irrespective of the number of entries in the cache. Although *PDoT* achieves lower average latencies when the cache is relatively empty, it has higher variability than Unbound. This is possibly due to the combination of our unoptimized caching implementation and latency of accessing enclave memory. Nevertheless, Figure 4.8 shows that even with the memory limitations of current hardware enclaves, *PDoT* can still benefit from caching a small number of domains.

4.7.6 Real-world evaluation

In this experiment, we evaluated the difference in latency between Cloudflare's `1.1.1.1` and *PDoT*.

Experiment setup. *PDoT* was set up on a Microsoft Azure VM with caching enabled. The VM was set up on the US East coast region. Meanwhile, we confirmed that `1.1.1.1` resides on the US West coast region. A Stubby client on a machine residing on the US West coast region. We ran the same latency evaluation script we used in Section 4.7.3. Since it is highly likely that `1.1.1.1` has the ten domains cached in its system, we also pre-populated the domains in *PDoT*'s cache before conducting the experiment.

Results and observations. Figure 4.9 shows the result of the latency measurement. The



(a) Latency comparison in cold start setting

(b) Latency comparison in warm start setting

Figure 4.9: Latency comparison of *PDoT* (red) and `1.1.1.1` (blue)

box and whisker plots have the same meaning as in Figure 4.4. We can see that in both settings there is a difference in latency between the two resolvers. In the cold start setting, the average latency of *PDoT* was 0.5 seconds while `1.1.1.1` was 0.02 seconds. In the warm start setting, the average latency of *PDoT* was 0.06 seconds while `1.1.1.1` was 0.005 seconds. The difference in latency in the warm start setting can be due to several reasons. This includes the network latency due to the location of the resolver, `1.1.1.1` using optimized caching mechanisms, more powerful machines, and load balancing methods. As the cold start setting includes the TLS handshake, we assume the difference in latency in the cold start setting is largely due to the fact that `1.1.1.1` uses optimized cryptographic operations and optimized TLS handshake methods. Overall, we can conclude that the query response latency of our proof-of-concept *PDoT* is well below standard timeout time, which is 5 seconds for DNS queries. A production-level *PDoT* can adopt the methods which `1.1.1.1` uses to further decrease its latency in resolving queries.

4.8 Discussion

4.8.1 Information Revealed by IP Addresses

Even if the connections between the client, RecRes, and NS-s are encrypted using TLS, some information is still leaked. The most prominent and obvious is source/destination IP addresses. The network adversary described in Section 4.3.1 can combine these cleartext IP addresses with packet timing information in order to correlate packets sent from client to RecRes with subsequent packets sent from RecRes to NS.

Armed with this information, the adversary can narrow down the client's domain name query to one of the records that could be served by that specific ANS. Assuming the ANS can serve R domain names, the adversary has a $1/R$ probability of guessing which domain name the user queried. When $R > 1$, we call this a *privacy-preserving ANS*. This prompts two questions: 1) what percentage of domains can be answered by a privacy-preserving ANS; and 2) what is the typical size of anonymity set (R) provided by a privacy-preserving ANS?

To answer these questions, we designed a scheme to collect records stored in various ANS-s. We sent DNS queries for 1,000,000 domains from the Majestic Million domain list [155], and gathered information about ANS-s that can possibly provide the answer for each. By collecting data on possible ANS-s, we can map domain names to each ANS, and thus estimate the number of records held by each ANS. Following the *Guidelines for Internet Measurement Activities* [52], we limited our querying rate, in order to avoid placing undue load on any servers.

As shown in Figure 4.10, only 5.7% of domains we queried were served by non-privacy-preserving ANS-s, i.e., those that hold only one record). Examples of domain names served from non-privacy-preserving ANS-s included: `tinyurl.com`[3], `bing.com`, `nginx.org`, `news.bbc.co.uk`,

[3]Since `tinyurl.com` is a URL shortening service, this is actually still privacy-preserving because the adversary can not learn which short URL was queried.

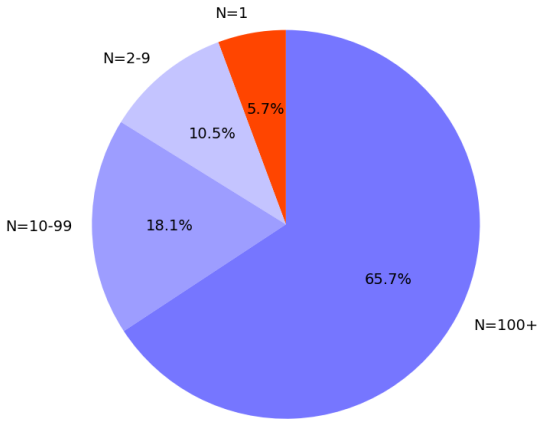


Figure 4.10: Percentage of Majestic Million domains answered by an ANS with at least N records

and `cloudflare.com`. On the other hand, 9 out of 10 queries were served by a privacy-preserving ANS, and 65.7% by ANS-s that hold over 100 records.

These results are still approximations. Since we do not have data for domains outside the Majestic Million list, we cannot make claims about whether these would be served by a privacy-preserving ANS. We hypothesize that the vast majority of ANS-s would be privacy-preserving for the simple reason that it is more economical to amortize the ANS's running costs over multiple domains. On the other hand, we can be certain that our results for the Majestic Million are a strict lower bound on the level of privacy because the ANS-s from which these are served could also be serving other domains outside of our list. It would be possible to arrive at a more accurate estimate by analyzing zone files of all (or at least most) ANS-s. However, virtually all ANS-s disable the interface to download zone files because this could be used to mount DoS attacks. Therefore, this type of analysis would have to be performed by an organization with privileged access to all ANS-s' zone files.

4.8.2 Supporting DNS-over-HTTPS (DoH)

Concurrently with the development of DNS-over-TLS, there has also been significant development of protocols and standards for DNS resolution over HTTPS (DoH) [119].

Similarly to DoT, the aim of DoH is to protect the confidentiality and integrity of DNS queries and responses in transit between the client and RecRes. DoH still uses TLS as the underlying secure channel, although layers the DNS queries and responses on top of HTTPS. Whereas DoT uses its dedicated port (853), DoH uses the same port as other HTTPS traffic (443). The last main difference is that DoH adds the ability for a server to *push* DNS responses to a client without the client having to first send a request.

DoH is already supported by a similar set of public DNS resolvers as DoT (e.g., Cloudflare's `1.1.1.1`) and is gaining support in web browsers (e.g., Firefox [42]) and operating systems (e.g., Windows 10 [133]). In February 2020, it was estimated that on average there are still seven times more DoT requests than DoH requests [17].

DoH faces similar challenges to DoT, and thus would also benefit from the security guarantees provided by *PDoT*. *PDoT* could be modified to support DoH by adding an HTTP parsing layer. Although this slightly increases the amount of software running within the TEE (i.e., the software trusted computing base), only minimal HTTP parsing functionality would be required.

4.9 Related Work

There has been much prior work aiming to protect the privacy of DNS queries [51, 240, 239, 154, 84, 208, 81]. For example, Lu et al. [154] proposed a privacy-preserving DNS that uses distributed hash tables, different naming schemes, and methods from computational private information retrieval. Federrath et al. [84] introduced a dedicated DNS Anonymity Service to protect the DNS queries using an architecture that distributes the top domains by broadcast and uses low-latency mixes for requesting the remaining domains. These schemes all assume that all parties involved do not act maliciously.

There have also been some activities in the Internet standards community that focused on

DNS security and privacy. DNS Security Extensions (DNSSEC) [20] provide data origin authentication and integrity via public key cryptography. However, it does not offer privacy. Bortzmeyer [35] proposed a scheme Also, though not Internet standards, several protocols have been proposed to encrypt and authenticate DNS packets between the client and the RecRes (DNSCrypt [195]) and RecRes and NS-s (DNSCurve [70]). Moreover, the original DNS-over-TLS paper has been converted into a draft Internet standard [120]. All these methods assume that the RecRes operator is trusted and does not attempt to learn anything from the DNS queries.

Furthermore, there has been some research on establishing trust through TEEs to protect confidentiality and integrity of network functions. Specifically, SGX has been used to protect network functions, especially middle-boxes. For example, Endbox [92] aims to distribute middle-boxes to client edges: clients connect through VPN to ensure confidentiality of their traffic while remaining maintainable. LightBox [79] is another middle-box that runs in an enclave; its goal is to protect the client's traffic from the third-party middle-box service provider while maintaining adequate performance. Finally, ShieldBox [217] aims to protect confidential network traffic that flows through untrusted commodity servers and provides a generic interface for easy deployability. These efforts focus on protecting confidential data that flows in the network and do not target DNS queries.

4.10 Conclusion & Future Work

This chapter proposed *PDoT*, a novel DNS RecRes design that operates within a TEE to protect privacy of DNS queries, even from a malicious RecRes operator. In terms of query throughput, our unoptimized proof-of-concept implementation matches the throughput of Unbound, a state-of-the-art DNS-over-TLS recursive resolver, while incurring an acceptable increase in latency (due to the use of a TEE). In order to quantify the potential for privacy leakage through traffic analysis, we performed an Internet measurement study which showed

that 94.7% of the top 1,000,000 domain names can be served from a *privacy-preserving* ANS that serves at least two distinct domain names, and 65.7% from an ANS that serves 100+ domain names. As future work, we plan to port the Unbound RecRes to Intel SGX and conduct a performance comparison with *PDoT*, as well as explore methods for improving *PDoT*'s performance using caching while maintaining client privacy. We also plan to investigate supporting DNS-over-HTTPS.

Chapter 5

CACTI: Captcha Avoidance via Client-side TEE Integration

Abstract

Preventing abuse of web services by bots is an increasingly important problem, as abusive activities grow in both volume and variety. CAPTCHAs are the most common way for thwarting bot activities. However, they are often ineffective against bots and frustrating for humans. In addition, some recent CAPTCHA techniques diminish user privacy. Meanwhile, client-side Trusted Execution Environments (TEEs) are becoming increasingly widespread (notably, ARM TrustZone and Intel SGX), allowing establishment of trust in a small part (trust anchor or TCB) of client-side hardware. This prompts the question: can a TEE help reduce (or remove entirely) user burden of solving CAPTCHAs?

In this chapter, we design CACTI: **C**APTCHA **A**voidance via **C**lient-side **T**EE **I**ntegration. Using client-side TEEs, CACTI allows legitimate clients to generate unforgeable *rate-proofs* demonstrating how frequently they have performed specific actions. These rate-proofs can be sent to web servers in lieu of solving CAPTCHAs. CACTI provides strong client privacy guarantees since the information is only sent to the visited website and authenticated using a group signature scheme. Our evaluations show that overall latency of generating and verifying a CACTI rate-proof is less than 0.25 sec, while CACTI's bandwidth overhead is over 98% lower than that of current CAPTCHA systems.

Research presented in this chapter appeared in the Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021) [175].

5.1 Introduction

In the past two decades, as Web use became almost universal and abuse of Web services grew dramatically, there has been an increasing trend (and real need) to use security tools that help prevent abuse by automated means, i.e., so-called **bots**. The most popular mechanism is CAPTCHAs: Completely Automated Public Turing test to tell Computers and Humans Apart [224]. A CAPTCHA is essentially a puzzle, such as an object classification task (Figure 5.1a) or distorted text recognition (see Figure 5.1b), that aims to confound (or at least slow down) a bot, while being easily[1] solvable by a human user. CAPTCHAs are often used to protect sensitive actions, such as creating a new account or submitting a web form.

Although primarily intended to distinguish humans from bots, it has been shown that CAPTCHAs are not very effective at this task [173]. Many CAPTCHAs can be solved by algorithms (e.g., image recognition software) or outsourced to human-driven *CAPTCHA-farms*[2] to be solved on behalf of bots. Nevertheless, CAPTCHAs are still widely used to increase the adversary's costs (in terms of time and/or money) and reduce the *rate* at which bots can perform sensitive actions. For example, computer vision algorithms are computationally expensive, and outsourcing to CAPTCHA-farms costs money and takes time.

From the users' perspective, CAPTCHAs are generally unloved (if not outright hated), since they represent a barrier and an annoyance (a.k.a. Denial-of-Service) for legitimate users. Another major issue is that most CAPTCHAs are visual in nature, requiring sufficient ambient light and screen resolution, as well as good eyesight. Much less popular audio CAPTCHAs are notoriously poor and require a quiet setting, decent-quality audio output facilities, as well as good hearing.

More recently, the reCAPTCHA approach has become popular. It aims to reduce user burden by having users click a checkbox (Figure 5.1c) while performing behavioral analysis

[1] Exactly what it means to be "easily" solvable is subject to some debate.

[2] A CAPTCHA farm is usually sweatshop-like operation, where employees solve CAPTCHAs for a living.

of the user's browser interactions. Acknowledging that even this creates friction for users, the latest version ("invisible reCAPTCHA") does not require any user interaction. However, the reCAPTCHA approach is potentially detrimental to **user privacy** because it requires maintaining long-term state, e.g., in the form of Google-owned cookies. Cloudflare recently decided to move away from reCAPTCHA due to privacy concerns and changes in Google's business model [62].

Notably, all current CAPTCHA-like techniques are server-side, i.e., they do not rely on any security features of, or make any trust assumptions about, the client platform. The purely server-side nature of CAPTCHAs was reasonable when client-side hardware security features were not widely available. However, this is rapidly changing with the increasing popularity of Trusted Execution Environments (TEEs) on a variety of computing platforms, e.g., TPM and Intel SGX for desktops/laptops and ARM TrustZone for smartphones and even smaller devices. Thus, it is now realistic to consider abuse prevention methods that include client-side components. For example, if a TEE has a trusted path to some form of user interface, such as a mouse, keyboard, or touchscreen, this *trusted User Interface (UI)* could securely confirm user presence. Although this feature is still unavailable on most platforms, it is emerging through features such as Android's Protected Confirmation [71]. This approach's main advantages are minimized user burden (e.g., just a mouse click) and increased security since it would be impossible for software to forge this action. Admittedly, however, this approach can be defeated by adversarial hardware e.g., a programmable USB peripheral that pretends to be a mouse or keyboard.

However, since the majority of consumer devices do not currently have a trusted UI, it would be highly desirable to reduce the need for CAPTCHAs using only existing TEE functionality. As discussed above, the main goal of modern CAPTCHAs is to increase adversarial costs and reduce the *rate* at which they can perform sensitive actions. Therefore, if legitimate users had a way to prove that their rate of performing sensitive actions is below some threshold,

a website could decide to allow these users to proceed without solving a CAPTCHA. If a user can not provide such proof, the website could simply fall back to using CAPTCHAs. Though this would not fully prevent bots, it would not give them any advantage compared to the current arrangement of using CAPTCHAs.

Motivated by the above discussion, this chapter presents CACTI, a flexible mechanism for allowing legitimate users to prove to websites that they are not acting in an abusive manner. By leveraging widespread and increasing availability of client-side TEEs, CACTI allows users to produce *rate-proofs*, which can be presented to websites in lieu of solving CAPTCHAs. A rate-proof is a simple assertion that:

1. The rate at which a user has performed some action is below a certain threshold, and

2. The user's time-based counter for this action has been incremented.

When serving a webpage, the server selects a *threshold* value and sends it to the client. If the client can produce a rate-proof for the given threshold, the server allows the action to proceed without showing a CAPTCHA. Otherwise, the server presents a CAPTCHA, as before. In essence, CACTI can be seen as a type of "express checkout" for legitimate users.

One of the guiding principles and goals of CACTI is user privacy – it reveals only the minimum amount of information and sends this directly to the visited website. Another principle is that the mechanism should not mandate any specific security policy for websites. Websites can define their own security policies e.g., by specifying thresholds for rate-proofs. Finally, CACTI should be configurable to operate without any user interaction, in order to make it accessible to all users, including those with sight or hearing disabilities.

Although chiefly motivated by the shortcomings of CAPTCHAs, we believe that the general approach of client-side (TEE-based) rate-proofs, can also be used in other common web scenarios. For example, news websites could allow users to read a limited number of articles

for free per month, without relying on client-side cookies (which can be cleared) or forcing users to log-in (which is detrimental to privacy). Online petition websites could check that users have not signed multiple times, without requiring users to provide their email addresses, which is once again, detrimental to privacy. We, therefore, believe that our TEE-based rate-proof concept is a versatile and useful web security primitive.

Anticipated contributions of this work are:

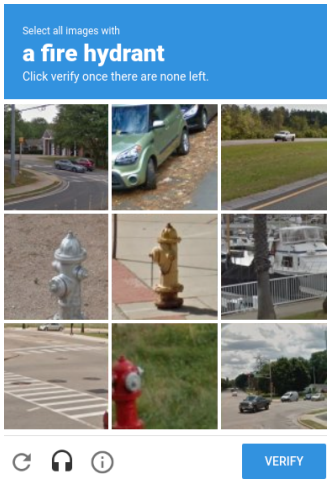
1. We introduce the concept of a *rate-proof*, a versatile web security primitive that allows legitimate users to securely prove that their rate of performing sensitive actions falls below a server-defined threshold.

2. We use the rate-proof as the basis for a concrete client-server protocol that allows legitimate users to present rate-proofs in lieu of solving CAPTCHAs.

3. We provide a proof-of-concept implementation of CACTI, over Intel SGX, realized as a Google Chrome browser extension.

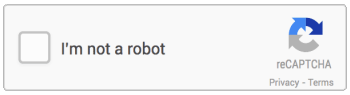
4. We present a comprehensive evaluation of security, latency, and deployability of CACTI.

Organization: Section 5.2 provides background information, and Section 5.3 defines our threat model and security requirements. Next, Section 5.4 presents our overall design and highlights the main challenges in realizing this. Then, Section 5.5 describes our proof of concept implementation and discusses how CACTI overcomes the design challenges, followed by Section 5.6 which presents our evaluation of the security, performance, and deployability of CACTI. Section 5.7 discusses further optimizations and deployment considerations, and Section 5.8 summarizes related work.



(a) Image-based object recognition re-CAPTCHA [103]

(b) Image-based text recognition re-CAPTCHA [103]



(c) Behavior-based reCAPTCHA [103]

Figure 5.1: Examples of CAPTCHAs

5.2 Background

5.2.1 Group Signatures

A group signature scheme aims to prevent the verifier from determining the group member who generated the signature. Each group member is assigned a group private key under a single group public key. In case a group member needs to be revoked, a special entity called *group manager* can open the signature. A group signature scheme is composed of five algorithms [25]:

- **Setup:** Given a security parameter, an efficient algorithm outputs a group public key and a master secret for the group manager.
- **Join:** A user interacts with the group manager to receive a group private key and a membership certificate.
- **Sign:** Using the group public key, group private key, membership certificate, and

message m, a group member generates a group signature of m.

- **Verify:** Using the group public key, an entity verifies a group signature.
- **Open:** Given a message, a putative signature on the message, the group public key, and the master secret, the group manager determines the identity of the signer.

A secure group signature scheme satisfies the following properties [25]:

- **Correctness:** Signatures generated with any member's group private key must be verifiable by the group public key.
- **Unforgeability:** Only an entity that holds a group private key can generate signatures.
- **Anonymity:** Given a group signature, it must be computationally hard for anyone (except the group manager) to identify the signer.
- **Unlinkability:** Given two signatures, it must be computationally hard to determine whether these were signed by the same group member.
- **Exculpability:** Neither a group member nor the group manager can generate signatures on behalf of other group members.
- **Traceability:** The group manager can determine the identity of a group member that generated a particular signature.
- **Coalition-resistance:** Group members cannot collude to create a signature that cannot be linked to one of the group members by the group manager.

Enhanced Privacy ID (EPID) [41] is a group signature scheme used by remote attestation of Intel SGX enclaves. It satisfies the above properties while providing additional privacy-preserving revocation mechanisms to revoke compromised or misbehaving group members.

Specifically, EPID's *signature-based revocation* protocol does not "Open" signatures. Rather, it uses a signature produced by the revoked member to notify other entities that this particular member has been revoked.

5.3 System & Threat Models

The ecosystem that we consider includes three types of principals/players: (1) servers, (2) clients, and (3) TEEs. There are multitudes of these three principal types. The number of clients is the same as that of TEEs, and each client houses exactly one TEE. Even though a TEE is assumed to be physically within a client, we consider it to be a separate security entity. Note that a human user can, of course, operate or own multiple clients, although there is clearly a limit and more clients imply higher costs for the user.

We assume that all TEEs are trusted: honest, benign, and insubvertible. We consider all side-channel and physical attacks against TEEs to be out of scope of this work and assume that all algorithms and cryptographic primitives implemented within TEEs are impervious to such attacks. We also consider cuckoo attacks, whereby a malicious client utilizes multiple (possibly malware-infected) machines with genuine TEEs, to be out of scope, since clients and their TEEs are not considered to be strongly bound. We refer to [237] and [78] as far as means for countering such attacks. We assume that servers have the means to authenticate and attest TEEs, possibly with the help of the TEE manufacturer.

All clients and servers are untrusted, i.e., they may act maliciously. The goal of a malicious client is to avoid CAPTCHAs, while a malicious server either aims to inconvenience a client (via DoS) or violate client's privacy. For example, a malicious server can try to learn the client's identity or link multiple visits by the same client. Also, multiple servers may collude in an attempt to track clients.

Our threat model yields the following requirements for the anticipated system:

- **Unforgeability:** Clients cannot forge or modify CACTI rate-proofs.
- **Client privacy:** A server (or a group thereof) cannot link rate-proofs to the clients that generated them.

We also pose the following non-security goals:

- **Latency:** User-perceived latency should be minimized.
- **Data transfer:** The amount of data transfer between client and server should be minimized.
- **Deployability:** The system should be deployable on current off-the-shelf client and server hardware.

5.4 CACTI Design & Challenges

This section discusses the overall design of CACTI and justifies our design choices.

5.4.1 Conceptual Design

Rate-proofs. The central concept underpinning our design is the *rate-proof* (RP). Conceptually, the idea is as follows: Assuming that a client has an idealized TEE, the TEE stores one or more named sorted lists of timestamps in its rollback-protected secure memory. To create a rate-proof for a specific list, the TEE is given the name of the list, a *threshold* (Th), and a new timestamp (t). The threshold is expressed as a starting time (t_s) and a count (k). This can be interpreted as: *"no more than k timestamps since t_s"*. The TEE checks that the specified list contains k or fewer timestamps with values greater than or equal to t_s. If so, it checks if the new timestamp t is greater than the latest timestamp in the list. If both checks succeed, the TEE pre-pends t to the list and produces a signed statement confirming that the named list is below the specified threshold and the new timestamp has been added.

If either check fails, no changes are made to the list and no proof is produced. Note that the rate-proof does not disclose the number of timestamps in the list.

Furthermore, each list can also be associated with a public key. In this case, requests for rate-proofs must be accompanied by a signature over the request that can be verified with the associated public key. This allows the system to enforce a *same-origin* policy for specific lists – proofs over such lists can only be requested by the same entity that created them. Note that this does not provide any binding to the *identity* of the entity holding the private key, as doing so would necessitate the TEE to check identities against a global public key infrastructure (PKI) and we prefer for CACTI not to require it.

Rate-proofs differ from rate *limits* because the user is allowed to perform the action any number of times. However, once the rate exceeds the specified threshold, the user will no longer be able to produce rate-proofs. The client can always decide to *not* use its TEE; this covers clients who do not have TEEs or those whose rates exceeded the threshold. On the other hand, if the server does not yet support CACTI, the client does not store any timestamps or perform any additional computation.

CAPTCHA-avoidance. In today's CAPTCHA-protected services, the typical interaction between the client (C) and server (S) proceeds as follows:

1. C requests access to a service on S.
2. S returns a CAPTCHA for C to solve.
3. C submits the solution to S.
4. If the solution is verified, S allows C access to the service.

Although modern approaches, e.g., reCAPTCHA, might include additional steps (e.g., communicating with third-party services), these can be abstracted into the above pattern.

Our CAPTCHA-avoidance protocol keeps the same interaction sequence while substituting steps 2 and 3 with rate-proofs. Specifically, in step 2, the server sends a threshold rate and the current timestamp. In step 3, instead of solving a CAPTCHA, the client generates a rate-proof with the specified threshold and timestamp, and submits it to the server. The server has two types of lists:

- **Server-specific:** The server requests a rate-proof over its own list. The name of the list could be the server's URL, and the request may be signed by the server. This determines the rate at which the client visits this specific server.

- **Global:** The server requests a rate-proof over a global list, with a well-known name, e.g. `CACTI-GLOBAL`. This yields the rate at which the client visits all servers that use the global list.

The main idea of CAPTCHA avoidance is that a legitimate client should be able to prove that its rate is below the server-defined threshold. In other words, the server should have sufficient confidence that the client is not acting in an abusive manner (where the threshold of between abusive and non-abusive behaviors is set by the server). Servers can select their own thresholds according to their own security requirements. A given server can vary the threshold across different actions or even across different users or user groups, e.g., lower thresholds for suspected higher-risk users. If a client cannot produce a rate-proof, or is unwilling to do so, the server simply reverts to the current approach of showing a CAPTCHA. CACTI essentially provides a *fast-pass* for legitimate users.

The original CAPTCHA paper [224] suggested that CAPTCHAs could be used in the following scenarios:

1. **Online polls:** to prevent bots from voting,

2. **Free email services:** to prevent bots from registering for thousands of accounts,

3. **Search engine bots:** to preclude or inhibit indexing of websites by bots,

4. **Worms and spam:** to ensure that emails are sent by humans,

5. **Preventing dictionary attacks.** to limit the number of password attempts.

As discussed in Section 5.1, it is unrealistic to assume that CAPTCHAs cannot be solved by bots (e.g., using computer vision algorithms) or outsourced to CAPTCHA farms. Therefore, we argue that all current uses of CAPTCHAs are actually intended to slow down attackers or increase their costs. In the list above, scenarios 2 and 5 directly call for rate-limiting, while scenarios 1, 3, and 4 can be made less profitable for attackers if sufficiently rate-limited. Therefore, CACTI can be used in all these scenarios.

In addition to CAPTCHAs, modern websites use a variety of abuse-prevention systems (e.g., filtering based on client IP address or cookies). We envision CACTI being used alongside such mechanisms. Websites could dynamically adjust their CACTI rate-proof thresholds based on information from these other mechanisms. We are aware that rate-proofs are a versatile primitive that could be used to fight abusive activity in other ways, or even enable new use-cases. However, in this chapter, we focus on the important problem of reducing the user burden of CAPTCHAs.

5.4.2 Design Challenges

In order to realize the conceptual design outlined above, we identify the following key challenges:

TEE attestation. In current TEEs, the process of remote attestation is not standardized. For example, in SGX, a verifier must first register with Intel Attestation Service (IAS) before it can verify TEE quotes. Other types of TEEs would have different processes. It is unrealistic to expect every web server to establish relationships with such services from all

manufacturers in order to verify attestation results. Therefore, web servers cannot directly verify the attestation, while still needing to ascertain that the client is running a genuine TEE.

TEE memory limitations. TEEs typically have a small amount of secure memory. For example, if the memory of an SGX enclave exceeds the size of the EPC (usually 128 MB), the CPU has to swap pages out of the EPC. This is a very expensive operation since these pages must be encrypted and integrity protected. Therefore, CACTI should minimize the required amount of enclave memory, since other enclaves may be running on the same platform.

Limited number of monotonic counters. TEEs typically have a limited number of hardware monotonic counters, e.g., SGX allows at most 256 per enclave. Also, the number of counter increments can be limited, e.g., in SGX the limit is 100 in a single epoch [122] – a platform power cycle, or a 24-hour period. This is a challenge because hardware monotonic counters are critical for achieving rollback-protected storage. Recall that CACTI requires rollback-protected storage for all timestamps, to prevent malicious clients from rolling-back the timestamp lists and falsifying rate-proofs. Furthermore, this storage must be updated every time a new timestamp is added, i.e., for each successful rate-proof.

TEE entry/exit overhead. Invoking TEE functionality typically incurs some overhead. For example, whenever an execution thread enters/exits an SGX enclave, the CPU has to perform various checks and procedures (e.g., clearing registers) to ensure that enclave data does not leak. Identifying and minimizing the number of TEE entries/exits, while maintaining functionality, can be challenging.

5.4.3 Realizing CACTI Design

We now present a detailed design that addresses aforementioned design challenges. We describe its implementation in Section 5.5.

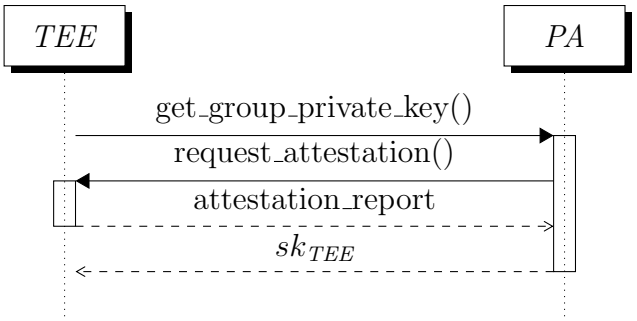


Figure 5.2: CACTI provisioning protocol. The interaction between the Provisioning Authority (PA) and the client's TEE takes place over a secure connection, using the client to pass the encrypted messages. After verifying the attestation report (and any other required information), the PA provisions the TEE with a group private key (sk_{TEE}).

Communication protocol

The web server must be able to determine that a supplied rate-proof was produced by a genuine TEE. Typically, this would be done using remote attestation, where the TEE proves that it is running CACTI code. If the TEE provides privacy-preserving attestation (e.g., the EPID protocol used in SGX remote attestation), this would also fulfill our requirement for client privacy, since websites would not be able to link rate-proofs to specific TEEs.

However, as described above, current TEE remote attestation is not designed to be verified by anonymous third parties. Furthermore, as CACTI is not limited to any particular TEE type, websites would need to understand attestation results from multiple TEE vendors, potentially using different protocols. Finally, some types of TEEs might not support privacy-preserving remote attestation, which would undermine our requirement for client privacy.

To overcome this challenge, we introduce a separate *Provisioning Authority* (PA) in order to unify various processes for attesting CACTI TEEs. Fundamentally, the PA is responsible for verifying TEE attestation (possibly via the TEE vendor) and establishing a privacy-preserving mechanism through which websites can also establish trust in the TEE. Specifically, the PA protects user privacy by using the EPID group signature scheme. The PA plays the role of the EPID *issuer*, and – optionally – the *revocation manager* [41]. During the pro-

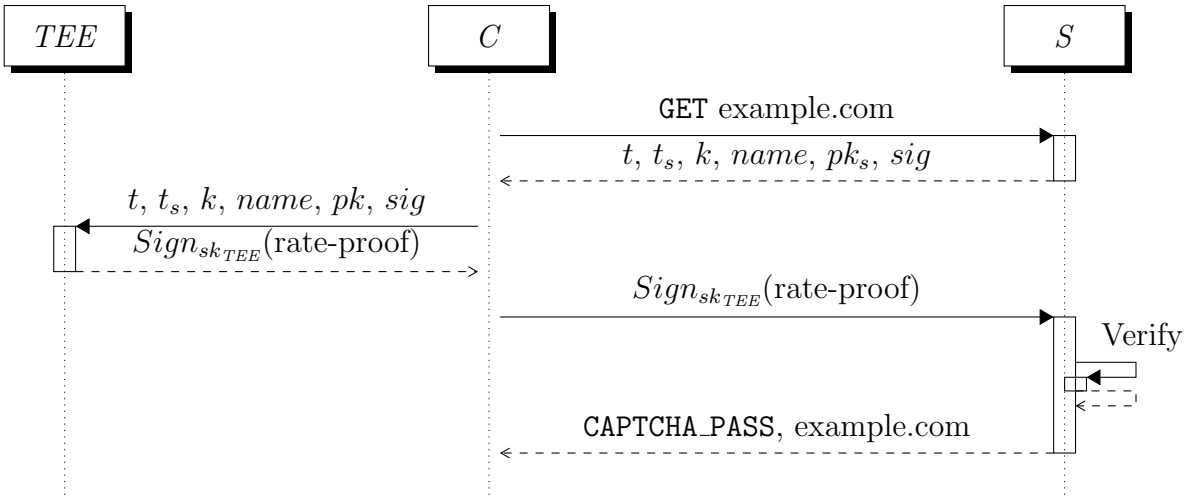


Figure 5.3: CACTI CAPTCHA-avoidance protocol. The client (C) requests a resource from the web server (S). In response, the server provides a timestamp for the current event (t), a threshold consisting of a starting time (t_s) and a count (k), and the *name* of the list. Optionally, the server also provides a signature (sig) over the request and the public key (pk_s) with which the signature can be verified. The client passes this information to its TEE in order to produce a rate-proof, signed by a group private key (sk_{TEE}), which can be verified by the server.

visioning phase (as shown in Figure 5.2), the *PA* verifies the attestation from the client's TEE and then runs the EPID `join` protocol with the client's TEE in order to provision the TEE with a group private key sk_{TEE}. The *PA* certifies and publishes the group public key pk_G. The *PA* may optionally require the client to prove their identity (e.g., by signing into an account) – this is a business decision and different *PA*s may take different approaches. After provisioning, the *PA* is unable to link signatures to any specific client thanks to the properties of the underling BBS+ signature scheme and signature-based revocation used in EPID [41]. We analyze security implications of malicious *PA*s in Section 5.6.1 and discuss the use of other group signature schemes in Section 5.7.2. There can be multiple *PA*s and websites can decide which *PA*s to trust. If a TEE is provisioned by an unsupported *PA*, the website would fall back to using CAPTCHAs.

Once the TEE has been provisioned, the client can begin to use CACTI when visiting supported websites, as shown in Figure 5.3. Specifically, when serving a page, the server includes

the following information: a timestamp t, a threshold Th (including start time t_s and count k), the name of the list (or `CACTI-GLOBAL` for the global list), and (optionally) a public key and signature for rates that enforce a same-origin policy. The client uses this information to request a rate-proof from their TEE. If the client's rate is indeed below the threshold, the TEE produces the rate-proof, signed with its group private key. The client then sends this to the server in lieu of solving a CAPTCHA.

TEE Design

To realize the conceptual design above, the client's TEE would ideally store all timestamps indefinitely in integrity-protected and rollback-protected memory. However, as discussed above, current TEEs fall short of this idealized representation, since they have limited integrity-protected memory and a limited number of hardware counters for rollback protection. To overcome this challenge, we store all data outside the TEE, e.g., in a standard database. To prevent dishonest clients from modifying this data, we use a combination of hash chains and Merkle Hash Trees (MHTs) to achieve integrity and rollback-protection.

Hash chains of timestamps. To protect integrity of stored timestamps, we compute a hash chain over each list of timestamps, as shown in Figure 5.4. Thus the TEE only needs to provide integrity and rollback-protected storage for the most recent hash in each hash chain. For efficiency, we store intermediate value of the hash chain along with each timestamp outside the TEE.

MHT of lists. Although it would be possible for the TEE to seal the most recent hash of each list individually, the lists may be updated independently, so the TEE would need separate hardware monotonic counters to provide rollback protection for each list. In a real-world deployment, the number of lists is likely to exceed the number of available hardware counters, e.g., 256 counters per enclave in SGX. To overcome this challenge, we combine the lists into a Merkle Hash Tree (MHT). As shown in Figure 5.5, each leaf of the MHT

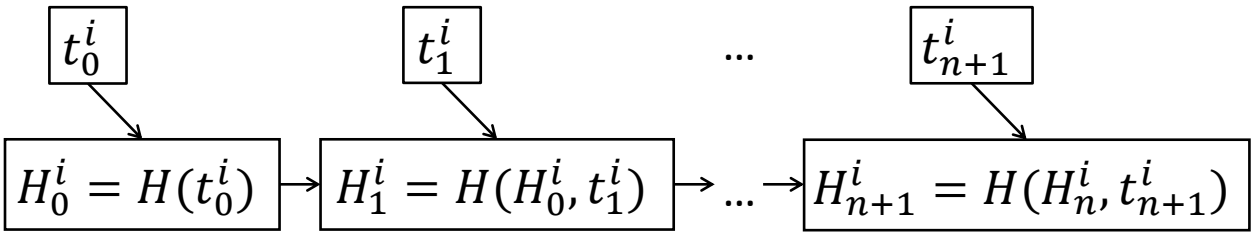


Figure 5.4: Hash chain of timestamps t_j^i for list i. $H()$ is a cryptographic hash function.

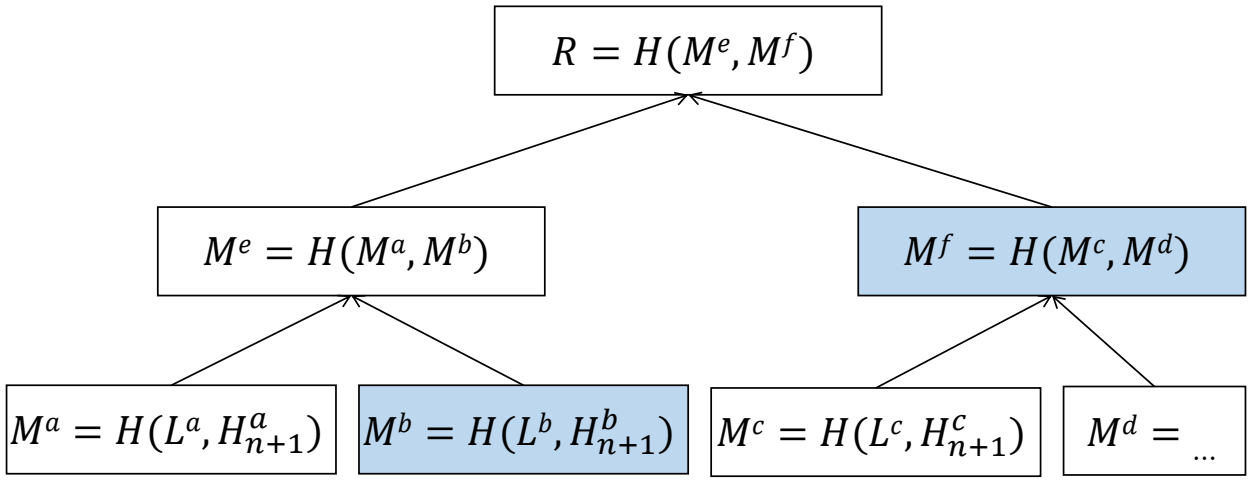


Figure 5.5: Merkle Hash Tree over lists $a...d$. Each leaf is a hash of the list information L^i (list name and public key) and the most recent hash of the list's hash chain H_{n+1}^i. $H()$ is a cryptographic hash function, R is the root of the MHT, and the nodes in blue illustrate the inclusion proof path for list b.

is a hash of the list information (list name and public key) and the most recent hash in the list's hash chain. With this arrangement, the TEE only needs to provide integrity and rollback-protected storage for the MHT root R, which can be achieved using sealing and a single hardware monotonic counter.

Producing a Rate-Proof

The TEE first needs to verify the integrity of its externally-stored data structures (i.e., hash chains and MHT described above), and if successful, update these with the new timestamp and produce the rate-proof, as follows:

1. TEE inputs. The client supplies its TEE with the list information and all timestamps

in the list that are greater than or equal to the server-defined start time t_s. The client also supplies the largest timestamp that is smaller than t_s, which we denote $t_{s-\delta}$, and the intermediate value of the hash chain up to, while discluding, $t_{s-\delta}$. The client supplies the sealed MHT root and intermediate hashes required to verify that the list is in the MHT.

2. Hash chain checks. The TEE first checks that $t_{s-\delta}$ is smaller than t_s and then recomputes the hash chain over included timestamps in order to reach the most recent value. During this process, it counts the number of included timestamps and checks that this is less than the value k specified in the threshold. The inclusion of one timestamp outside the requested range ($t_{s-\delta}$) ensures that the TEE has seen all timestamps within the range. This process requires $\mathcal{O}(n)$ hashes, where n is the number of timestamps in the requested range.

3. MHT checks. The TEE then unseals the MHT root and uses the hardware counter to verify that it is the latest version. The TEE then checks that the list information and that the calculated most recent hash value is indeed a leaf in the MHT. This process requires $\mathcal{O}(log(s))$ hashes, where s is the number of lists. Including the list name in the MHT leaf ensures that the timestamps have not been substituted from another list. If the list has an associated public key, the TEE uses this to verify the signature on the server's request.

4. Starting a new list. If the rate-proof is requested over a new list (e.g., when the user first visits a website), the TEE must also verify that the list name does not appear in any MHT leaves. In this case, the client supplies the TEE with all list names and their most recent hash values. The TEE reconstructs the full MHT and checks that the new list name does not appear. This requires $\mathcal{O}(s)$ string comparisons and hashes for s lists.

5. Updating a list. If the above verification steps are successful, the TEE checks that the new timestamp t supplied by the server exceeds the latest timestamp in the specified list. If so, the TEE adds t to the list and updates the MHT to obtain a new MHT root. The new root is sealed alongside the TEE's group private key. The TEE then produces a

signed rate-proof, using its group private key. The rate-proof includes a hash of the original request provided by the server, thus confirming that the TEE checked the rate and added the server-supplied timestamp. The TEE returns the rate-proof to the client, along with the new sealed MHT root for the client to store. In the above design, the whole process of producing the rate-proof can be performed in a single call to the TEE, thus minimizing the overhead of entering/exiting the TEE.

Reducing Client-Side Storage

The number of timestamps stored by CACTI grows as the client visits more websites. However, in most use-cases, it is unlikely that the server will request rate-proofs going back beyond a certain point in time t_P.

To reduce client-side storage requirements, we provide a mechanism to *prune* a client's timestamp list by merging all timestamps prior to t_P. Specifically, the server can include t_P in any rate-proof request, and upon receiving this, the client's TEE counts and records how many timestamps are older than t_P. The old timestamps and associated intermediate hash values can then be deleted from the database. In other words, the system merges all timestamps prior to t_P into a single count value c_P. The TEE stores t_P and the count value in the database outside the TEE and protects their integrity by including both values in the list information that forms the MHT leaf. Pruning can be done repeatedly: when a new pruning request is received for $t_{P'} > t_P$, CACTI fetches and verifies all timestamps up to $t_{P'}$ and adds these to c_P to create $c_{P'}$. It then replaces t_P and c_P with $t_{P'}$ and $c_{P'}$ respectively.

This pruning mechanism does not reduce security of CACTI. If the server does request a rate-proof going back beyond t_P, CACTI will include the full count of timestamps stored alongside t_P. This is always greater than or equal to the actual number of timestamps; thus, there is no incentive for the server to abuse the pruning mechanism. Similarly, even if a malicious client could trigger this pruning (i.e., assuming the list is not associated to

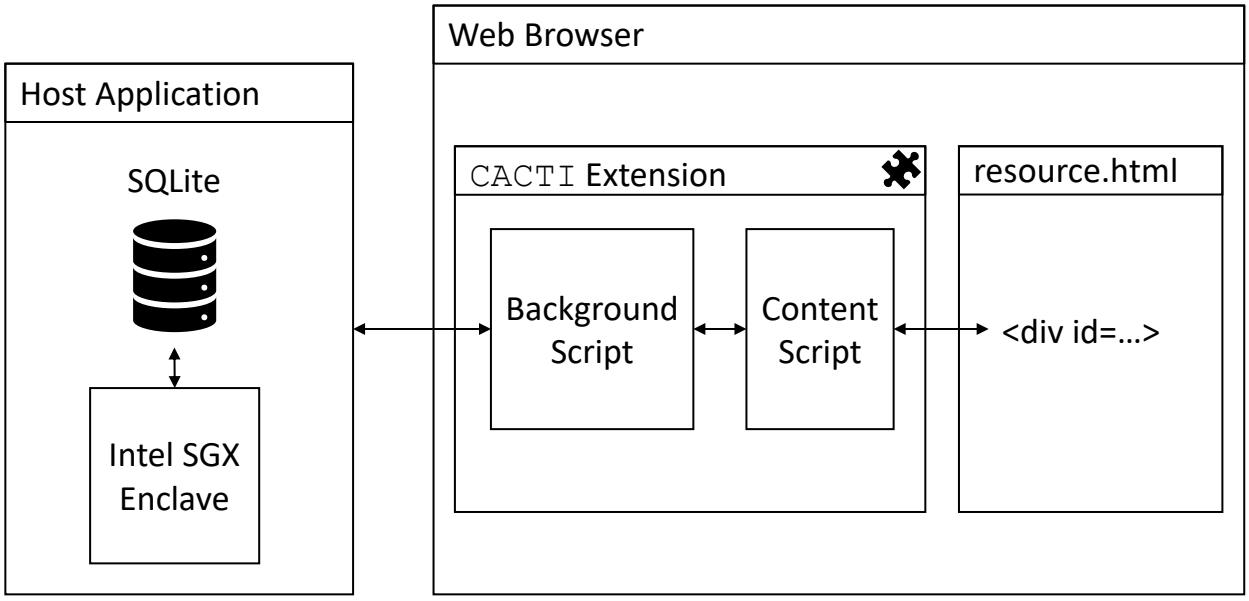


Figure 5.6: Overview of CACTI client-side components.

the server's public key), there is no incentive to do so because it would never decrease the number of timestamps included in rate-proofs.

Since the global list CACTI-GLOBAL is used by all websites, the client is always allowed to prune this list to reduce storage requirements. CACTI blocks servers from pruning CACTI-GLOBAL since this can be used as an attack vector to inflate the client rate by compressing all rates into one value – thus preventing use of CACTI on websites that utilize CACTI-GLOBAL. Thus, we expect pruning of CACTI-GLOBAL to be done automatically by the CACTI host application or browser extension.

5.5 Implementation

We now describe the implementation of the CACTI design presented in the previous section. We focus on proof-of-concept implementations of: client-side browser extension, native host application, and CACTI TEE, as shown in Figure 5.6. Finally, we discuss how CACTI is integrated into websites.

5.5.1 Browser Extension

The browser extension serves as a bridge between the web server and our host application. We implemented a proof-of-concept browser extension for the Chrome browser (build 79.0.3945.130) [99]. Chrome extensions consist of two parts: a content script and a background script.

- **Content script:** scans the visited web page for an HTML `div` element with the id `CACTI-div`. If the page contains this, the content script parses the parameters it contains and sends them to the background script.

- **Background script:** we use Chrome Native Messaging to launch the host application binary when the browser is started and maintain an open port [100] to the host application until the browser is closed. The background script facilitates communication between the content script and the host application.

User notification. The browser extension is also responsible for notifying the user about requests to access CACTI. Notifications can include information, such as server's domain name, timestamp to be inserted, and threshold used to generate the rate-proof. By default, the background script notifies the user whenever a server requests to use CACTI, and waits for user confirmation before proceeding. This prevents malicious websites from abusing CACTI by adding multiple timestamps without user permission (for possible attacks, see Section 5.6.1). However, asking for user confirmation for every request could cause UI fatigue. Therefore, CACTI could allow the user to choose from the following options: (1) *Always ask* (the default), (2) *Ask only upon first visit to site*, (3) *Only ask for untrusted sites*, (4) *Only ask for more than x requests per site per time period*, and (5) *Never ask*. Advanced users can also modify our extension or code their own extension to enforce arbitrary policies for requesting user confirmation. The notification is displayed using Chrome's Notification API [95].

5.5.2 Host Application

The host application running on the client is responsible for: (1) creating the CACTI TEE, which we implement as an SGX enclave, and exposing its `ECALL` API to the browser extension; (2) storing (and forwarding) timestamps and additional integrity information for secure calculation of rate-proofs (to the enclave); and (3) returning the enclave's output to the browser extension.

The host application is implemented in C and uses Chrome Native Messaging [102] to communicate with the browser extension. Since Chrome Native Messaging only supports communication with JSON objects, the host application uses a JSON parser to extract parameters to the API calls. We used the JSMN JSON parser [236]. Moreover, the host application implements the Chrome Native Messaging protocol [94] and communicates with the browser extension using Standard I/O (`stdio`), since this is currently the only means to communicate between browser extensions and native applications.

The host application stores information in an SQLite database. This database has two tables: `LISTS` stores the list names and associated public keys, and `TIMESTAMPS` stores all timestamps and intermediate values of the hash chains. For each rate-proof request, the host application queries the database and provides the data to the enclave.

Since the timestamps are stored unencrypted, we use existing features of the SQLite database to retrieve only the necessary range of timestamps for a given list. Note that since data integrity is maintained through other mechanisms (i.e., hash chains and MHT), the mechanism used by the host application to store this data does not affect the security of the system. Alternative implementations could use different database types and/or other data storage approaches. Instead of hash chains and MHTs, it is possible to use a database managed by the enclave, e.g., EnclaveDB [193]. However, this would increase the amount of code running inside the enclave, thus bloating the trusted code base (TCB).

5.5.3 SGX Enclave

We implemented the TEE as an SGX enclave using the OpenEnclave SDK [164] v0.7.0. OpenEnclave was selected since it aims to unify the programming model across different types of TEEs. The process of requesting a rate-proof is implemented as a single `get_rate ECALL`. For timestamps, we use the UNIX time which denotes the number of seconds elapsed since the UNIX Epoch (midnight 1/1/1970) and is represented as a 4-byte signed integer. We use cryptographic functions from the `mbed` TLS library [22] included in OpenEnclave. Specifically, we use SHA-256 for all hashes and ECDSA for all digital signatures. For EPID signatures, we use Intel EPID SDK (v7.0.1) [121] with the performance-optimized version of Intel Integrated Performance Primitives (IPP) Cryptography library [123]. We use a formally-verified and platform-optimized MHT implementation from EverCrypt [196]. As an optimization, if the MHT is sufficiently small, we can cache fully inside the enclave. When a request for a rate-proof is received, the enclave recalculates the timestamp hash chain and then directly compares the most recent value to the corresponding leaf in the cached MHT, as described in Section 5.4.3.

OpenEnclave currently does not support SGX hardware monotonic counters, so we could not include these in the proof-of-concept implementation. However, a production implementation can easily include hardware counter functionality. Although our implementation uses SGX, CACTI can be realized on any suitable TEE. For example, OpenEnclave is currently being updated to support ARM TrustZone. When this version is released, we plan to port the current implementation to TrustZone, with minimal expected modifications.

5.5.4 Website Integration

Integrating CACTI into a website involves two aspects: sending the rate-proof request to the client, and verifying the response. The server generates the rate-proof request (see Section 5.4.3) and encodes it as `data-*` attributes in the `CACTI-div` HTML `div`. The server also

includes the URL to which the generated rate-proofs should be sent. The browser extension determines whether the website supports CACTI by looking for the `CACTI-div` element. The server implements an HTTP endpoint for receiving and verifying rate-proofs . If the verification succeeds, this endpoint notifies the website and the user is granted access.

Integrating CACTI into a website is thus very similar to using existing CAPTCHA systems. For example, reCAPTCHA adds the `g-recaptcha` HTML `div` to the page, and implements various endpoints for receiving and verifying the responses [104]. We evaluate server-side overhead of CACTI, in terms of both processing and data transfer requirements, in Section 5.6.

5.6 Evaluation

We now present and discuss the evaluation of CACTI. We start with a security analysis, based on the threat model and requirements defined in Section 5.3. Next, we evaluate performance of CACTI in terms of latency and bandwidth. Finally, we discuss CACTI deployability issues.

5.6.1 Security Evaluation

Data integrity & rollback attacks. Since timestamps are stored outside the enclave, a malicious host application can try to modify this data, or roll it back to an earlier version. If successful, this might trick the enclave into producing falsified rate-proofs. However, if any timestamp is modified outside the enclave, this would be detected because the most recent value of the hash chain would not match the corresponding MHT leaf. Assuming a suitable collision-resistant cryptographic hash function, it is infeasible for the malicious host to find alternative hash values matching the MHT root. Similarly, a rollback attack against the MHT is detected by comparing the included counter with the hardware monotonic counter.

Timestamp omission attacks. A malicious application can try to provide the enclave

with only a subset of the timestamps for a given request, e.g., to pretend to be below the threshold rate. Specifically, the host could try to omit one or more timestamps at the start, in the middle, and/or at the end, of the range. If timestamps are omitted at the start, the enclave detects this when it checks that the first timestamp supplied by the host is *prior to* the start time of request t_s. If timestamps are omitted in the middle (or at the end) of the range, the most recent hash value will not match the value in the MHT leaf.

List substitution attacks. A malicious client might attempt to use a timestamp hash chain from a different list, or claim that the requested list does not exist. The former is prevented by including list information (list name and public key) in the MHT leaf. If there is a mismatch between the name and the timestamp chain, the resulting leaf would not exist in the MHT. For the latter, when the host calls the enclave's `get_rate` function for a new list, the enclave checks the names of all lists in the MHT to ensure that the new list name does not already exist.

TEE reset attacks. A malicious client might attempt to delete all stored data, including the sealed MHT root, in order to reset the TEE. Since the group private key received from the provisioning authority is sealed together with the MHT root, it is impossible to delete one and not the other. Deleting the group private key would force the TEE to be re-provisioned by the provisioning authority, which may apply its own rate-limiting policies on how often a given client can be re-provisioned.

CACTI **Farms.** Similar to CAPTCHA farms, a multitude of devices with TEE capabilities could be employed to satisfy rate thresholds set by servers. However, this would be infeasible because: *(1)* CACTI enclaves would stop producing rate-proofs after reaching server thresholds and would thus require a TEE reset and CACTI re-provisioning – which is a natural rate limit; *(2)* the cost of purchasing a device would be significantly higher than CAPTCHA solving costs. For example, currently, the cheapest service charges $1.8 for solv-

ing 1,000 reCAPTCHAs [18][3], while a low-end bare-bones CPU with SGX support alone costs ≈ $70 [125], in addition to the maintenance and running costs.

CACTI Botnets. An adversary might try to build a CACTI botnet consisting of compromised devices with suitable TEEs in order to bypass CAPTCHAs at scale, similar to a CACTI farm. However, if the compromised devices are not yet running CACTI, the adversary would have to provision them using a suitable *PA*, which could be made arbitrarily costly and time-consuming. Alternatively, if the compromised devices are already running CACTI, the adversary gains little advantage because the legitimate users will likely have been using CACTI to create their own rate-proofs. Furthermore, the legitimate user would probably notice any overuse/abuse of their system due to quickly exceeding the thresholds.

Client-side malware. A more subtle variant of the reset attack can occur if malware on the client's own system corrupts or deletes TEE data. This is a type of denial-of-service (DoS) attack against the client. However, defending against such DoS attacks is beyond the scope of this work, since this type of malware would have many other avenues for causing DoS, e.g., deleting critical files.

Other DoS attacks. A malicious server might try to mount a DoS attack against an unsuspecting client by inserting a timestamp for a future time. If successful, the client would be unable to insert new timestamps and create rate-proofs for any other servers, since the enclave would reject these timestamps as being in the past. This attack can be mitigated if the client's browser extension and/or host application simply check that the server-provided timestamp is not in the future.

Client tracking. A malicious server (or group of servers) might attempt to track clients by sending multiple requests for rate-proofs with different thresholds in order to learn the precise number of timestamps stored by the client. A successful attack of this type could potentially

[3]See a comparison of CAPTCHA solving services [197]

reduce the client's anonymity set to only those clients with the same rate. However, this attack is easy to detect by monitoring the thresholds sent by the server. A more complicated attack targeting a specific client is to send an excessive number of successful rate-proof requests in order to increase the client's rate. The goal is to reduce the size of the target's anonymity set. This attack is also easy to detect or prevent by simply rate-limiting the number of increments accepted from a particular server. Note that the window of opportunity for this targeted attack is limited to a single session because malicious servers cannot reliably re-identify the user across multiple sessions (since this is what the attack is trying to achieve). The above attacks cannot be improved even if multiple servers collude.

Rogue *PA*s. A malicious *PA* might try to compromise or diminish client privacy. However, this is prevented by CACTI's use of the EPID protocol [41]. Specifically, due to the BBS+ signature scheme [26] during EPID key issuance, clients' private keys are never revealed to *PA*s. Also, EPID's signature-based revocation mechanism does not require member private keys to be revealed. Instead, signers generate zero-knowledge proofs showing that they are not on the revocation list. Therefore, client privacy does not depend on any *PA* business practices, e.g., log deletion or identifier blinding.

Each website has full discretion to decide which *PA*s it trusts; if a server does not trust the PA who issued the member private key to the TEE, it can simply fall back to CAPTCHAs. This provides no advantage to attackers, and websites can be as conservative as they desire. If higher levels of assurance are required, *PA*s can execute within TEEs and provide attestation of correct behavior; we defer the implementation of this optional feature to future work.

Overall, we claim that CACTI meets all security requirements defined in Section 5.3 and significantly increases the adversary's cost to perform DoS attacks. Specifically, the **Unforgeability** requirement is satisfied since it is impossible for the host to perform rollback, timestamp exclusion, and list substitution attacks. **Client privacy** is achieved because the rate-proof does not reveal the actual number of timestamps included, and is signed using a

group signature scheme.

5.6.2 Latency Evaluation

We conducted all latency experiments on an Intel NUC Kit NUC7PJYH [124] with an Intel Pentium Silver J5005 Processor (4M Cache, up to 2.80 GHz); 4 GB DDR4-2400 1.2V SO-DIMM Memory; running Ubuntu 16.04 with the Linux 4.15.0-76-generic kernel Intel SGX DCAP Linux 1.4 drivers.

Recall that the host application is responsible for initializing the enclave, fetching data necessary for enclave functionality, performing `ECALL`s, and finally updating states according to enclave output. Therefore, we consider the latency in the following four key phases in the host application:

- *Init-Enclave*: Host retrieves the appropriate data from the database and calls `init_mt ECALL` initializing the MHT within the enclave.[4]
- *Pre-Enclave*: Host retrieves the required hashes and timestamps from the database.
- *In-Enclave*: Host calls the `get_rate ECALL`. This phase concludes when the `ECALL` returns.
- *Post-Enclave*: Host updates/inserts the data it received from the enclave into the database.

We investigated the latency impact by varying (1) the number of timestamps in the rate-proof (Section 5.6.2), and (2) the number of lists in the database (Section 5.6.2). We evaluated the end-to-end latency in Section 5.6.2. Unless otherwise specified, each measurement is the average of 10 runs.

Note: The ECDSA and EPID signature operations are, by far, the dominant contributors to

[4]Init-Enclave is done only when the enclave starts.

latency. However, they represent a fixed latency overhead that does not vary with the number of timestamps or servers. Therefore, for clarity's sake, figures in the following sections do not include these operations. We analyze them separately in Section 5.6.2.

Varying Number of Timestamps in Query

We measured the effect of varying the number of timestamps included in the query while holding the number of lists constant. As shown in Figure 5.7, query latency increases linearly with the number of timestamps included in the query. The most notable increase is in the in-enclave phase since this involves calculating a longer hash chain. However, even with 10,000 timestamps in a query, the total latency only reaches ˜40 milliseconds (excluding signature operations).

Varying Number of Lists

Next, we varied the number of lists while holding the number of timestamps fixed at one per list. We considered two separate scenarios: adding a new list and updating an existing list.

Adding a new list. As shown in Figure 5.8, the latency for the pre-enclave phase is lower compared to Figure 5.7. This is because we optimize the host to skip the expensive `TIMESTAMPS` table look-up operation if the host knows that this is a new list. The in-enclave phase increases as the number of lists increases due to the string comparison operations performed by the enclave to prevent list substitution attacks. However, this phase can be optimized by sorting the server names inside the enclave during initial MHT construction. The post-enclave latency is due to the cost of adding entries to the `TIMESTAMPS` table. Figure 5.8 assumes the enclave has already been initialized (see Figure 5.9 for the corresponding init-enclave phase).

Updating an existing list. As shown in Figure 5.9, the latency of the init-enclave phase increases as the number of lists increases. This is expected since the enclave reconstructs

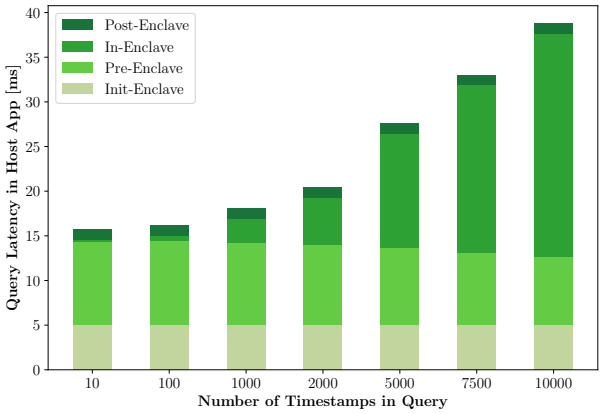


Figure 5.7: Latency of initializing the enclave and creating a rate-proof for different numbers of timestamps in the query (excluding signature operations).

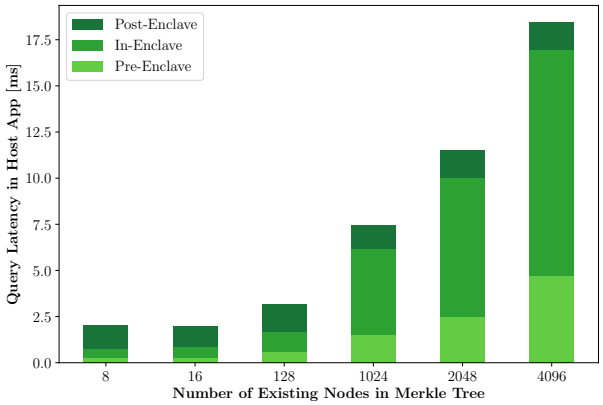


Figure 5.8: Latency of creating the first rate-proof in a new list for different numbers of existing lists (excluding enclave initialization and signature operations).

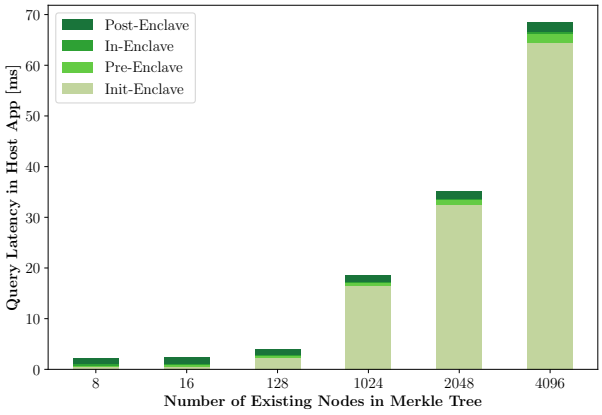


Figure 5.9: Latency of initializing the enclave and updating an existing list for different numbers of existing lists (excluding signature operations).

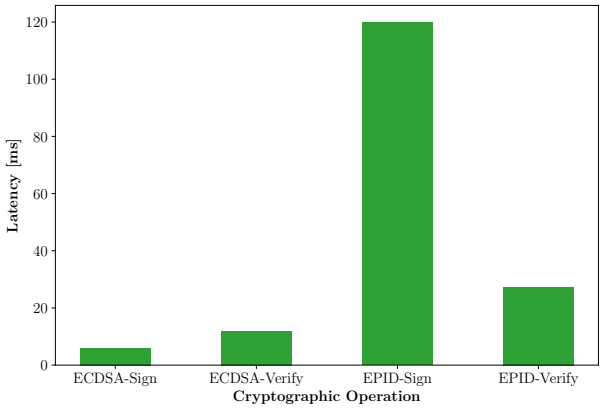


Figure 5.10: Microbenchmarks of signature operations. ECDSA signatures were created and verified using the `mbed` TLS library [22] and EPID signatures with the Intel EPID SDK [121].

the MHT in this phase. The pre-enclave phase also increases slightly due to the database operations.

Signature Operation Latency

Evaluation results presented thus far have not included the ECDSA signature verification or EPID signature creation operations. Specifically, the server creates an ECDSA signature on

Table 5.1: End-to-End Latency of CACTI for different numbers of timestamps and lists. The *Browser* column represents the latency of the browser extension marshaling data to and from the host application. The other columns are as described above.

	ECDSA-Sign	**Browser**	**Pre-Enclave**	**In-Enclave**	**Post-Enclave**	**EPID-Verify**	**Total**
10,000 timestamps in 1 list	6.3 ms	15.2 ms	7.7 ms	181.7 ms	1.0 ms	27.3 ms	239.2 ms
4,096 lists with 1 timestamp each	6.3 ms	15.2 ms	1.8 ms	157.4 ms	2.0 ms	27.3 ms	210.0 ms

the request, which the enclave verifies. The enclave creates an EPID group signature on the response, which the server verifies using the EPID group public key. The average latencies over 10 measurements for these four signature operations are shown in Figure 5.10. We can see that the EPID group signature generation operation is an order magnitude slower compared to the other cryptographic operations including EPID group signature verification. The latency of our enclave is thus dominated by the EPID signature generation operation.

End-to-End Latency

Table 5.1 shows the end-to-end latency (excluding network communication) from when the server begins generating a request until it has received and verified the response from the client. In both settings, the end-to-end latency is below 250 milliseconds. The latency will be lower if there are fewer lists or included timestamps. Compared to other types of CAPTCHAs, image-based CAPTCHAs take ˜10 seconds to solve [45] and behavior-based reCAPTCHA takes ˜400 milliseconds, although this might change depending on the client's network latency.

5.6.3 Bandwidth Evaluation

We measured the amount of additional data transferred over the network by different types of CAPTCHA techniques. Minimizing data transfer is critical for both servers and clients. We compared CACTI against image-based and behavior-based reCAPTCHA [103] (see Fig-

Table 5.2: Additional data received and sent by the client for image-based and behavior-based reCAPTCHA, compared with CACTI.

	Received	Sent	Total
Image-based	140.05 kB	28.97 kB	169.02 kB
Behavior-based	54.38 kB	26.12 kB	80.50 kB
CACTI	0.82 kB	1.10 kB	1.92 kB

ure 5.1). The former asks clients (one or more times) to find and mark certain objects in a given image or images, while the latter requires clients to click a button. To isolate the data used by reCAPTCHA, we hosted a webpage with the minimal auto-rendering reCAPTCHA example [104]. We visited this webpage and recorded the traffic using the Chrome browser's debugging console.

Table 5.2 shows the additional data received and sent by the client to support each type of CAPTCHA. Image-based reCAPTCHA incurs the highest bandwidth overhead since it has to download images, often multiple times. Although not evaluated here, text-based CAPTCHAs also use images and would thus have a similar bandwidth overhead. Behavior-based reCAPTCHA downloads several client-side scripts. Both types of reCAPTCHA made several additional connections to Google servers. Overall, CACTI achieves at least a 97% reduction in client bandwidth overhead compared to reCAPTCHA.

5.6.4 Server Load Evaluation

We analyzed the additional load imposed on the server by CACTI. Unfortunately, CAPTCHAs offered as services, such as reCAPTCHA [103] and hCAPTCHA [129], do not disclose their source code and we have no reliable way of estimating their server-side overhead. Therefore, we compared CACTI against two open-source CAPTCHA projects published on GitHub (both have more than 1,000 stars and have been forked more than a hundred times):

dchest/captcha [74] (Figure 5.11a) generates image-based text recognition CAPTCHAs

(a) dchest/captcha image-based CAPTCHA [74].

(b) produck/svg-captcha image-based CAPTCHAs [194].

Figure 5.11: CAPTCHAs generated using open-source libraries.

consisting of transformed digits with noise in the form of parabolic lines and additional clusters of points. It can also generate audio CAPTCHAs, which are pronunciations of digits with randomized speed and pitch and randomly-generated background noise.

produck/svg-captcha [194] (Figure 5.11b) generates similar image-based text recognition CAPTCHAs, as well as challenge-based CAPTCHAs consisting of simple algebraic operations on random integers. Noise is introduced by varying the text color and adding parabolic lines.

Table 5.3 shows the time to generate different types of CAPTCHAs using the above libraries with typical configuration parameters (e.g., eight characters for text CAPTCHAs). Since CAPTCHA verification with these libraries is a simple string comparison, we assume this is negligible. CACTI's server-side processing is due almost entirely to the EPID signature verification operation. We expect that this time could be improved by using more optimized implementations of this cryptographic operation. Additionally, CACTI uses significantly less communication bandwidth than other approaches, which also reduces the server load (which is not captured in this measurement). Most importantly, the biggest gain of CACTI is on the user side; saving more than ˜10 seconds per CAPTCHA for users.

5.6.5 Deployability Analysis

We analyze the deployability of CACTI by considering changes required from both the server's and client's perspectives:

Server's perspective. The server will have to make the following changes: (1) create and

Table 5.3: Server-side processing time for generating a CAPTCHA and verifying the response.

Library	Type	Time
dchest/captcha	Audio	13.3 ms
	Image-based text	1.7 ms
produck/svg-captcha	Image-based text	2.2 ms
	Image-based math	1.4 ms
CACTI	Rate-proof	33.6 ms

maintain a new public/private key pair and obtain a certificate for the public key, (2) add an additional `div` to pages for which they wish to enable CACTI, (3) create and sign requests using the private key, and (4) add an HTTP endpoint to receive and verify EPID signatures. The server-side deployment could be further simplified by providing the request generation and signature operations as an integrated library.

Client's perspective. The client will have to make the following changes: (1) download and install the CACTI native software, and (2) download and install the browser extension. Although CACTI requires the client to have a suitable TEE, this is a realistic assumption given the large and increasing deployed base of devices with e.g., ARM TrustZone or Intel SGX TEEs.

5.7 Discussion

5.7.1 *PA* Considerations

As discussed in Section 5.4.3, CACTI's use of a provisioning authority (*PA*) provides the basis for client privacy. CACTI does not prescribe the *PA*'s policies. For example, the *PA* has the choice of running the provisioning protocol (Figure 5.2) as a one-off operation (e.g., when installing CACTI) or on a regular basis, depending on its risk appetite. If there are attacks or exploits threatening the Intel SGX ecosystem (and consequently the security of group private

keys), the *PA* can revoke all group member keys. This would force all enclaves in the group to re-register with the *PA*. A similar scenario applies if key-rotation is implemented on the *PA*, e.g., the master secret held by the *PA* is rotated periodically. This forces all enclaves to regularly contact the *PA* to obtain new group member keys. Frequent key-rotation introduces a heavier burden on the clients (although this can be automated), however, provides better security.

5.7.2 EPID

Even though CACTI uses EPID group signatures to protect client privacy, CACTI is agnostic to the choice of the underlying signature scheme as long as it provides signer unlinkability and anonymity. We also considered other schemes, such as Direct Anonymous Attestation (DAA) [39], as used in the Trusted Platform Module (TPM). However, DAA is susceptible to various attacks [148, 202, 40] and, due to its design targeting low-end devices, suffers from performance problems. In contrast, EPID is used in current Intel SGX remote attestation and is thus a good fit for enclaves. Moreover, as mentioned in the previous section, the *PA* must revoke group member keys in the event of a compromise. EPID offers privacy-preserving *signature-based revocation*, wherein the issuer can revoke any key using only a signature generated by that key. Signature verifiers use signature revocation lists published by issuers to check whether the group member keys are revoked. Using this mechanism, CACTI provides *PA*s with revocation capabilities without allowing them to link keys to individual users. *PA*s can define their own revocation policies to maximize their reputation and trustworthiness.

5.7.3 Optimizations

Database Optimizations

As with most modern database management systems, SQLite supports creating indexes in database tables to reduce query times. Also, as discussed in Section 5.6, placing all timestamps for all servers in one table and conducting `JOIN` operations incurs performance overhead. An alternative is to use a separate table per list. However, we presented CACTI evaluation results without creating any indexes or separate timestamp tables in order to show the worst-case performance. Performance optimizations, such as changing the database layout, can be easily made by third parties since they do not affect the security of CACTI.

System-level Optimizations

As a system-level optimization, CACTI can perform some processing steps in the background while waiting for the user to confirm the action. For example, while the browser extension is displaying the notification and waiting for user approval, the request can already be sent to the enclave to begin processing (e.g., loading and verifying the hash chain of timestamps and the MHT). While the enclave creates the signed rate-proof, it does not release the proof or update the hash chain until the user approves the action. This optimization reduces user-perceived latency to that of client-side post-enclave and server-side EPID verification processes, which is less than 14% of the end-to-end latency reported in Section 5.6.2.

Optimizing Pruning

Although it is possible to create another `ECALL` for pruning, this might incur additional enclave entry/exit overhead (see Section 5.4.2). Instead, pruning can be implemented within the `get_rate` `ECALL`. Since `get_rate` already updates the hash chain and MHT, the pruning can be performed at the same time, thus eliminating the need for an additional `ECALL` and hash chain and MHT update.

5.7.4 Deploying CACTI

Integration with CDNs and 3^{rd} Party Providers

Although CACTI aims to reduce developer effort by choosing well-known primitives (e.g., SQLite and EPID), we do not expect all server operators to be experienced in implementing CACTI components. The server-side components of CACTI can be provided by Content Delivery Networks (CDNs) or other independent providers.

CDNs are widely used to reduce latency by serving web content to clients on behalf of the server operator. CDNs have already recognized the opportunity to provide abuse prevention services to their customers. For example, Cloudflare offers CAPTCHAs as a free rate-limiting service [60] to its customers [62]. CACTI could easily be adapted for use by CDNs, which would bring usability benefits across all websites served by the CDN.

In addition, independent CACTI providers could offer rate-proof services that are easy to integrate into websites – similar to how CAPTCHAs are currently offered by reCAPTCHA [103] or hCAPTCHA [129]. These services would implement the endpoints described in Section 5.5.4 and could be integrated into websites with minimal effort.

Website Operator Incentives

There are several incentives for website operators to support CACTI. Firstly, in terms of usability, CACTI can drastically improve user experience by allowing legitimate users to avoid having to solve CAPTCHAs. Secondly, in terms of privacy, some concerns have been raised about existing CAPTCHA services [62]. By design, CACTI rate-proofs cannot be linked to specific users or to other rate-proofs created by the same user. Thirdly, in terms of bandwidth usage, CACTI requires an order of magnitude fewer data transfer than other CAPTCHA systems.

User demand for privacy-preserving systems that reduce the amount of time spent solving

CAPTCHAs has led Cloudflare to offer *Privacy Pass* [73], a system designed to reduce the number of CAPTCHAs presented to legitimate users, especially while using VPNs or anonymity networks [63].

PA Operator Incentives

In CACTI, *PA*s are only involved when provisioning credentials to CACTI enclaves (i.e., not when the client produces a rate-proof). This is a relatively lightweight workload from a computational perspective. *PA*s could be run by various different organizations with different incentives, for example:

1. TEE hardware vendors that want to increase the desirability of their hardware;
2. Online identity providers (e.g., Google, Facebook, Microsoft) who already provide federated login services;
3. For-profit businesses that charge fees and provide e.g., a higher level of assurance;
4. Non-profit organizations, similar to the Let's Encrypt Certificate Authority service.

CACTI users can, and are encouraged to, register with multiple *PA*s and randomly select which private key to use for generating each rate-proof. This allows new *PA*s to join the CACTI ecosystem and ensures that clients have maximum choice of *PA* without the risk of vendor lock-in.

Client-side components

On the client-side, CACTI could be integrated into web browsers, and would thus work "out of the box" on platforms with a suitable TEE.

5.8 Related Work

CACTI is situated in the intersection of multiple fields of research, including DoS (or Distributed DoS (DDoS)) protection, human presence, and CAPTCHA improvements and alternatives. In this section, we discuss related work in each of these fields and their relevance to CACTI.

Network-layer defenses. The main purpose of network-layer DoS/DDoS protection mechanisms is to detect malicious network flows targeting the availability of the system. This is done by using filtering [152] or rate-limiting [54] (or a combination thereof) according to certain characteristics of a flow. We refer the reader to [190] for an in-depth survey of network-level defenses. Moreover, additional countermeasures can be employed depending on the properties of the system under attack (e.g., sensor-based networks [184], peer-to-peer networks [191], and virtual ad-hoc networks [138]).

Application-layer defenses. Application-layer measures for DoS/DDoS protection focus on separating human-originated traffic from bot-originated traffic. To this end, problems that are hard to solve by computers and (somewhat) easy to solve by humans comprise the basis of application-layer mechanisms. As described in Section 5.1, CAPTCHAs [224] are used extensively. Although developing more efficient CAPTCHAs is an active area of research [108, 226, 204, 72], research aiming to subvert CAPTCHAs is also prevalent [171, 234, 91, 89]. In addition to such automated attacks, CAPTCHAs suffer from inconsistency when solved by humans (e.g., perfect agreement when solved by three humans is 71% and 31% for image and audio CAPTCHAs, respectively [45]). [173] suggest that although CAPTCHAs succeed at telling humans and computers apart, by using CAPTCHA-solving services (operated by humans), with an acceptable cost, CAPTCHAs can be defeated. Moreover, apart from questions regarding their efficacy, one other concern about CAPTCHAs is their usability. Studies such as [87, 45] show that CAPTCHAs are not only difficult but also time-consuming for humans, with completion time of ≈10 seconds on average. While less time-consuming

behavioral CAPTCHAs are available, some raise privacy concerns. A prevalent example, reCAPTCHA [103], analyzes user behavioral data (which requires sharing this data with the CAPTCHA provider) and claims to work more efficiently if used on multiple pages. In contrast, CACTI can provide at least the equivalent of abuse-prevention as CAPTCHAs, while minimizing the burden on users and offering strong privacy guarantees.

Human presence detection. Human presence refers to determining whether specific actions were performed by a human. VButton [149] proposes a system design based on ARM's TrustZone [24]. Secure detection of human presence is achieved by setting the display and the touch input peripherals as secure peripherals which can only be controlled by the TEE while VButton UI is displayed. With a secure I/O mechanism in place, user actions can be authenticated to originate from VButton UI by a remote server using software attestation. Similarly, Not-a-Bot [115] designs a system based on TPMs by tagging each network request with an attestation assuring that the request has been performed not long after a keyboard or mouse input by the user. Unfortunately, Intel SGX does not support secure I/O and it is not currently possible to implement similar systems on devices with only Intel SGX support. SGXIO [228] proposes an architecture for creating secure paths to I/O devices from enclaves using a trusted stack that contains a hypervisor, I/O drivers, and an enclave for trusted boot. In addition, an untrusted VM hosts secure applications. The communication between secure applications and drivers is encrypted using keys generated at the end of the local attestation process. Unfortunately, the implementation of this system is not yet available. Fidelius [82] protects user secrets from a compromised browser or OS by protecting the path from the input and output peripherals to the hardware enclave. Similar to SGXIO, this is a promising step towards general-purpose trusted UI. If trusted UI capabilities do become widely available on TEEs, these can complement our CACTI design (e.g., providing stronger assurance of human presence).

Privacy Pass. Privacy Pass [73] implements a browser extension to reduce the burden of

CAPTCHAs for legitimate users when visiting websites served by Cloudflare. When a user solves a CAPTCHA, Cloudflare sends the user multiple anonymous cryptographic tokens, which the user can later "spend" to access Cloudflare-operated services without encountering additional CAPTCHAs Although Privacy Pass significantly benefits benign users, it could still be exploited by CAPTCHA farms. Additionally, Privacy Pass' is currently limited to Cloudflare users.

5.9 Conclusion & Future Work

CACTI is a novel approach for leveraging client-side TEEs to help legitimate clients avoid solving CAPTCHAs on the Web. The unforgeable yet privacy-preserving rate-proofs generated by the TEE provide strong assurance that the client is not behaving abusively. Our proof-of-concept implementation demonstrates that rate-proofs can be generated in less than 0.25 seconds on commodity hardware, and that CACTI reduces data transfer by more than 98% compared to existing CAPTCHA schemes. As for future work, we plan to employ optimization techniques discussed in Section 5.7, implement and evaluate CACTI on ARM TrustZone using OpenEnclave, and explore new types of web security applications that are enabled using client-side TEEs.

Chapter 6

VICEROY: GDPR-/CCPA-compliant Enforcement of Verifiable Accountless Consumer Requests

Abstract

Recent data protection regulations (notably, GDPR and CCPA) grant consumers various rights, including the right to *access*, *modify* or *delete* any personal information collected about them (and retained) by a service provider. To exercise these rights, one must submit a *verifiable consumer request* proving that the collected data indeed pertains to them. This action is straightforward for consumers with active accounts with a service provider at the time of data collection, since they can use standard (e.g., password-based) means of authentication to validate their requests. However, a major conundrum arises from the need to support consumers *without accounts* to exercise their rights. To this end, some service providers began requiring such *accountless* consumers to reveal and prove their identities (e.g., using government-issued documents, utility bills, or credit card numbers) as part of issuing a verifiable consumer request. While understandable and reasonable as a short-term fix, this approach is cumbersome and expensive for service providers as well as privacy-invasive for consumers.

Consequently, there is a strong need to provide better means of authenticating requests from accountless consumers. To achieve this, we propose `VICEROY`, a privacy-preserving and scalable framework for producing *proofs of data ownership*, which form a basis for verifiable consumer requests. Building upon existing web techniques and features, `VICEROY` allows accountless consumers to interact with service providers, and later prove that they are the same person in a privacy-preserving manner, while requiring minimal changes for both parties. We design and implement `VICEROY` with emphasis on security/privacy, deployability, and usability. We also assess its practicality via extensive experiments.

Research presented in this chapter appeared in the Proceedings of the 30th Network and Distributed System Security Symposium (NDSS 2023) [135].

6.1 Introduction

Several new data protection regulations have been enacted in recent years, notably the European Union General Data Protection Regulation (GDPR) [83] and the California Consumer Privacy Act (CCPA) [48]. These regulations grant consumers various new legal rights. For example, consumers gain the right to *access* personal data collected about them and held by service providers (GDPR Art. 15, CCPA 1798.100), *request correction* (GDPR Art. 16, CCPA 1798.106) or *request deletion* of their personal data (GDPR Art. 17, CCPA 1798.105).

Importantly, these regulations expand the definition of *personal data* beyond that associated with a person's real name. For example, GDPR Rec. 30 states that natural persons "*may be associated with online identifiers provided by their devices, applications, tools and protocols, such as internet protocol addresses, cookie identifiers or other identifiers*". This means that any website[1] collecting information about consumers[2] based on identifiers, such as IP addresses or cookies, may be collecting personal information, and thus have to comply with these new regulations. The website must therefore provide a means by which consumers can access, request correction of, or request deletion of their personal information.

When dealing with a consumer request, the website must verify that the requestor is indeed the consumer to whom the personal information pertains. This is critical to prevent erroneous disclosure (which would be a serious violation of any data protection regulation), unauthorized modification, or deletion of personal information. This is called a "*verifiable consumer request*" (VCR) (CCPA 1798.140(y)).[3]

For consumers who have pre-existing accounts on a given website, submitting a VCR is relatively straight-forward. To wit, CCPA 1798.185 requires: "*treating a request submitted*

[1] As shorthand, we use the term *website* to represent the entity operating a website, which (we assume) falls into the category of entities that the GDPR and CCPA call *controller* and *business*, respectively.

[2] We use *consumer* or *client* to refer to: (1) the GDPR term *data subject*, (2) the CCPA term *consumer*, and (3) the equivalent terms in other regulations.

[3] These requests are sometimes also referred to as Subject Rights Request (SRR) or Subject Access Request (SAR).

through a password-protected account maintained by the consumer ... as a verifiable consumer request". However, there remains a major challenge of how to support a VCR from casual or **accountless** consumers, as required by CCPA 1798.185, while protecting such consumers' privacy.

The only mechanisms that are currently suitable for accountless consumers are those that require: a device cookie, a government-issued ID, a signed (and possibly witnessed) statement, a utility bill, a credit card number, or taking part in a phone interview [189]. However, these mechanisms are cumbersome (and some are time-consuming) for consumers as well as insecure, as demonstrated by prior work [189, 46, 75, 34]. Such mechanisms also typically require manual processing, which is both error-prone and costly. Moreover, such methods (apart from device cookies) are privacy-invasive for the consumer and open the door for further consumer data exposure. For example, a government-issued ID or utility bill reveals even more private information to the website.

In light of these issues, the most appealing choice appears to be the use of device cookies. Cookies are already used pervasively by websites to link multiple sets of activities (sessions) to the same consumer. At first glance, asking for device cookies as part of a VCR appears to meet the GDPR and CCPA requirements: only the authorized consumer should possess the correct cookie (*unforgeability*), and providing a cookie does not reveal additional information (*privacy*). However, this essentially means treating device cookies as *authentication tokens*, which has at least three disadvantages:

First, in the general case, there is no requirement for a cookie's value to be unguessable. A recent large-scale study [93] found cookie values containing URLs, email addresses, timestamps, and even JSON objects. Although *authentication cookies* are designed to be unguessable, these would typically only be used once the user has logged into an account.

Second, cookies are used (i.e., sent over the network) whenever the consumer interacts with

the website. Although secure communication channels (e.g., TLS) can protect cookies in transit, the MITRE ATT&CK framework lists several recent examples of techniques for stealing web cookies [168].

Third, consumers must protect the cookies stored on their devices, especially considering e.g., client-side spyware, even after the cookies expire. Secure storage may become more challenging as the number of stored cookies increases since consumers should not delete cookies for which they might subsequently issue a VCR. If an adversary can guess or obtain the cookie through any of the above vectors, they would be able to request, modify, or delete all the consumer's data.

Motivated by aforementioned issues, this chapter constructs `VICEROY`, a first-of-its-kind framework that allows *accountless* consumers to request their data in a private manner, while allowing website operators to efficiently and securely verify such requests. `VICEROY` introduces a one-to-one mapping between Web sessions and consumer-generated public keys. At session initiation, the consumer generates a public key (a VCR public key) and supplies it to the server. At a later time, the consumer digitally signs their request using the private key corresponding to the VCR public key for the session.

To ensure consumer privacy, our key derivation mechanism uses unlinkable public keys derived from a single master public key. This also allows `VICEROY` to only require consumers to securely store a single private key, regardless of the number of sessions they have generated. Moreover, since this private key is only needed when generating VCRs, it can be protected using well-known secure key storage mechanisms (e.g., hardware security devices).

`VICEROY` is composed of well-known cryptographic primitives. However, to meet the necessary requirements of security, scalability, and privacy, this must be done through the careful selection of such primitives. Furthermore, `VICEROY`'s design prioritizes deployability, requiring only minimal changes to existing websites and no changes to existing cookie usage.

The contributions of this work are:

- Design of VICEROY– a secure, scalable, and privacy-preserving VCR mechanism for accountless consumers.
- Careful selection of cryptographic protocols to balance the three requirements of VICEROY.
- Proof-of-concept implementation of VICEROY for web browsers, including a VICEROY-compatible hardware security device.
- Thorough security, performance, and deployability evaluation of VICEROY.

Organization. Section 6.2 presents background on GDPR/CCPA and verifiable consumer requests (VCRs). Next, Section 6.3 presents our threat model and defines requirements for VICEROY. Sections 6.4 and 6.5 then describe the design and proof-of-concept implementation of VICEROY. Section 6.6 presents our evaluation methodology and results. Further aspects of VICEROY are discussed in Section 6.7 and related work is overviewed in Section 6.8. Section 6.9 concludes the chapter.

Code Availability. Source code for all VICEROY components and the Tamarin model is available at [213].

6.2 GDPR/CCPA Background

This section overviews Personally Identifiable Information and consumer rights under the GDPR and the CCPA. Given familiarity with GDPR and CCPA, it can be skipped without any loss of continuity.

6.2.1 Personally Identifiable Information (PII)

Both GDPR and CCPA pertain to the combination of *personal and personally identifiable* information, often referred to as: *Personally Identifiable Information*[4] or PII.

Information is *personal* if it relates to a person, e.g., contact information, geolocation, applications and devices used, how an application is used, interests, websites visited, consumer-generated content, identities of people with whom a consumer communicated, content of communication, audio, video, and sensor data [136].

Information is *personally identifiable* if the person to which it pertains is either identified or identifiable. If information is paired with a name, telephone number, email address, government-issued identifier, or postal address, then it is considered to be personally identifiable [134]. If information is paired with an IP address, a device identifier (e.g., an IMEI), or an advertising identifier, it is *likely* to be considered personally identifiable [134]. Information paired with an identifier created by a business (e.g., a cookie) is personally identifiable if it can be combined with other information to allow the consumer to whom it relates to be identified [134].

6.2.2 Rights of Access and Erasure

Both GDPR and CCPA require a business that collects PII to disclose, typically in its privacy policy, the categories of PII collected, the purposes for collecting it, and the categories of entities with which that PII is shared [134]. Both regulations give consumers the right:

- To learn about, and control, information relating to them that a business has collected. Specifically, consumers have the right to request access to the *specific pieces of PII* that the business has collected (GDPR Art. 15; CCPA Sec. 1798.110(a)(5)).

- To request that their incorrect PII be corrected (GDPR Art. 16; CCPA Sec. 1798.106).

[4]The GDPR uses the term *personal data* and the CCPA uses *personal information.*

- To request that a business delete their PII (GDPR Art. 17; CCPA Sec. 1798.105).

6.2.3 Verifiable Consumer Requests (VCRs)

The consumer's rights to access, and request correction or deletion of, their PII are contingent upon verification that the consumer is indeed the person to whom that PII relates. However, GDPR and CCPA differ in requirements of methods of verification. Both regulations require a business to use reasonable measures to verify the consumer's identity (GDPR Rec. 64; CCPA Sec. 1798.140(ak)). If a consumer has a password-protected account with a business, both require a business to treat requests submitted via that account as verified (GDPR Rec. 57; CCPA Sec. 1798.185(a)(7)).

However, both regulations also recognize that PII is often collected about casual consumers, who do not have password-protected accounts. In this case, they envision a consumer request being verified by associating additional consumer-supplied information with PII that the business previously collected about that consumer (CCPA Sec. 1798.130(a)(3)(B)(i)). The CCPA further specifies that any information provided by the consumer in the request can be used solely for the purposes of verification (CCPA Sec. 1798.130(a)(7)). However, if a business has not linked PII to a consumer or a household, and cannot link it without the acquisition of additional information, then neither the GDPR nor the CCPA requires a business to acquire additional information to verify a consumer request (GDPR Rec. 57; CCPA Sec. 1798.145(j)(3)). Thus, some requests may be unverifiable.

The CCPA [47] recognizes that consumer verification is not absolutely certain. It establishes two thresholds of certainty. The lower threshold, called *reasonable degree of certainty*, may be satisfied by matching at least two pieces of information provided by the consumer (CCPA Regs. §999.325(b)). The higher threshold, called *reasonably high degree of certainty*, may be satisfied by matching at least three pieces of information provided by the consumer, and obtaining a signed declaration from the consumer (CCPA Regs. §999.325(c)). However,

other means of verification may also satisfy these thresholds. Verification of the consumer identity must always, at a minimum, meet the *reasonable degree of certainty* threshold. Furthermore, requests to learn specific pieces of PII must meet the *reasonably high degree of certainty threshold*. Finally, a consumer may choose to use a third-party verification service (CCPA Regs. §999.326).

The design of a verification method should balance the administrative burden on the consumer (CCPA Sec. 1798.185(a)(7)) with the likelihood of unauthorized access and the risk of harm (CCPA Regs. §999.323(b)(3)).

6.3 Threat Model and Requirements

Our system model assumes a typical Web environment with two types of principals: *(1)* clients and *(2)* servers. Clients are *consumers* who access Internet services offered by servers. Servers collect and store data during interactions with consumers by associating such data with identifiers issued to the consumer. Each client can own multiple devices and at least one of the client's devices can be trusted to store a secret, e.g., a private key. This trusted device could be a smartphone, a dedicated key storage device, or a secure hardware wallet. All access to the secret is controlled by the client. Physical and side-channel attacks against the trusted client device are beyond the scope of this work.

We assume secure communication channels between clients and servers, which can be realized using standard means, e.g., HTTPS. Use of secure channels to deliver web content has become a de-facto standard, as shown by Felt et al. [85], which reports that up to 87% of all webpages were served via HTTPS in 2017. This number is expected to increase, as shown by Google's 2022 Transparency Report [107], which claims that 80–98% of top-100 websites use HTTPS. Moreover, standards such as DTLS [200] and QUIC [131] allow devices that cannot use TCP to establish similarly secure channels.

We consider three types of adversarial behavior:

- **Malicious clients:** Attempt to impersonate other clients in order to perform operations on data that is not theirs.
- **Client-side malware:** Attempts to perform unauthorized operations on client data without client's knowledge. We assume that the client's trusted device is free of malware, while all other client devices can be potentially infected.
- **Honest-but-curious servers:** Attempt to identify clients who submit requests, or to link multiple requests to the same client. Multiple servers might collude to link client requests and/or to learn client identities.

As usual, we assume all relevant cryptographic primitives are implemented and used correctly and cannot be attacked via side-channels or any other weaknesses. Similarly, we assume digital signatures can only be generated by the true owner of the private key.

Based on the above system model, we define the following requirements for `VICEROY`:

- **Unforgeability:** Only the client who originally interacted with the server can create a valid VCR.
- **Replay resistance:** A server will only accept a valid VCR at most once.
- **Consumer Privacy:** An honest-but-curious server (or a set thereof) should be unable to link a VCR to a specific client, or to link multiple VCRs to the same client.

6.4 `VICEROY` Design & Challenges

This section discusses `VICEROY`'s goals, design features, and challenges encountered. Note that we use the terms *client* and *consumer* interchangeably.

6.4.1 Design Motivation

One straight-forward way to support VCRs from accountless consumers is to require them to provide the same cookie(s) they were issued when originally visiting the server website. (Indeed this is one of the mechanisms that [189] encountered in their survey of how businesses respond to access requests.) The rationale is that only the consumer from whom the data was collected should have access to the cookie, which ties all consumer activity that constitutes one session. This method has several advantages: First, it is *privacy-preserving* in that, when making a request, the consumer does not reveal any further personal information that the server didn't already have. Furthermore, if the consumer submits multiple VCRs based on different cookies, the server cannot link them.[5] Second, this mechanism is easily deployable, since cookies are supported by virtually every device that uses the Web.

However, per Section 6.1, there are also several significant disadvantages: First, this method essentially makes cookies into *symmetric* authentication tokens: anyone in possession of the cookie can create VCRs. This is problematic because cookies, in general, are not required to be unguessable and may contain predictable information, such as URLs or email addresses [93]. A subset of cookies, namely *authentication cookies*, are designed to be unguessable, however, these would typically only be used once the consumer has logged into an account (i.e., no longer an accountless consumer). Second, cookies are used in all interactions with the website, and several techniques for stealing cookies have been demonstrated [168]. Third, since consumers visit many different websites, they would have to *securely* and *reliably* store a potentially large number of cookies. This differs from the usual client-side cookie management since cookies would be additionally valuable as a means to issue VCRs. Also, if cookies are lost (e.g., due to disk failure), the consumer would be unable to exercise their GDPR/CCPA rights. This underscores the importance of cookie storage reliability. Furthermore, if the server for any reason also stores copies of cookies, the same security requirements

[5] Potential "fingerprinting" of the consumer's browser or network interface notwithstanding.

apply. If these cookies are leaked as a result of a data breach, the server would have to invalidate them in bulk, thus preventing legitimate consumers from submitting VCRs, or risk attackers requesting consumers' personal data.

6.4.2 Conceptual Design

Motivated by aforementioned challenges, we construct `VICEROY` to avoid the drawbacks discussed. We now describe key features of `VICEROY`.

Asymmetric tokens. When interacting with a server, a client provides the server with the public part of an asymmetric key-pair called the *VCR public key*. Upon receiving a VCR public key, the server associates this key with a particular *session*. Generally, a session is any linkable set of interactions between a client and a server. For example, in the Web context, a session most likely corresponds to an HTTP(S) session, which is managed using cookies. To protect the client's privacy, a new VCR public key, which is unlinkable to any previous keys, can be used for each session. Finally, to submit a VCR for a particular session, the client creates a request and signs it using the corresponding VCR private key for that session.

This approach addresses the drawbacks of using only cookies to authenticate the request since the client's signature is assumed to be unforgeable and the client's VCR private keys are never sent over the network. It does not matter if the adversary learns the VCR public keys. The use of digital signatures also allows additional information/parameters to be cryptographically bound to the request (e.g., a request to correct personal information could, in some cases, already include the corrected information).

Cookie wrappers. At first glance, mapping data collected during a session to the VCR public key seems to be an efficient way of storing such data on the server side, especially when the consumer submits a VCR. However, from a deployability perspective, it would be infeasible to replace existing Web cookies with public keys because this would require

non-trivial modifications to the way servers use cookies. For example, cookie values containing URLs, email addresses, timestamps, and even JSON objects have been observed in practice [93].

To avoid changing the existing and ubiquitous cookie mechanism, `VICEROY` introduces the concept of a *cookie wrapper* – a cryptographic binding between an existing server-generated session identifier (e.g., cookie) and a client-generated VCR public key. The server generates a cookie wrapper by signing the hash of the server-generated cookie and the VCR public key using the server's long-term wrapper signing key. This allows the client to verify whether the wrapper was created correctly. This wrapper is created contemporaneously with the cookie, and at most one wrapper is created per cookie. The wrappers are then sent to and stored by the client alongside the cookie and VCR public key. The use of cookie wrappers significantly improves deployability by allowing servers to add support for `VICEROY` without modifying current cookie management.

Submitting VCRs. When the client issues a VCR signed with the relevant VCR private key (as described above), the client also sends the corresponding cookie wrapper to the server, along with the request. The server first verifies that the wrapper is valid, by verifying its own signature on the wrapper. If valid, the server then uses the VCR public key specified in the wrapper to verify the client's signature over the request. If this in turn is valid, the server is assured that this request was generated by the same client who received the original cookie (i.e., the legitimate consumer).

6.4.3 Design Challenges

The conceptual design described above presents several design challenges. This section outlines the main challenges and presents the key insights used to realize `VICEROY`.

Avoiding Key Explosion

For privacy reasons, the client cannot use a static public key. A static public key would allow a server (or a set thereof) to link together multiple sessions by the same client. This linkage could take place at session initiation time, i.e., when the client requests a wrapper, or when the client issues a VCR. Also, if a client's static public key is leaked, it becomes possible to track that client's sessions globally. However, requiring each client to have a distinct public/private key-pair per session can cause a "key explosion" in which the client has to manage the large number of public keys and more importantly, securely store many private keys.

To avoid this issue, we use the concept of *derivable asymmetric keys*, specifically, the key derivation scheme used in Bitcoin Improvement Proposal (BIP) 32 [231]. This type of key derivation scheme allows a chain of *child public keys* to be derived from a single *parent public key*. Importantly, the derivation of public keys does not require access to the corresponding parent private key. Furthermore, the corresponding child private keys can only be derived from the parent/master private key. We denote the *derivation path* of a key as `a/b/c/...`, where `a/b` is the b^{th} child key of `a`, and `a/b/c` is the c^{th} child key of `a/b`. This approach minimizes public key storage requirements of `VICEROY` – only the parent public key must be stored, while all other public keys can be derived. When a new session is initiated, the parent public key is used to generate a new child public key.

Multiple Devices

The client may interact with websites from multiple different devices and may subsequently want to issue VCRs for one or more of these sessions.

By design, `VICEROY` allows clients to use any number of devices with a single trusted device. Specifically, the master private key is used to generate a new *device public key* which is stored on each of the client's devices. The device public key can in turn be used for generating all

other VCR public keys needed on the device. Note that even though the device public keys are derived from the same master private key, they are unlinkable and thus cannot be used by websites to link together sessions from the client's different devices.

Secure Key Management

VCR private keys for each session must be stored in a secure environment, access-controlled by the client. Leakage of private keys would allow an adversary to issue VCRs for the client's sessions, thus giving them the ability to learn, modify, or delete, potentially sensitive information.

Our use of derivable asymmetric keys reduces the number of private keys that need to be stored securely to just one – the master private key. Another benefit of using derivable asymmetric keys is that the master private key can be stored offline. This is because the master private key is only needed when generating a new device public key (i.e., when enrolling a new device) or creating VCRs, which are expected to be relatively infrequent operations.

This feature provides `VICEROY` significant flexibility in terms of how the master private key is stored, in order to accommodate different levels of security. For example, at one end of the spectrum, clients with low-security requirements can simply store their keys on any device they trust, e.g., a phone or laptop. Clients with higher security requirements can store their keys in hardware-backed keystores, such as the Android Keystore [16] or Apple Secure Enclave [19]. On PCs, clients could make use of hardware-enforced enclaves, such as Intel SGX [128] or Windows Virtualization-base security (VBS), to protect the keys. At the top end of the spectrum, clients with the highest security requirements could store their keys in hardware security devices (e.g., YubiKey [235], Solokey [211], or Ledger [145]). These clients may also enforce additional physical security controls, such as keeping the hardware security device in a locked safe until it is needed. Clients can also make back-up copies of

their master private keys to allow recovery if the trusted device fails.

We emphasize that a separate trusted device is *recommended* and not mandated. The only requirement for the trusted device is that it can derive child private keys and create signatures. Section 6.5 shows that these requirements can be met even by resource-constrained hardware security devices.

Long-Term Storage

A VCR may be submitted long (possibly, years) after the corresponding session ends. This typically requires clients to store state information for numerous sessions in a secure and highly available manner. Naturally, the amount of state per session must be minimal.

In `VICEROY` neither the cookies themselves nor the wrappers can be used to issue VCRs without a signature from the client's master private key. Therefore, the security requirements for the *storage* of cookies and wrappers are minimal – the integrity and availability of the cookies and wrappers must be maintained. Importantly, these pieces of information do not need to be kept confidential (assuming the cookies themselves have expired and are no longer useful, e.g., for authentication).

This opens up potential new business opportunities for third-party *cookie storage* providers to offer a service for safely storing cookies and wrappers on clients' behalf. This service can take care of all cookie and wrapper management as well as provide API endpoints to their customers. Note that security requirements for, and trust burden on, such services would be significantly higher if cookies alone were sufficient to issue VCRs. Also, third-party cookie storage is not a requirement; clients who are uncomfortable with third-part providers storing their cookies can store them on their local devices. Clients may also use their preferred cloud-storage service, e.g., OneDrive.

Broad Application Support

Many non-browser applications also communicate with application-specific servers, which, similarly to Web servers, collect consumer data. `VICEROY` is sufficiently flexible to be used in these applications as well. This is achieved through the design pattern of using wrappers instead of directly modifying the session identifiers. For example, if a non-browser application uses a different form of session identifier (i.e., not a Web cookie), `VICEROY` can still be used since the wrapper can be used to cryptographically bind the client's VCR public key to any type of session identifier.

6.4.4 Overall `VICEROY` Design

Bringing together the design concepts discussed in the preceding sections, this section presents the overall design of `VICEROY`. The precise protocol messages exchanged between the various principals are shown in Figure 6.1 and the various cryptographic keys are defined in Table 6.1.

Setup and device provisioning

The client first generates a master private key $sk(t)$ on a trusted consumer device t. This key is then used to derive a device-specific public key for each device the client will use for interacting with websites, e.g., $pk(t/i)$ for device i. Device public keys are provisioned using a simple request-response protocol, as shown in Figure 6.1 (**A**). The device public keys are stored within the respective devices for rapid future VCR key generation.

Website Interaction

When initiating a session with a server, upon a client request, the server generates a unique client id and sends it to the client in the form of a cookie. The client generates a new VCR public key $pk(t/i/j)$ for session j, derived from the device public key. The client then sends this newly-generated key, along with the client id cookie, to the server as shown in

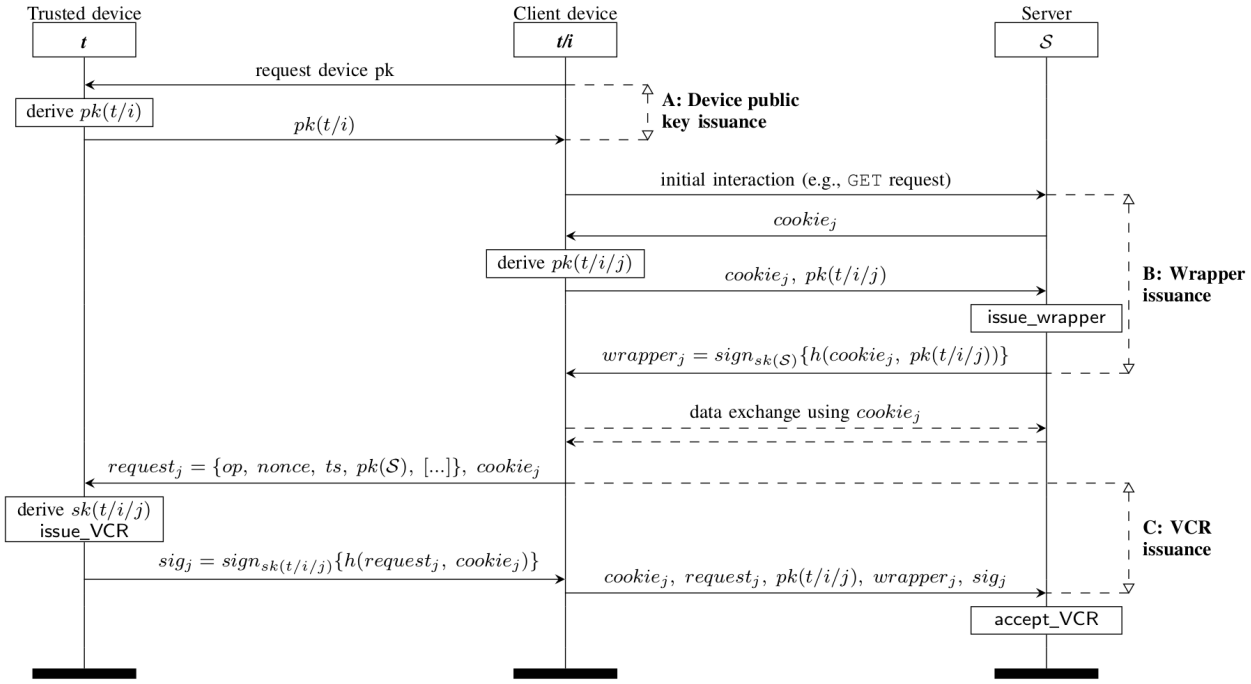


Figure 6.1: Protocol messages exchanged in VICEROY. $sign_k(m)$ denotes a cryptographic signature on message m using key k, and $h(m)$ denotes a cryptographic hash of message m. The events issue_wrapper(j), issue_VCR(j), and accept_VCR(j) are used in the formal security analysis in Section 6.6.

Figure 6.1 (**B**). After receiving the VCR key, the server signs the hash of the VCR key and cookie to generate a wrapper. Then it returns the wrapper to the client, attesting to the association between the client id cookie and the VCR key. To check the integrity of the wrapper, the client verifies the signature using the server wrapper public key which can be obtained and verified the same way it obtains and verifies the server's TLS public key.

VCR Issuance

As shown in Figure 6.1 (**C**), the client generates a request, which consists of the specified operation (e.g., retrieve, modify, or delete), the current time, and any optional parameters. The client uses the master private key $sk(t)$ on the trusted device to derive the respective signing key $sk(t/i/j)$ for device i and session j. The signing key $sk(t/i/j)$ is then used to sign a hash of the request and the associated cookie, which is returned to the client. The client then sends this signature, along with the cookie, request, public key, and wrapper to

Table 6.1: List of keys and their role in VICEROY

Key	Name and description
$sk(t)$	Master private key: generated and stored in trusted device.
$pk(t/i)$	Device i public key: derived from master private key $sk(t)$ within the trusted device, and stored on device i.
$pk(t/i/j)$	VCR j public key: derived from device public key $pk(t/i)$ when requesting a new wrapper.
$sk(t/i/j)$	VCR j private key: derived from master private key $sk(t)$ within the trusted device when issuing a VCR.
$sk(\mathcal{S})$	Wrapper signing key: long-term private key used by server when generating a wrapper.
$pk(\mathcal{S})$	Wrapper public key: long-term public key used by client when verifying a wrapper. Obtained via standard PKI.

the server, constituting the VCR.

Upon receiving a VCR, the server first verifies its own signature on the wrapper to confirm the authenticity of the wrapper. The server then verifies the client's signature on the VCR using the public key $pk(t/i/j)$ from the wrapper. If these checks succeed, the server accepts the VCR and proceeds with the requested data operation.

To prevent an attacker from replaying a valid VCR to the server, the client includes a unique *nonce* in the request, which is thus also included in the client's signature sig_j in Figure 6.1. Upon receiving this VCR, the server checks that it has not already processed a VCR containing that *nonce*. The client also includes a timestamp ts in the request, representing the time at which the VCR was issued. Each server defines its own recency threshold (e.g., 12 hours) and rejects any VCRs that are older than this threshold. This means that the server only has to store nonces for up to this threshold in order to check that new requests are unique. This also ensures that an attacker cannot delay valid VCRs arbitrarily (e.g., if the attacker were able to block a VCR and then release it months later, this could have unintended consequences for consumers).

An alternative would be to use a challenge-response protocol where the server generates a challenge that the client must include in the signed VCR. Although this would avoid

the need for a nonce and timestamp, the complexity of the system increases and could be abused to mount a denial of service attack against the server, similar to a TCP SYN flooding attack (although well-known cookie-based and application-level countermeasures such as CAPTCHAs [225] or rate-limiting mechanisms [176] could also be used.).

Device Unprovisioning

Finally, the full device lifecycle may require *unprovisioning*, e.g., if the device is lost/stolen, or sold/recycled. We separately consider the implications for a regular or trusted device.

Regular Device. Since such devices do not hold private VCR keys, unprovisioning is straight-forward. The client only has to unlink the old device from the respective trusted device to prevent any further VCR issuance. This can be done from the trusted device, even if the old device is lost/stolen. The client may wish to back-up or transfer any cookies and wrappers from the old device.

Trusted Device. From an availability perspective, the client should be able to recover the master private key from a backup. From a security perspective, the trusted device should ideally have some type of access control (e.g., using a PIN and/or a fingerprint) to protect the private key even from an adversary who has physical access to the device. If the trusted device is being sold/recycled, the client should securely back-up or transfer the master private key to a new trusted device using techniques such as Presence Attestation [238].

6.5 Implementation

We now describe our implementation of `VICEROY`, which consists of: a browser extension, a trusted device implementation, and a modified web server.

6.5.1 Browser Extension

This component provides most of the client-side functionality on a regular consumer device. Specifically, it manages VCR public keys, implements client-side aspects of the VCR flow, and includes a client-facing interface for controlling this flow. It also handles communication between the browser and the trusted device (or application) that holds the master private key. Although we chose to implement a proof-of-concept extension for Google Chrome, no (or only minor) mods would be required to get it to work with any modern browser.

To realize derivable asymmetric keys, we used a mechanism proposed for hierarchical deterministic wallets, commonly known as Bitcoin Improvement Proposal 32 [231], or `BIP32`. `BIP32` has the notion of an *extended key*, with 256 bits of entropy (called *chain code*) added to a normal public/private key-pair. Extended keys can be used to derive one or more child keys, following the rule that private keys can be used to derive private or public keys, while public keys can only derive public keys.[6] `BIP32` is also well-suited for low-end devices, since it was designed for (resource-constrained) Bitcoin hardware wallets. Section 6.5.3 describes our proof-of-concept of a resource-constrained trusted device.

The browser extension comprises a background and a pop-up script, both written in `JavaScript`, with additional `HTML` and `CSS` for the pop-up. As no `JavaScript` version of several `Node.js` libraries (e.g., `crypto`, `BIP32`) were available, we used Browserify [43] to convert such libraries into `JavaScript` files that can be loaded by the browser.

Background Script

This component houses most VCR-side functionality. It uses the browser's API (`chrome.webRequest.onHea`
to scan `HTTP` response headers to detect which servers support `VICEROY`. If present, it parses the relevant `VICEROY` endpoints and the client id cookie from the headers.

[6] `BIP32` also has the notion of *hardened* vs. *non-hardened* keys, though we only use the latter. The difference between these two is the algorithm used when deriving them from the parent key.

Using the device public key, it derives a session-specific VCR public key using a port of the official `BIP32` implementation [33]. The derivation path of a VCR public key is in the form `t/i/j`, where `t` denotes the master private key, `i` the device ID, and `j` the total number of sessions created on the device. In other words, `t/i` represents the device public key and `t/i/j` represents the child public key for the j^{th} session.

After deriving a VCR public key, the background script either includes this in the next request header sent to the server or sends a `POST` request to a separate server-defined VCR wrapper request endpoint (we implemented the latter). The server then responds with a newly generated wrapper.

The background script then stores the server-returned wrapper along with the key derivation path. Since we generate a new one for each session, the number of stored wrappers may grow large, depending on how many new sessions the client establishes. However, the storage overhead of the wrappers is not significant, as the number of wrappers is at most the same as the number of cookies the client must store (see Section 6.6.4 for client-side storage evaluation). To improve efficiency of searching for a client id cookie, we use a hashmap of client id cookies and their corresponding wrapper information. We also store the URL of the website and the time of the visit alongside the wrapper to assist clients when selecting a session during VCR issuance. The background script can store this data using any storage service. Our implementation uses the local storage API (`chrome.storage.local`). Other choices include cloud storage services or Google Chrome's synced storage API (`chrome.storage.sync`), which would allow clients to synchronize `VICEROY` data between different devices.[7]

Note that all the above operations are performed asynchronously, in the background. Thus, the client does not experience any additional latency in loading the page.

[7]Unfortunately, we found that Chrome synced storage currently imposes a limit on the amount of stored data.

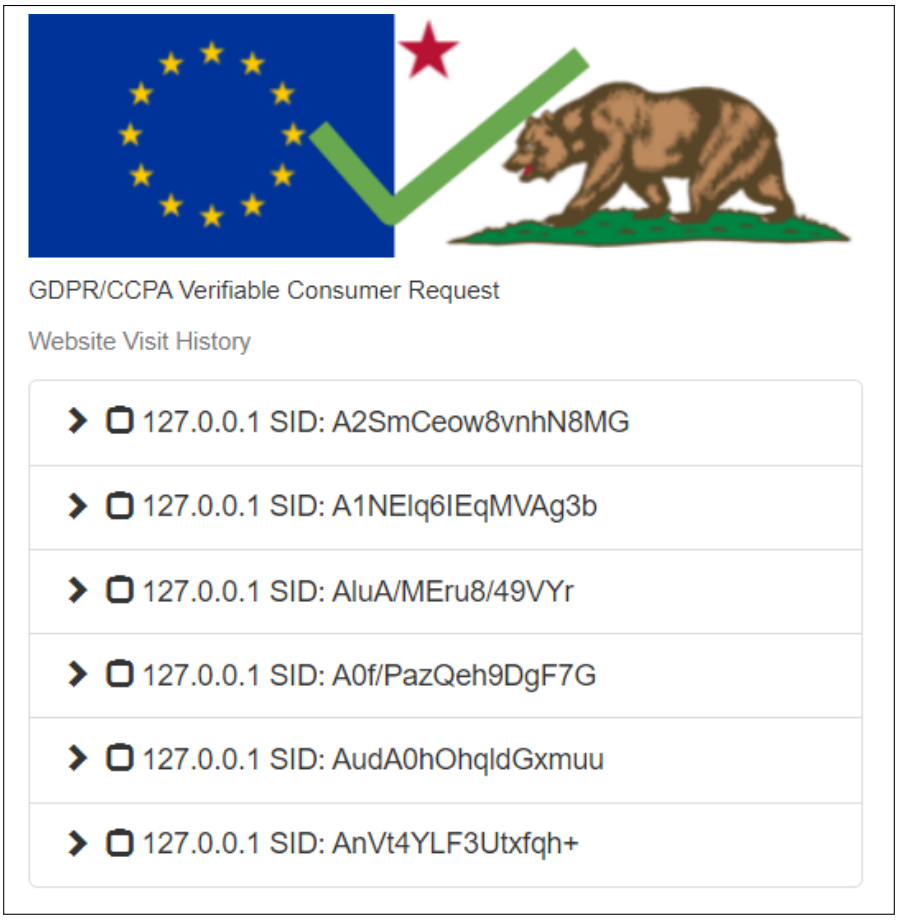


Figure 6.2: `VICEROY` browser extension pop-up displaying multiple sessions. `SID` is the first few bytes of VCR public key.

Pop-up

The extension pop-up is a small web page that appears when the client clicks on the extension icon next to the URL bar. As shown in Figures 6.2 and 6.3, it displays session information for a VCR key along with the history of web links visited when the session was active. It also displays the types of VCRs (access, modify, and delete) that the client can submit.

To issue a VCR, the client first chooses the session(s) and the type of request. Next, the pop-up script prepares a request for signing. It includes a timestamp in the signature to prevent replay attacks, due to its simple design. The resulting client request is then passed to a Python native application [56] which relays it to the trusted device (see Section 6.5.3) for signing. Once signed using the VCR private key, the request is retrieved by the pop-up

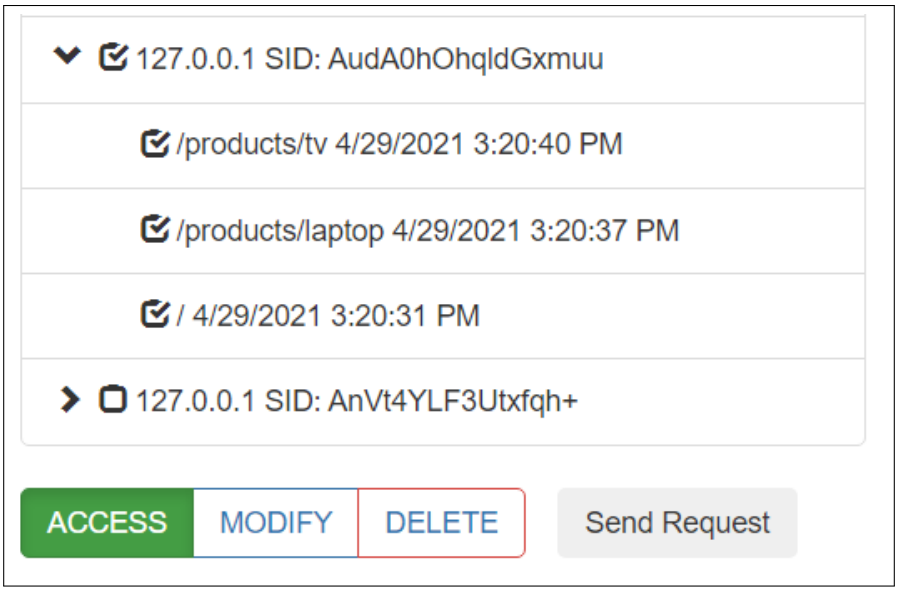


Figure 6.3: `VICEROY` browser extension pop-up displaying the history of visits in each session. Clients can select which session and which type of VCR (`ACCESS/MODIFY/DELETE`) they wish to generate.

(using the background script) through the native application, and the VCR is sent to the server's VCR verification endpoint. Finally, the server's response is displayed in the client's browser.

6.5.2 Native Messaging Application

To simulate a client-controlled trusted device, we created a `Node.js` [179] application that holds the master private key. Alike the trusted device, this application signs VCRs received from the background script using private keys derived from the master private key with the provided key derivation path, e.g., `t/0/1`. Communication between this application and the browser is done via the native messaging protocol [56]. We use the `native-messaging` package [210] to support both Firefox and Chrome. As mentioned in Section 6.4.3, the key storage mechanism must support derivable asymmetric key operations (e.g., using `BIP32`). There are publicly available implementations of `BIP32` in different languages, including: JavaScript [33], Golang [203], Python [130], Java [180] and C [218].

Figure 6.4: Solokey hardware security token in use.

Figure 6.5: Solokey token compared to a standard AA battery.

6.5.3 Trusted Device

We implemented a proof-of-concept trusted device using the Solokey Hacker [211], a security token with open-source software and hardware. Solokey includes an STM32L432KC microprocessor with an Arm Cortex-M4 MCU (80MHz), 64 kB of RAM, 256 kB of flash memory, a true random number generator (TRNG), and a physical button for presence attestation, as shown in Figures 6.4 and 6.5. We extended the FIDO2 Client to Authenticator Protocol (CTAP) API on the device with the following three calls:

- `KEYGEN`: Generates a master private key using the TRNG. This key never leaves Solokey.
- `DEVKEY`: Takes a key derivation path (`t/i`) and outputs a generated device public key (see Figure 6.1 (**A**)).
- `VCRGEN`: Takes a key derivation path (`t/i/j`) and a consumer request (in the form of a cryptographic hash), and outputs a signed VCR (see Figure 6.1 (**C**)).

All three trusted device API calls require human confirmation achieved by the client pressing the physical button on the Solokey. For key derivation and signing we used the `BIP32` implementation from Trezor [218] due to its popularity and suitability for embedded devices. The native messaging application calls the individual API according to the command it receives from the background script. Once the client confirms the action by pressing the button and the native messaging application receives a response from Solokey, the application relays the response back to the background script.

To protect against tampering with the consumer request by client-side malware, the trusted device could also display information about the VCR that is about to be generated, including the request type, the website URL, and the timestamp. In our prototype, this could be as simple as changing LED color on the Solokey. Displaying more detailed information about the request may require trusted devices to have a dedicated, human-perceivable output means, e.g., a display.

6.5.4 `VICEROY`-enabled Web Server

We implemented a proof-of-concept `VICEROY`-enabled `HTTP` server using the Express Node.js web framework [214]. It uses `HTTP` sessions and indexes data collected during each session using session cookies. For signing, it uses ECDSA with curve `secp256k1` [32], although any secure signature scheme can be used.

When the client first visits a web page hosted on our server, the latter creates an `HTTP` cookie that includes a client id (uuid [181]).Hereafter, all data collected by the server about this client is associated with the client's unique id.[8] In response to the initial client request, the server notifies the client that it supports `VICEROY`.

The server provides the client with an `HTTP` endpoint for obtaining a wrapper. The client's

[8] `VICEROY` can also use any existing client identifier cookie scheme – such as Google Analytics' `_ga` and `_gid` cookies [98].

browser extension sends the client id cookie set by the server and a freshly-generated VCR public key to this endpoint, and the server then generates a wrapper that cryptographically binds these two pieces of information. This endpoint can be configured to only issue wrappers for a short duration after the cookie was issued, to avoid adversaries attempting to bind their own public keys to arbitrary cookies. Alternatively, the client can provide a freshly-generated VCR public key in the next `HTTP` request, and the server can issue the wrapper in the next response.

The server also provides the client with a VCR endpoint, to which the client can later submit VCRs for this website. As discussed in Section 6.4.4, the client sends a wrapper and a signed VCR, which the server then verifies. To support requested VCR actions, the consumer's request may include metadata specific to the requested action. For instance, for data access requests, metadata may include an encryption key to be used by the server to encrypt data to be returned to the client.

6.6 Evaluation

We now evaluate security of `VICEROY` as well as its latency, data transfer, and storage requirements.

6.6.1 Security Analysis

To evaluate security of `VICEROY`, we defined a formal specification of the protocol using the Tamarin prover [160, 29]. The full specification is provided in Listing A.1 in the Appendix. In Tamarin, a protocol P is modeled as a set of labeled transition rules operating on *facts*. A transition consumes linear facts from state s_{i-1}, generates new facts for state s_i, and labels the transition as a. An execution of protocol P is a finite sequence of states and transition labels $(s_0, a_1, s_1, \ldots, a_n, s_n)$ such that $s_0 = \emptyset$ and $s_{i-1} \xrightarrow{a_i} s_i$ for $1 \leq i \leq n$. The sequence of transition labels $(a_1, \ldots, a_n)$ is a *trace* of P and the set of all traces is denoted $traces(P)$.

Security properties are specified using first-order logic formulas on traces.

Unforgeability. Using this specification, we formally define the security property of *unforgeability* (see Section 6.3), for both wrappers and VCRs. As shown in Figure 6.1, let $\mathsf{issue\_wrapper}(s, o, pk)$ denote server s issuing a wrapper for cookie o and public key pk (corresponding to private key sk); let $\mathsf{issue\_VCR}(c, o, pk, r)$ denote client c issuing a VCR for request r, cookie o, and public key pk; and let $\mathsf{accept\_VCR}(s, o, pk, r)$ denote server s accepting a VCR for request r, cookie o, and public key pk.

Definition 6.1 (Wrapper unforgeability)**.** *A protocol P satisfies the property of* wrapper unforgeability *if for every $\alpha \in traces(P)$:*

$$\forall s, o, pk, r, j.\ \mathsf{accept\_VCR}(s, o, pk, r) \in \alpha_j \implies \exists i.\ \mathsf{issue\_wrapper}(s, o, pk) \in \alpha_i \wedge i < j$$

Definition 6.2 (VCR unforgeability)**.** *A protocol P satisfies the property of* VCR unforgeability *if for every $\alpha \in traces(P)$:*

$$\forall s, c, o, pk, r, j.\ \mathsf{accept\_VCR}(s, o, pk, r) \in \alpha_j \implies \exists i.\ \mathsf{issue\_VCR}(c, o, pk, r) \in \alpha_i \wedge i < j$$

The Tamarin prover verifies that, in the protocol as specified, both of these properties hold for an unbounded number of protocol runs (the strongest possible result). Since wrappers and VCRs are ultimately verified by the server, Definition 6.1 requires that, whenever a server accepts a VCR, that server must have issued a corresponding wrapper at some prior time point. Similarly, Definition 6.2 requires that, whenever a server accepts a VCR, there must exist a client that issued that VCR at some prior time point. Intuitively, these properties show that the adversary cannot forge a valid wrapper or VCR for a given pk. Since the corresponding private key is only known to the client, we conclude that the unforgeability property is satisfied.

Replay resistance. To prevent an attacker from obtaining a genuine VCR (e.g., by eavesdropping) and later replaying it to the server, we formally define the security property of *replay resistance* (see Section 6.3) for VCRs.

Definition 6.3 (Replay resistance)**.** *A protocol P satisfies the property of* replay resistance *if for every $\alpha \in traces(P)$:*

$$\forall s, o, pk, r, j.\ \mathsf{accept\_VCR}(s, o, pk, r) \in \alpha_i \wedge \quad \mathsf{accept\_VCR}(s, o, pk, r) \in \alpha_j \implies i = j$$

The Tamarin prover verifies that this property holds for an unbounded number of protocol runs. Definition 6.3 states that, if there are two trace events in which a server accepts the same VCR, these must be the same event. Since Tamarin does not model time-based properties, the formal model uses only a nonce in the VCR to check for uniqueness. In practice, the client would also include a timestamp in the VCR, as described in Section 6.4.4.

Consumer/Device/Request Linking. To protect clients' privacy, an honest-but-curious server should be unable to link a VCR to a specific client or to link multiple VCRs to the same client (see Section 6.3). This requirement ensures that the use of `VICEROY` does not reveal any additional information to the server about potential links between users, devices, and sessions (e.g., if a single user is using multiple devices).

Since unlinkability is not a trace property, we cannot use Tamarin to model or verify this property. Instead, we follow an existing approach for reasoning about unlinkability [223, 222, 188] and show that the messages sent by the client to the server do not contain any information that could be used by a server to link VCRs to clients or to other VCRs.

As shown in Figure 6.1, the only new pieces of information provided by the client (which the server does not already know) are (1) the *request*, (2) the VCR public key ($pk(t/i/j)$), and (3) the client's signature. The *request* does not contain any information that uniquely

identifies the client or allows it to be linked to other requests. Formally, given any two requests, a server would not be able to distinguish whether or not they were issued by the same client. `BIP32` guarantees that derived public keys are unlinkable to each other and to their parent keys. This ensures that neither the VCR public key nor the signature created using the corresponding private key are linkable.[9] Formally, given any two derived public keys or signatures, a server would not be able to distinguish whether or not they were issued by the same client. We can therefore conclude that the protocol satisfies the *unlinkability* property.

Of course, if the device public key is leaked from the client's device, different VCR could be linkable. However, even in this case, `VICEROY` is no worse than the current use of web cookies, since an attacker that can steal a device public key could also steal cookies from the victim's device and use these to link/track the victim's sessions and VCRs.

Although `VICEROY` provides unlinkability by design, the nature of how VCRs are submitted may point servers in the direction of clients. For instance, by observing metadata such as IP addresses of different VCR requests, a server might link them to the same client. One possible mitigation is to use anonymity networks (e.g., Tor) and avoid issuing VCR *bouquets* whereby multiple requests are submitted through the same connection. Random delays between VCRs can be used to prevent timing-based correlation.

Public Key Injection. The adversary might attempt to replace the client VCR public key with its own public key when obtaining a wrapper for a cookie. This can occur if there is either: *(1)* an active network-level adversary, and/or *(2)* malware on client device. For (1), this is mitigated by the use of secure communication channels (e.g. TLS) or by simply having `VICEROY` browser extension compare the public key it sent with the public key in the returned wrapper. In contrast, (2) is difficult to defend against. Malware in full control of

[9]By design, a VCR can be linked to the corresponding wrapper – indeed the latter is included in the former.

the client device can replace the client public key with its own during the wrapper request. Even if the consumer attempts to verify the public key in the returned wrapper, the malware can subvert this check. The only way to prevent this is to verify wrappers in an environment isolated from client-side malware, e.g., a Trusted Execution Environment (TEE). Of course, this approach would also require securely sharing the public key to be verified with the TEE and displaying the verification result to the user without malware interference. Overall, we consider (2) to be out of scope since malware in total control of a client device already has access to any data that could be collected by the server about the client.

Client-Side Malware. Malware on the client's device might conduct unauthorized operations on client data. This might be possible either via: *(1)* replay attacks, or *(2)* by generating VCRs without the owner's consent. This issue highlights one important difference between using asymmetric tokens and symmetric tokens for VCRs. The former allows generation of one VCR per client authorization. In contrast, symmetric tokens, even if encrypted and decrypted on demand with client's approval, need to be available in plaintext at some point in order to be sent to the servers. Client-side malware can use these exposed tokens to generate future VCRs. Also, using symmetric tokens allows malware to access data from *before* its infection period. For example, assume that malware infects the client's device at time t. It can access pre-stored tokens and learn data generated *prior* to t. Since we can prevent such attacks via asymmetric tokens, `VICEROY` provides better security and privacy compared to current symmetric token-based systems.

Key Leakage. An attacker might exploit weaknesses in `BIP32` to learn the private key. One well-known weakness of `BIP32` is that knowledge of a parent extended public key as well as of any non-hardened child private key (descended from that parent public key) can leak the parent extended private key [231]. However, in `VICEROY`, the non-hardened child private key is generated within, and never leaves, the trusted device. Therefore, the attacker must compromise the trusted device to obtain the non-hardened child private key, which we

consider to be infeasible, per Section 6.3.

6.6.2 Latency Analysis

Most operations in VICEROY result in no user-perceptible latency because they occur asynchronously with normal web browsing. Nevertheless, we discuss them to quantify computational costs of VICEROY. The only user-perceivable latency occurs when a VCR is issued, which is expected to be an infrequent operation. For these experiments, we used as the client device an Intel NUC with an Intel Core i5-7260U 2.20GHz quad-core CPU with 32.0 GB of RAM running Ubuntu 18.04 LTS, with Chrome version 97.0.4692.71 64-bit official build. Unless otherwise stated, all results are averages over 10 runs, with storage left unchanged between runs. All data was in local storage and results may vary depending on the underlying storage technology, e.g., memory vs. hard-drives vs. cloud-hosted databases.

Obtaining a Wrapper. We divide the process of obtaining a wrapper into the following four phases:

1. *Key Derivation*: When an unknown client makes a request, the server returns a cookie and the VCR endpoints. The client parses these endpoints, derives a VCR public key, and prepares a wrapper request.

2. *Wrapper Generation*: The server generates a wrapper using the client-provided VCR public key and cookie.

3. *Wrapper Verification*: After receiving the wrapper from the server, the client verifies the wrapper using the server's public key and confirms that the wrapper associates the correct VCR public key and cookie.

4. *Wrapper Storage*: The client saves the wrapper along with the public key derivation path and endpoints.

Table 6.2: Latency Results for VICEROY Wrappers.

Key Derivation	Wrapper Generation	Wrapper Verification	Wrapper Storage
24.6 ms	0.4 ms	18.8 ms	6.5 ms

Table 6.3: VCR Latency Results.

	VCR Generation	VCR Verification
VCR Flow	1357.4 ms	1.5 ms

Table 6.2 shows the average time of each phase. These measurements exclude network latency, as this will vary depending on the locations of the client and server. Wrapper Verification takes the longest as it includes a signature verification. Wrapper Generation is noticeably faster than Key Derivation and Wrapper Verification since the former is performed by a native application and the latter two run in the browser extension.

Issuing a VCR. We divide the process of issuing a VCR into two steps:

VCR Generation: When a client selects a session and a VCR type, the browser extension prepares a request to be signed by the trusted device and a key derivation path. Both are sent to the trusted device via the native messaging application. Next, the trusted device signs the overall request using the private key corresponding to the derived public key. Finally, the signature is then returned to the extension. The above steps in total take on average 1357.4 ms using a modified Solokey Hacker as the trusted device.

VCR Verification: The server receives the VCR and verifies the wrapper. Also, it extracts the VCR public key for this session from the wrapper and verifies the overall VCR using the VCR public key. This step takes on average 1.5 ms.

Table 6.3 shows latency results (excluding network latency). Similar to Table 6.2, a signature generation operation performed by the native messaging application takes longer, compared to a standalone server. This is due to data passing delay between pop-up and background

scripts, as well as the native messaging protocol between the application and the browser. Based on these results, server's cost to verify VCRs is minimal.

Trusted Device Latency. Finally, we benchmarked the latency of key-generation operations performed by the trusted device. Generation of the master private key takes 332 ms and generation of a device key takes 724 ms, which are both reasonable for these types of operations. As discussed in Section 6.5, we use a modified Solokey Hacker as an example of a trusted device, while noting that this resource-constrained hardware token is likely to be the slowest type of a trusted device. We emphasize that these are very infrequent operations, taking place once per trusted device, and once per new client device respectively.

6.6.3 Data Transfer Analysis

We measured the amount of data transferred to obtain wrappers and issue VCRs. For demonstration purposes, our server kept the visit history for each client, which was returned to the client upon successful VCR verification. The setting was the same as for latency analysis, and we used the browser's debugging console to measure the amount of data exchanged between the browser and server.

The client first sent an `HTTP GET` request to the server. After receiving the VCR endpoints and a cookie, the client sent a `POST` request to the wrapper request endpoint. The client then generated and sent a VCR to the server requesting to access the data, and received the visit history with a single entry. This request included: wrapper, VCR public key, and signature on the request. Table 6.4 shows the `HTTP` header and payload sizes transmitted between the client and server. For both obtaining a wrapper and issuing a VCR, the amount of data exchanged was a fraction of a typical web interaction.

Table 6.4: Data transfer in kB (HTTP header + payload).

	Request	**Response**	**Total**
Obtain Wrapper	0.72 kB (0.62 + 0.10)	0.38 kB (0.23 + 0.15)	1.10 kB
Issue VCR	0.99 kB (0.68 + 0.31)	0.28 kB (0.23 + 0.05)	1.27 kB

6.6.4 Storage Analysis

For the client-side data storage evaluation, we measured client-side data storage requirements using the `chrome.storage.sync.getBytesUsed` function. The date was represented using the UNIX standard and the only the URL path was stored in the history section.

We first measured the minimum storage for a client with no visit history: it stores server endpoints, VCR and server public keys, plus metadata used to issue VCRs, requiring 0.38 kB in total. We then measured storage for a client who visited a particular URL 100 times, where `VICEROY` stored the details of each visit. This required 5.06 kB. Although storage increases linearly with the number of visited web pages, the gradient is small and the overall magnitude is similar to that of a typical web browser history.

We can extrapolate from the results above to estimate the amount of storage required by a typical client to store all wrappers generated over a long period of time. Crichton et al. [66] recently found that, on average, a user visits 163 distinct web pages per day. Note that "distinct web page" refers to a unique URL and not necessarily to a unique domain, i.e., the number of distinct domains may be smaller. This study does not report the fraction of web pages where a user has an account. We make the following conservative assumptions: (i) the 163 web pages correspond to distinct domains; (2) the user has no accounts on any of these web pages; and (3) the user stores all wrappers for one year before issuing a VCR request.[10] Under these assumptions, `VICEROY` would require 0.38 kB for each of the 163 web pages for 365 days, resulting in a total storage requirement of 22.61 MB.

[10]Note that this analysis is purely illustrative and that the storage requirements could increase or decrease if these assumptions change.

6.6.5 Deployability Analysis

From the server perspective, main changes are: (1) create and maintain a public/private key-pair for generating wrappers, and (2) create and maintain relevant endpoints. By design, the server does not have to change how it assigns identifiers to clients or uses cookies. The integration of `VICEROY` into existing servers can be further simplified by releasing `VICEROY` modules for popular server frameworks. From the server's perspective, `VICEROY` is a cheaper and simpler approach to complying with data protection regulations, compared to existing third-party identity verification services.

From the client's perspective, `VICEROY` requires: (1) generating a master private key on a trusted device and (2) installing the browser extension and native messaging application on other devices. These software packages could be made available via popular app stores. Once installed, `VICEROY` operates transparently to the client and does not disrupt the normal flow of web browsing. The anticipated incentives for clients to use `VICEROY` are that it is both more automated and more privacy-preserving than current identity verification methods.

6.7 Discussion

6.7.1 Multi-Device Support

Given the proliferation of smartphones, computers, and various IoT gadgets im many spheres of everyday life, we expect that most clients own (or soon will own) multiple devices with varying capabilities. `VICEROY` has been designed with this scenario in mind. First, computational requirements of `VICEROY` can be met by any device that can establish TLS connections. Any device that can perform a TLS handshake is sufficiently powerful to verify the signature on a wrapper. Second, storage is not an issue because wrappers can be stored anywhere. This also allows devices without a display to request wrappers and commit these to synced storage. Thereafter, any other device with an appropriate display owned by the same client

can fetch these and perform data operations. Third, the use of BIP32 allows VICEROY to generate an arbitrary number of device public keys, which can be used to derive any number of VCR public keys for interaction with different websites. Since private keys are not stored on such devices, there is no threat against security of VICEROY, even if the number of devices increases. Fourth, VCRs do not need to be issued from the same device that originally interacted with the server. Devices may need to be unprovisioned, although the client should still be able to issue VCRs for interactions they made on those devices. In VICEROY, the client can always transfer cookies and wrappers between devices.

6.7.2 Multi-VCR Support

Unlinkability of VCRs is critical for protecting consumers' privacy (as defined in Section 6.3). However, it requires clients to generate and sign VCRs for each session. To reduce overhead, a client can amend its key derivation mechanism. For example, if a client prefers to use only one VCR to refer to combined collected data for all sessions with a particular website, it can use the derivation path `t/i/s/j` where `t/i` is the derivation path of the device key as before, `s` is a server id and `j` is a server-specific (rather than global) session counter. The client then collects all wrappers and generates a unified VCR by signing it with the private key corresponding to the server VCR public key (`t/i/s`). This server VCR public key is then sent to the server. The server derives VCR public keys for individual sessions and verifies all wrappers. For a new session with the same server, the client simply updates the server id (`s`) and repeats the process.

6.7.3 Multi-Communication Protocol Support

So far, we focused on VICEROY being used over HTTP(S), the most common way to access Web services. However, it can support any stateless protocol that assigns a unique identifier by using that identifier when generating a wrapper. Thus, as described in Section 6.4.3, VICEROY is also applicable to applications that use other protocols to interact with online

servers. Such applications can use VICEROY wrappers to bind client-generated public keys to any type of symmetric session identifier, and the same protocol to issue VCRs for data associated with that identifier.

6.7.4 Shared Devices

Certain types of client devices may be shared by multiple individuals, e.g., a smart TV used by all household members. In the worst case, it may not be possible to associate usage data with a specific individual, e.g., if two or more people are using the device concurrently. This a general *policy* question in the field of data protection; it is not unique to VICEROY. There is no clear guidance in either GDPR or CCPA as to how to handle such situations. However, VICEROY provides some mechanisms that could be used to assist with *enforcing*, rather than *defining*, data ownership policies for shared devices. For example, one conceivable data ownership policy for a shared device is that *all* users of the device must consent to VCRs being issued for sessions originating from this device. In the example above, this would require all smart TV users to consent to a data access request, which may also help address the policy question of who actually *owns* the data. This could be achieved by using a signature scheme that requires all users to participate in the creation of a VCR. The actual policy and procedures regarding data from shared devices should be defined by the regulators, while tools such as VICEROY should enable, rather than dictate, policy.

6.7.5 3^{rd} Party Storage

One distinctive feature of VICEROY, as opposed to simply using cookies, is that possession of a wrapper alone is insufficient to issue a VCR. Thus, wrappers can be stored by third parties and retrieved only when needed. This relaxes client-side storage requirements and creates a possible new business opportunity for (paid) service providers that manage wrappers on behalf of clients.

6.7.6 Broad Identifier Support

VICEROY wrappers are a general means of binding client identification cookies to client-generated public keys. Importantly, this method is *non-invasive* and does not impose any constraints on the cookie, which is useful if a server changes its identification method, e.g., cookie content. VICEROY can also support future identification methods, as a server can simply issue wrappers that bind public keys to any new type of identifier, instead of the cookies.

6.7.7 3^{rd}-party Cookie Support

Third-party cookies are claimed to provide better, more personalized advertisements. Although such cookies are commonly considered detrimental to privacy [157], they are widely used; around 79% of 109 million web pages include third-party cookies [187]. VICEROY can support such cookies by modifying the browser extension to capture traffic going to third parties and extract and store all third-party cookies. To obtain the wrappers, the client can either: (1) visit the third-party wrapper endpoints individually, or (2) send all cookies to the first-party server, which would obtain wrappers on the client's behalf. Note that VICEROY does not require enabling third-party cookies in order to function. This is related to *cookie syncing* [1, 50, 106, 220] in which, instead of placing multiple cookies on the client device, a set of servers associate the data they collect under a unified identifier. VICEROY is capable of supporting this type of cookie and obtaining the wrapper by visiting the relevant endpoint.

6.7.8 Further privacy considerations

When a consumer issues a VCR for a particular session, a potential risk arises that this action could reduce consumer privacy by allowing the service provider to link multiple sessions to the same consumer. For example, there is a one-to-one mapping of a VCR public key and a session, thus revealing VCR keys to the servers might allow servers to link VCRs as well.

To prevent this, we consider two approaches:

First, for data access requests, cryptographic techniques, such as Private Information Retrieval (PIR) [55], can hide the identity (i.e., VCR public key) of the data requested. For modify and delete operations, the problem is more challenging, because, if data is in plaintext, servers could detect what has been updated/deleted. Furthermore, PIR is likely to incur a high bandwidth burden, since databases may retain data for very long periods of time (e.g., 10 years) and that large database might need to be sent to the client.

An exciting approach to overcome both these problems is to introduce Trusted Execution Environments (TEEs) on the server side. Using remote attestation [127], after ensuring that expected code is running on a server-side TEE, clients can create a secure channel to the TEE and send their VCR keys over it. The TEE can find the matching row in the database and return associated data. This TEE-secured database can be populated with data collected during a secure connection between a client and a server, e.g., using techniques such as LibSEAL [27]. Server-side TEEs also allow servers to prove to clients how their data is used, also using remote attestation. VCR responses can be generated with such guarantees, providing more transparency and trust between clients and servers.

6.7.9 Further Applications

Although `VICEROY` focuses on VCRs from accountless clients, it can also supplement verification of VCRs from account-holding clients. This can be useful, considering that passwords suffer from dictionary attacks and are often re-used on multiple servers.

`VICEROY` can also be used as a basis in scenarios that require client re-authentication. For example, in the context of monetary transactions, receipts are currently used to prove that a client bought something from a merchant (server) in order to accept returns or perform exchanges. With `VICEROY`, a client can supply a fresh VCR public key during the purchase

transaction and later generate proof of ownership of the corresponding private key, which would anonymously confirm to the server that this is indeed the same customer.

6.8 Related Work

Supporting VCR requests from accountless consumers. Until now, the only means of authenticating accountless consumers have been *ad hoc*. [189] reports that such means may require one or more of: device cookies, government-issued IDs, signed and witnessed statements, utility bills, credit card numbers, or participation in a phone interview. However, these mechanisms are burdensome for consumers. Furthermore, they are insecure (as shown in [189, 46, 75, 34]), error-prone (due to the manual processing), and privacy-invasive due to the additional information collected. In contrast, `VICEROY` allows consumers to submit VCRs in a secure and private manner without requiring any human interaction on the server side.

Security of GDPR Subject Access Requests. As described in Section 6.2, the GDPR and CCPA grant subjects the right to request access to their personal data collected by businesses, by submitting a VCR or Subject Access Request (SAR). Unfortunately, insecure (or easily circumventable) SAR verification practices open the door to potential leakage of personal data to unauthorized third parties. Prior work [46, 75, 189] has investigated various social engineering techniques for bypassing existing SAR verification practices.

Cagnazzo et al. [46] demonstrated that an unauthorized adversary can abuse the functionality provided by a business to update a victim subject's email and residential addresses. The adversary could then request access to *"their"* data from this new address. Out of 14 organizations tested, 10 gave out personal information and 7 of these contained sensitive data.

Di Martino et al. [75] investigated the use of address spoofing techniques (e.g., using ho-

moglyphs), as well as more sophisticated techniques such as manipulation of identity card images. It was found that 15 out of 41 organizations with manual verification processes leaked personal data. The remaining 14 organizations required an account-based login, which was impervious to such attacks, however is not available for the accountless consumers we consider in this work.

Pavur and Knerr [189] performed an extensive evaluation of 150 companies' practices for SAR verification. Results indicated that email address-based and account login were the most common, followed by device cookies, government IDs, and signed statements. Some organizations also requested utility bills, phone interviews, or credit card numbers. To bypass SAR verification, [189] created and sent a vague SAR letter to organizations. Out of 150, 24% disclosed personally-identifying information.

Boniface et al. [34] also analyzed SAR verification practices for popular websites and third-party trackers. The findings were that, in addition to possibly being insecure, SAR verification could undermine the privacy of subjects in order to verify the request.

General Studies on GDPR Subject Access Requests. Urban et al. [219] performed a two-sided study of both data subjects and data-collecting organizations, with a focus on online advertising. For data subjects, consumer surveys were used to evaluate the usability of data transparency tools offered by the organizations and to learn more about consumers' perceptions of these tools. [219] also conducted surveys and interviews with organizations to get their views on the privacy regulations and business practices for SARs. The results paint a picture of a discrepancy between the consumer's perspectives and the collected data, which is also corroborated by Ausloos and Dewitte [28]. Furthermore, consumers seemed to show little interest in seeing raw technical data. Similarly, Urban et al. [220] investigated SAR practices for online advertising companies and used cookie IDs to request collected data. This approach is similar to the symmetric approach in Section 6.4. [220] reported that some companies requested ID cards or affidavits, while others directly used the cookie IDs in the

browser. Neither approach proves that the requestor is really the consumer about whom the data was collected.

Kröger [141] studied mobile applications and observed an even more fragile ecosystem with discontinued apps and disappearing consumer accounts while processing SARs. Another conclusion of this study was to move away from email-initiated and manual processes which are prone to errors. In terms of compliance, the analysis by Herrmann and Lindemann [116] showed 43% compliance with access requests vs. 57% compliance with deletion requests.

In addition to the above, Dabrowski et al. [68] investigated cookie usage and how it is affected by privacy regulations, reporting that 11% of EU-related websites set cookies for US-based consumers, though not for EU-based consumers. Furthermore, up to 46.7% of websites that appear in both the 2016 and 2018 Alexa top 100,000 sites stopped using persistent cookies without consumer permission. In the standardization realm, Zimmeck and Alicki [242] focused on "Do Not Sell" requests, which inform the websites that they may not share the consumer's information with third parties. [242] developed a browser extension (OptMeowt) that conveys "Do Not Sell" requests to websites through headers and cookies.

Asymmetric access tokens. The most similar work to `VICEROY` from a technical perspective is Origin Bound Certificates (OBCs) [77] (also see RFC 8471 [192]), which aims to strengthen TLS client authentication by converting cookies to asymmetric access tokens. In OBC, the client generates a unique self-signed TLS client certificate for each website, in order to remain unlinkable across websites. Although this does not authenticate the client to the website (due to the self-signed certificate), it does allow the server to ascertain whether this is the *same* client from a previous interaction. One benefit of this is that cookies can be bound to an OBC, such that, even if stolen, they cannot be used by an adversary. This is very similar to how we bind wrappers to a client-generated key. One key difference is that, in OBC, the cookies themselves are modified, whereas our use of wrappers means that `VICEROY` can be incrementally deployed on top of existing systems without needing to modify how

they use cookies. Another difference is that per-site certificates would not be suitable for accountless consumers, as these would allow servers to link together different visits from the same client. The alternative of generating per-TLS-session certificates introduces the key explosion problem, which we address in Section 6.4. Finally, OBC and `VICEROY` differ in terms of their primary objectives: OBC aims to strengthen TLS channels in general, thus is tightly coupled to the TLS protocol, whereas `VICEROY` aims to provide a specific mechanism for supporting VCRs, which can be run over any communication protocol.

Another technology related to asymmetric access tokens is the FIDO Universal Authentication Framework (UAF) [88]. UAF allows users to authenticate to servers using mechanisms other than passwords, e.g., biometrics. It also supports multi-factor authentication, e.g., requiring both a PIN and a biometric. The devices used to obtain such factors are called *authenticators*. During registration, authenticators generate and register a server-specific *authentication key*, which they then use for subsequent authentications. While similar to the approach used in `VICEROY`, there are several reasons why the FIDO UAF protocol is not directly suitable for `VICEROY`. First, `VICEROY` requires a fresh key pair to be generated for every *session* per website, since these keys will be paired with session-specific cookies. FIDO UAF does not meet this requirement, since only one key pair is generated per website. Even if the FIDO UAF protocol were modified to generate a key pair per session per website, this would lead to the key explosion problem described in Section 6.4.3, and it would be particularly challenging to store all these keys in a resource-constrained device. In contrast, in `VICEROY`, the use of BIP32 is critical to handle the significantly larger volume of keys and facilitate implementation on resource-constrained devices. Second, using FIDO UAF would require trusted devices to be online when generating cookie wrappers, since both the public and private keys are generated by the trusted device. This is not ideal for security-conscious users who might prefer to keep their trusted device offline most of the time (e.g., in a locked safe). In contrast, `VICEROY` supports both casual and security-conscious use cases by not requiring the trusted device to be present when generating public keys for cookie wrappers.

6.9 Conclusions & Future Work

Motivated by recent GDPR and CCPA regulations granting (even) accountless consumers rights to access data gathered about their behavior by web servers, we construct and evaluate `VICEROY`, a framework for authenticating accountless consumers. `VICEROY` is secure with respect to malicious clients and honest-but-curious servers, easy to deploy, and imposes fairly low overhead. Natural directions for future work include: (1) integration with client-side trusted execution environments (TEEs), (2) more extensive support for the MODIFY VCR type, and (3) support for unilateral server deletion of accountless-consumer data, which can occur if a server decides to delete consumer data without an explicit request.

Chapter 7

Final Remarks

This dissertation proposed four systems that leverage trusted hardware components and provide improved security and privacy properties that were not possible without the help of such components.

Chapter 3 described COMIT, a software-only design that eliminates the migration issue which arises when utilizing in-process TEEs in the cloud, hindering availability. The challenge of COMIT is being able to stop a TEE at any given point in time, migrate it, and allow it to resume execution at its new location, all while leaving the underlying hardware architecture untouched and preserving the security properties. We have shown that the TEE itself plays an important role in overcoming this challenge. Evaluation results demonstrate that the overhead is minimal even when migrating large TEEs.

PDoT, introduced in Chapter 4, tackles a key issue in the DNS-over-TLS architecture: DNS query privacy under malicious recursive resolvers. We showed that TEEs play a crucial role when hiding DNS queries as well as providing minimal latency. The challenges of limited TEE memory and functionalities are overcome with a unique threading model. Evaluation results display acceptable latency and throughput under certain realistic conditions.

CACTI, proposed in Chapter 5, allows clients to avoid solving CAPTCHAs while providing the same security properties and improved privacy features and client-side TEEs are necessary for CACTI when implementing such features. By combining a set of different data structures and advanced cryptographic primitives, CACTI tackles challenges such as limited TEE functionalities. Our measurements show that the end-to-end latency of CACTI is under 0.25 seconds and is capable of reducing the bandwidth by 98% compared to other conventional CAPTCHA systems.

Finally, Chapter 6 focused on VICEROY, a protocol that allows consumers without accounts to exercise their data ownership rights over their collected data while protecting their security and privacy. The key challenge of VICEROY is authenticating the consumers without identifying who they are. VICEROY overcomes this challenge by leveraging cryptographic primitives for privacy and scalability. Additionally, secure hardware tokens play a significant role when providing the highest level of security and we show that the latency is acceptable even when using this resource-limited hardware token. We also formally prove the protocol using the Tamarin prover.

We hope that the systems presented in this dissertation inspire future designs of next-generation security services that aim to take advantage of current and future trusted hardware components. Aside from improving the proposed systems, there are several natural directions for future work: (i) exploring other systems that can potentially benefit from trusted hardware components; (ii) identifying the limitation of current trusted hardware components, and designing and implementing improved components based on the findings; and (iii) deploying the proposed system(s).

Bibliography

[1] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[2] G. Acs, M. Conti, P. Gasti, C. Ghali, and G. Tsudik. Cache privacy in named-data networking. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 41–51, 2013.

[3] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner. S-faas: Trustworthy and accountable function-as-a-service using intel sgx. In *ACM Cloud Computing Security Workshop*, CCSW '19, 2019.

[4] F. Alder, A. Kurnikov, A. Paverd, and N. Asokan. Migrating SGX Enclaves with Persistent State. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

[5] F. Alliance. Fido alliance specifications overview - fido alliance. `https://fidoalliance.org/specifications/`. [Online] Accessed: 2023-01-06.

[6] F. Alliance. FIDO Authentication. `https://fidoalliance.org/how-fido-works/`. [Online] Accessed: 2023-02-08.

[7] M. Alomari, M. Cahill, A. Fekete, and U. Rohm. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *2008 IEEE 24th International Conference on Data Engineering*, pages 576–585, 2008.

[8] Amazon. AWS Availability. `https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/availability.html`. [Online] Accessed: 2022-05-18.

[9] Amazon. AWS Nitro Enclaves. `https://aws.amazon.com/ec2/nitro/nitro-enclaves/`. [Online] Accessed: 2022-05-14.

[10] Amazon. Regions and Availability Zones. `https://aws.amazon.com/about-aws/global-infrastructure/regions_az/#Availability_Zones`. [Online] Accessed: 2022-05-18.

[11] Amazon. What is AWS Nitro Enclaves? `https://docs.aws.amazon.com/enclaves/latest/user/nitro-enclave.html`. [Online] Accessed: 2023-02-12.

[12] AMD. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/. [Online] Accessed: 2022-05-14.

[13] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. [Online] Accessed: 2022-05-14.

[14] AMD. Protecting VM register state with SEV-ES. https://www.amd.com/system/files/TechDocs/Protecting%20VM%20Register%20State%20with%20SEV-ES.pdf. [Online] Accessed: 2023-02-12.

[15] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, page 7. ACM New York, NY, USA, 2013.

[16] Android Developer Documentation. Android Keystore system. https://developer.android.com/training/articles/keystore. [Online] Accessed: 2021-05-06.

[17] D. Anstee. Disappearing DNS: DoT and DoH, Where one Letter Makes a Great Difference. https://www.securitymagazine.com/articles/91674-disappearing-dns-dot-and-doh-where-one-letter-makes-a-great-difference, 2020. [Online] Accessed: 2020-05-15.

[18] AntiCAPTCHA. AntiCAPTCHA. https://anti-captcha.com/mainpage. [Online] Accessed: 2020-05-22.

[19] Apple. Storing Keys in the Secure Enclave. https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/storing_keys_in_the_secure_enclave. [Online] Accessed: 2021-05-06.

[20] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. Technical report, RFC Editor, Mar 2005.

[21] Arm. Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture. [Online] Accessed: 2022-05-14.

[22] Arm. Mbed TLS. https://github.com/ARMmbed/mbedtls. [Online] Accessed: 2020-02-14.

[23] ARM. ARM Security Technology - Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html, 2009. [Online] Accessed: 2019-05-29.

[24] ARM Holdings. ARM Security Technology, Building a Secure System using TrustZone Technology, 2009.

[25] G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik. A Practical and Provably Secure Coalition-Resistant Group Signature Scheme. In M. Bellare, editor, *Advances in Cryptology — CRYPTO 2000*, pages 255–270, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[26] M. H. Au, W. Susilo, and Y. Mu. Constant-size dynamic k-TAA. In *International conference on security and cryptography for networks*, pages 111–125. Springer, 2006.

[27] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch. LibSEAL: Revealing service integrity violations using trusted execution. In *Thirteenth EuroSys Conference*, 2018.

[28] J. Ausloos and P. Dewitte. Shattering one-way mirrors. data subject access rights in practice. *Data Subject Access Rights in Practice (January 20, 2018). International Data Privacy Law*, 8(1):4–28, 2018.

[29] D. Basin, C. Cremers, J. Dreier, S. Meier, R. Sasse, and B. Schmidt. Tamarin prover. `https://tamarin-prover.github.io/`. [Online] Accessed: 2022-10-10.

[30] B. Berger. Trusted computing group history. *Inf. Secur. Tech. Rep.*, 10(2):59–62, 2005.

[31] R. M. Best. Preventing software piracy with crypto-microprocessors. In *Proceedings of IEEE Spring COMPCON*, volume 80, pages 466–469, 1980.

[32] bitcoinjs. tiny-secp256k1. `https://www.npmjs.com/package/tiny-secp256k1`. [Online] Accessed: 2021-05-06.

[33] bitcoinjs/bip32. Bitcoin Improvement Protocol 32. `https://github.com/bitcoinjs/bip32`. [Online] Accessed: 2021-05-06.

[34] C. Boniface, I. Fouad, N. Bielova, C. Lauradoux, and C. Santos. Security analysis of subject access request procedures. In *Privacy Technologies and Policy. APF*, 2019.

[35] S. Bortzmeyer. DNS Query Name Minimisation to Improve Privacy. Technical report, RFC Editor, Mar 2016.

[36] S. Bortzmeyer. Encryption and authentication of the DNS resolver-to-authoritative communication. `https://tools.ietf.org/html/draft-bortzmeyer-dprive-resolver-to-auth-01`, 2018. [Online] Accessed: 2019-05-29.

[37] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. 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, page 788–800, New York, NY, USA, 2019. Association for Computing Machinery.

[38] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software grand exposure: SGX cache attacks are practical. In W. Enck and C. Mulliner, editors, *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. USENIX Association, 2017.

[39] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, 2004.

[40] E. Brickell, L. Chen, and J. Li. A static diffie-hellman attack on several direct anonymous attestation schemes. In *International Conference on Trusted Systems*, pages 95–111. Springer, 2012.

[41] E. Brickell and J. Li. Enhanced Privacy ID: A Direct Anonymous Attestation Scheme with Enhanced Revocation Capabilities. In *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, WPES '07, page 21–30, New York, NY, USA, 2007. Association for Computing Machinery.

[42] J. Brodkin. Firefox turns encrypted DNS on by default to thwart snooping ISPs. `https://arstechnica.com/information-technology/2020/02/firefox-turns-encrypted-dns-on-by-default-to-thwart-snooping-isps/`, 2020. [Online] Accessed: 2020-05-15.

[43] Browserify. Browserify. `https://github.com/browserify/browserify`. [Online] Accessed: 2021-05-06.

[44] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, Aug. 2018. USENIX Association.

[45] E. Bursztein, S. Bethard, C. Fabry, J. C. Mitchell, and D. Jurafsky. How good are humans at solving CAPTCHAs? A large scale evaluation. In *2010 IEEE symposium on security and privacy*, pages 399–413. IEEE, 2010.

[46] M. Cagnazzo, T. Holz, and N. Pohlmann. GDPiRated - stealing personal information on- and offline. In *European Symposium on Research in Computer Security*, 2019.

[47] California Attorney General. California Consumer Privacy Act Regulations. `https://oag.ca.gov/sites/all/files/agweb/pdfs/privacy/oal-sub-final-text-of-regs.pdf`, 2020. [Online] Accessed: 2021-05-06.

[48] California Legislature. California Consumer Privacy Act of 2018 (as amended by the California Privacy Rights Act of 2020). `https://www.oag.ca.gov/privacy/ccpa`, 2020. [Online] Accessed: 2021-05-06.

[49] S. Cannady and M. Wiseman. Using the TPM to Solve Today's Most Urgent Cybersecurity Problems. https://trustedcomputinggroup.org/wp-content/uploads/17.pdf, May 2014. [Online] Accessed: 2023-01-06.

[50] C. Castelluccia, L. Olejnik, and T. Minh-Dung. Selling off privacy at auction. In *Network and Distributed System Security Symposium*, 2014.

[51] S. Castillo-Perez and J. García-Alfaro. Anonymous resolution of DNS queries. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems: OTM 2008, OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, November 9-14, 2008, Proceedings, Part II*, volume 5332 of *Lecture Notes in Computer Science*, pages 987–1000. Springer, 2008.

[52] V. Cerf. Guidelines for Internet Measurement Activities. Technical report, RFC Editor, Oct 1991.

[53] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. F. Oswald, and F. D. Garcia. Voltpillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In M. Bailey and R. Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 699–716. USENIX Association, 2021.

[54] C.-M. Cheng, H. Kung, and K.-S. Tan. Use of spectral analysis in defense against DoS attacks. In *Global Telecommunications Conference, 2002. GLOBECOM'02. IEEE*, volume 3, pages 2143–2148. IEEE, 2002.

[55] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *IEEE 36th Annual Foundations of Computer Science*, 1995.

[56] Chrome Developer Documentation. Native Messaging. https://developer.chrome.com/docs/apps/nativeMessaging/. [Online] Accessed: 2021-05-06.

[57] Cloudflare. 1.1.1.1 Resolver Examination Report. https://www.cloudflare.com/compliance/. [Online] Accessed: 2020-05-14.

[58] Cloudflare. Announcing 1.1.1.1: the fastest, privacy-first consumer DNS service. https://blog.cloudflare.com/announcing-1111/. [Online] Accessed: 2020-05-14.

[59] Cloudflare. Announcing the Results of the 1.1.1.1 Public DNS Resolver Privacy Examination. https://blog.cloudflare.com/announcing-the-results-of-the-1-1-1-1-public-dns-resolver-privacy-examination/. [Online] Accessed: 2020-05-14.

[60] Cloudflare. Cloudflare Rate Limiting. https://www.cloudflare.com/rate-limiting/. [Online] Accessed: 2020-05-19.

[61] Cloudflare. DNS over TLS - Cloudflare Resolver. https://1.1.1.1/dns/. [Online] Accessed: 2019-05-29.

[62] Cloudflare. Moving from reCAPTCHA to hCaptcha. https://blog.cloudflare.com/moving-from-recaptcha-to-hcaptcha/. [Online] Accessed: 2020-05-19.

[63] Cloudflare. Using Privacy Pass with Cloudflare. https://support.cloudflare.com/hc/en-us/articles/115001992652-Using-Privacy-Pass-with-Cloudflare. [Online] Accessed: 2020-06-01.

[64] M. Costa, L. Esswood, O. Ohrimenko, F. Schuster, and S. Wagh. The Pyramid Scheme: Oblivious RAM for Trusted Processors. *CoRR*, abs/1712.07882, 2017.

[65] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 857–874. USENIX Association, 2016.

[66] K. Crichton, N. Christin, and L. F. Cranor. How do home computer users browse the web? *ACM Trans. Web*, 16(1), sep 2021.

[67] cs.nic. Knot Resolver. https://www.knot-resolver.cz/. [Online] Accessed: 2019-05-29.

[68] A. Dabrowski, G. Merzdovnik, J. Ullrich, G. Sendera, and E. Weippl. Measuring cookies and web privacy in a post-GDPR world. In *International Conference on Passive and Active Network Measurement*, 2019.

[69] F. Dall, G. D. Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom. Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):171–191, 2018.

[70] Daniel Bernstein. Introduction to DNSCurve. https://dnscurve.org/index.html, 2009. [Online] Accessed: 2019-05-29.

[71] J. Danisevskis. Android Protected Confirmation: Taking transaction security to the next level. https://developer.android.com/training/articles/security-android-protected-confirmation. [Online] Accessed: 2020-02-05.

[72] R. Datta, J. Li, and J. Z. Wang. IMAGINATION: a robust image-based CAPTCHA generation system. In *Proceedings of the 13th annual ACM international conference on Multimedia*, pages 331–334, 2005.

[73] A. Davidson, I. Goldberg, N. Sullivan, G. Tankersley, and F. Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proceedings on Privacy Enhancing Technologies*, 2018(3):164–180, 2018.

[74] dchest. Package captcha. https://github.com/dchest/captcha. [Online] Accessed: 2020-05-21.

[75] M. Di Martino, P. Robyns, W. Weyts, P. Quax, W. Lamotte, and K. Andries. Personal information leakage by abusing the GDPR 'Right of Access'. In *Fifteenth Symposium on Usable Privacy and Security*, 2019.

[76] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. Technical report, RFC Editor, Aug 2008.

[77] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 317–331, 2012.

[78] X. Ding and G. Tsudik. Initializing trust in smart devices via presence attestation. *Computer Communications*, 131:35 – 38, 2018.

[79] H. Duan, C. Wang, X. Yuan, Y. Zhou, Q. Wang, and K. Ren. LightBox: Full-stack Protected Stateful Middlebox at Lightning Speed. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2351–2367. ACM, 2019.

[80] D. Eastlake. Domain Name System (DNS) IANA Considerations. Technical report, RFC Editor, Apr 2013.

[81] A. Edmundson, P. Schmitt, and N. Feamster. ODNS: Oblivious DNS. `https://odns.cs.princeton.edu/`, 2018. [Online] Accessed: 2019-05-29.

[82] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. Garcia, E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh. Fidelius: Protecting user secrets from compromised browsers. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 264–280, 2019.

[83] European Parliament and Council. General Data Protection Regulation, Regulation (EU) 2016/679 (as amended). `https://eur-lex.europa.eu/eli/reg/2016/679/2016-05-04`, 2016. [Online] Accessed: 2021-05-06.

[84] H. Federrath, K. Fuchs, D. Herrmann, and C. Piosecny. Privacy-preserving DNS: analysis of broadcast, range queries and mix-based protection methods. In V. Atluri and C. Díaz, editors, *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, volume 6879 of *Lecture Notes in Computer Science*, pages 665–683. Springer, 2011.

[85] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz. Measuring HTTPS adoption on the web. In *26th USENIX Security Symposium*, 2017.

[86] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen. Scalable memory protection in the PENGLAI enclave. In A. D. Brown and J. R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 275–294. USENIX Association, 2021.

[87] C. A. Fidas, A. G. Voyiatzis, and N. M. Avouris. On the necessity of user-friendly CAPTCHA. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2623–2626, 2011.

[88] FIDO. FIDO UAF. https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-overview-v1.2-ps-20201020.html. [Online] Accessed: 2021-12-10.

[89] H. Gao, W. Wang, and Y. Fan. Divide and conquer: an efficient attack on Yahoo! CAPTCHA. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 9–16. IEEE, 2012.

[90] O. Goldreich. Towards a theory of software protection. In A. M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 426–439. Springer, 1986.

[91] P. Golle. Machine learning attacks against the Asirra CAPTCHA. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 535–542, 2008.

[92] D. Goltzsche, S. Rusch, M. Nieke, S. Vaucher, N. Weichbrodt, V. Schiavoni, P.-L. Aublin, P. Cosa, C. Fetzer, P. Felber, P. Pietzuch, and R. Kapitza. EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '18, pages 386–397. IEEE, Jun 2018.

[93] R. Gonzalez, L. Jiang, M. Ahmed, M. Marciel, R. Cuevas, H. Metwalley, and S. Niccolini. The cookie recipe: Untangling the use of cookies in the wild. In *Network Traffic Measurement and Analysis Conference*, 2017.

[94] Google. Chrome Native Messaging Protocol. https://developer.chrome.com/extensions/nativeMessaging#native-messaging-host-protocol. [Online] Accessed: 2020-02-09.

[95] Google. Chrome Notifications. https://developer.chrome.com/apps/notifications. [Online] Accessed: 2020-02-14.

[96] Google. Designing resilient systems. https://cloud.google.com/compute/docs/tutorials/robustsystems. [Online] Accessed: 2022-05-18.

[97] Google. Designing resilient systems. https://cloud.google.com/compute/docs/tutorials/robustsystems#distribute. [Online] Accessed: 2022-05-18.

[98] Google. Google Analytics. https://developers.google.com/analytics/devguides/collection/analyticsjs/cookie-usage. [Online] Accessed: 2021-05-06.

[99] Google. Google Chrome. https://www.google.com/chrome/. [Online] Accessed: 2020-02-11.

[100] Google. Google Chrome: runtime.Port. https://developer.chrome.com/extensions/runtime#type-Port. [Online] Accessed: 2020-02-12.

[101] Google. Google Cloud confidential computing. https://cloud.google.com/compute/confidential-vm/docs/about-cvm. [Online] Accessed: 2022-05-14.

[102] Google. Native Messaging. https://developer.chrome.com/extensions/nativeMessaging. [Online] Accessed: 2020-02-13.

[103] Google. reCAPTCHA. https://www.google.com/recaptcha/intro/v3.html. [Online] Accessed: 2020-02-05.

[104] Google. reCAPTCHA v2. https://developers.google.com/recaptcha/docs/display. [Online] Accessed: 2020-02-13.

[105] Google. DNS over TLS support in Android P Developer Preview. https://security.googleblog.com/2018/04/dns-over-tls-support-in-android-p.html, 2018. [Online] Accessed: 2019-05-29.

[106] Google Developer Documentation. Cookie Matching. https://developers.google.com/authorized-buyers/rtb/cookie-guide. [Online] Accessed: 2021-05-06.

[107] Google Transparency Report. HTTPS encryption on the web. https://transparencyreport.google.com/https/overview. [Online] Accessed: 2022-10-10.

[108] R. Gossweiler, M. Kamvar, and S. Baluja. What's up CAPTCHA? A CAPTCHA based on image orientation. In *Proceedings of the 18th international conference on World wide web*, pages 841–850, 2009.

[109] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel SGX. In C. Giuffrida and A. Stavrou, editors, *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*, pages 2:1–2:6. ACM, 2017.

[110] T. C. Group. Trusted Computing Platform Alliance (TCPA) Main Specification Version 1.1b. https://trustedcomputinggroup.org/wp-content/uploads/TCPA_Main_TCG_Architecture_v1_1b.pdf, 2003. [Online] Accessed: 2023-02-08.

[111] T. C. Group. TPM 1.2 Main Specification. https://trustedcomputinggroup.org/resource/tpm-main-specification/, 2009. [Online] Accessed: 2023-02-08.

[112] T. C. Group. Trusted Platform Module Library Part 1: Architecture. https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf, November 2019. [Online] Accessed: 2023-01-06.

[113] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li. Secure Live Migration of SGX Enclaves on Untrusted Cloud. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[114] J. Guerreiro, R. Moura, and J. N. Silva. TEEnder: SGX enclave migration using HSMs. *Comput. Secur.*, 96:101874, 2020.

[115] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-Bot: Improving Service Availability in the Face of Botnet Attacks. In *NSDI*, volume 9, pages 307–320, 2009.

[116] D. Herrmann and J. Lindemann. Obtaining personal data and asking for erasure: Do app vendors and website owners honour your privacy rights? *arXiv preprint arXiv:1602.01804*, 2016.

[117] A. Herzberg and S. S. Pinter. Public protection of software. In H. C. Williams, editor, *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 158–179. Springer, 1985.

[118] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11(10.1145):2487726–2488370, 2013.

[119] P. Hoffman. DNS Queries over HTTPS (DoH). Technical report, RFC Editor, Oct 2018.

[120] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman. Specification for DNS over Transport Layer Security (TLS), 2016.

[121] Intel. EPID SDK. `https://github.com/Intel-EPID-SDK/epid-sdk`. [Online] Accessed: 2020-02-14.

[122] Intel. Intel Dynamic Application Loader Developer Guide: Monotonic Counters. `https://software.intel.com/en-us/dal-developer-guide-features-monotonic-counters`. [Online] Accessed: 2020-02-05.

[123] Intel. Intel Integrated Performance Primitives Cryptography. `https://github.com/intel/ipp-crypto`. [Online] Accessed: 2020-05-28.

[124] Intel. Intel NUC Kit NUC7PJYH. `https://ark.intel.com/content/www/us/en/ark/products/126137/intel-nuc-kit-nuc7pjyh.html`. [Online] Accessed: 2020-02-11.

[125] Intel. Intel Pentium Processor G4400. `https://ark.intel.com/content/www/us/en/ark/products/88179/intel-pentium-processor-g4400-3m-cache-3-30-ghz.html`. [Online] Accessed: 2020-05-19.

[126] Intel. Intel Trust Domain Extensions. `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html`. [Online] Accessed: 2022-05-14.

[127] Intel. SGX EPID Attestation. `https://api.portal.trustedservices.intel.com/EPID-attestation`. [Online] Accessed: 2021-05-06.

[128] Intel. Software Guard Extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html. [Online] Accessed: 2021-05-06.

[129] Intuition Machines, Inc. hCaptcha. https://www.hcaptcha.com/. [Online] Accessed: 2020-05-21.

[130] ismailakkila/bip32. Bitcoin Improvement Protocol 32. https://github.com/ismailakkila/bip32. [Online] Accessed: 2021-05-06.

[131] J. Iyengar and M. Thomson. QUIC: A UDP-Based multiplexed and secure transport. RFC 9000, May 2021.

[132] P. Jain, S. J. Desai, M.-W. Shih, T. Kim, S. M. Kim, J.-H. Lee, C. Choi, Y. Shin, B. B. Kang, and D. Han. OpenSGX: An Open Platform for SGX Research. In *NDSS*, volume 16, pages 21–24, 2016.

[133] T. Jensen, I. Pashov, and G. Montenegro. Windows will improve user privacy with DNS over HTTPS. https://techcommunity.microsoft.com/t5/networking-blog/windows-will-improve-user-privacy-with-dns-over-https/ba-p/1014229, 2019. [Online] Accessed: 2020-05-15.

[134] S. Jordan. Strengths and weaknesses of notice and consent requirements under the GDPR, the CCPA/CPRA, and the FCC Broadband Privacy Order. *Cardozo Arts & Entertainment Law Journal, Forthcoming*, 2021.

[135] S. Jordan, Y. Nakatsuka, E. Ozturk, A. Paverd, and G. Tsudik. VICEROY: GDPR-/CCPA-compliant Enforcement of Verifiable Accountless Consumer Requests. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27- March 3, 2023*. The Internet Society, 2023.

[136] S. Jordan, S. Narasimhan, and J. Hong. Deficiencies in the disclosures of privacy policy and in user choice. *Loyola consumer law review, Forthcoming*, 2022.

[137] S. T. Kent. *Protecting externally supplied software in small computers*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1980.

[138] C. A. Kerrache, N. Lagraa, C. T. Calafate, and A. Lakas. TFDD: A trust-based framework for reliable data delivery and DoS defense in VANETs. *Vehicular Communications*, 9:254–267, 2017.

[139] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij. Integrating remote attestation with transport layer security. *CoRR*, abs/1801.05863, 2018.

[140] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[141] J. L. Kröger, J. Lindemann, and D. Herrmann. How do app vendors respond to subject access requests?: a longitudinal privacy study on iOS and Android Apps. In M. Volkamer and C. Wressnegger, editors, *ARES 2020: The 15th International Conference on Availability, Reliability and Security, Virtual Event, Ireland, August 25-28, 2020*, pages 10:1–10:10. ACM, 2020.

[142] S. Lab. PDoT Source Code. `https://github.com/sprout-uci/PDoT`, 2019.

[143] N. Labs. Stubby. `https://dnsprivacy.org/wiki/display/DP/DNS+Privacy+Daemon+-+Stubby`. [Online] Accessed: 2019-05-29.

[144] N. Labs. Unbound. `https://nlnetlabs.nl/projects/unbound/about/`. [Online] Accessed: 2019-05-29.

[145] Ledger SAS. Ledger Hardware Wallets. `https://www.ledger.com/`. [Online] Accessed: 2021-05-06.

[146] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[147] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, Vancouver, BC, Aug. 2017. USENIX Association.

[148] A. Leung, L. Chen, and C. J. Mitchell. On a possible privacy flaw in direct anonymous attestation (DAA). In *International Conference on Trusted Computing*, pages 179–190. Springer, 2008.

[149] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. 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*, pages 28–40, 2018.

[150] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, Aug. 2018. USENIX Association.

[151] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, May 2015.

[152] X. Liu, X. Yang, and Y. Lu. To filter or to authorize: Network-layer DoS defense against multimillion-node botnets. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 195–206, 2008.

[153] C. Lu, B. Liu, Z. Li, S. Hao, H. Duan, M. Zhang, C. Leng, Y. Liu, Z. Zhang, and J. Wu. An end-to-end, large-scale measurement of DNS-over-encryption: How far have we come? In *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, pages 22–35, New York, NY, USA, oct 2019. Association for Computing Machinery.

[154] Y. Lu and G. Tsudik. Towards Plugging Privacy Leaks in the Domain Name System. In *2010 IEEE Tenth International Conference on Peer-to-Peer Computing*, P2P, pages 1–10. IEEE, Aug 2010.

[155] Majestic. Majestic Million. `https://blog.majestic.com/development/majestic-million-csv-daily/`, 2012. [Online] Accessed: 2018-08-08.

[156] S. Matala, T. Nyman, and N. Asokan. Historical insight into the development of Mobile TEEs. `https://blog.ssg.aalto.fi/2019/06/historical-insight-into-development-of.html`, June 2019. [Online] Accessed: 2023-02-08.

[157] J. R. Mayer and J. C. Mitchell. Third-Party Web Tracking: Policy and Technology. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 413–427. IEEE Computer Society, 2012.

[158] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[159] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 1–1, New York, New York, USA, 2013. ACM Press.

[160] S. Meier, B. Schmidt, C. Cremers, and D. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification*, 2013.

[161] Microsoft. Azure availability zones. `https://docs.microsoft.com/en-us/azure/availability-zones/az-overview`. [Online] Accessed: 2022-05-18.

[162] Microsoft. Azure confidential computing. `https://azure.microsoft.com/en-us/solutions/confidential-compute`. [Online] Accessed: 2022-05-14.

[163] Microsoft. Microsoft Azure Attestation. `https://learn.microsoft.com/en-us/azure/attestation/overview`. [Online] Accessed: 2023-02-09.

[164] Microsoft. Open Enclave SDK. `https://openenclave.io/sdk/`. [Online] Accessed: 2021-05-19.

[165] Microsoft. Regions for virtual machines in Azure. https://docs.microsoft.com/en-us/azure/virtual-machines/regions#feature-availability. [Online] Accessed: 2022-05-18.

[166] Microsoft. Introducing Azure confidential computing. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/, 2017. [Online] Accessed: 2019-05-29.

[167] Microsoft. BitLocker. https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview, December 2022. [Online] Accessed: 2023-01-06.

[168] MITRE ATT&CK Techniques. Steal web session cookie. https://attack.mitre.org/techniques/T1539/. [Online] Accessed: 2021-05-06.

[169] P. Mockapetris. Domain names - implementation and specification. Technical report, RFC Editor, Nov 1987.

[170] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In W. Fischer and N. Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 69–90. Springer, 2017.

[171] G. Mori and J. Malik. Recognizing objects in adversarial clutter: Breaking a visual CAPTCHA. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 1, pages I–I. IEEE, 2003.

[172] R. Mori and S. Tashiro. The concept of software service system (SSS). *Systems and Computers in Japan*, 19(5):38–49, 1988.

[173] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage. Re: CAPTCHAs-Understanding CAPTCHA-Solving Services in an Economic Context. In *USENIX Security Symposium*, volume 10, page 3, 2010.

[174] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy*, pages 1466–1482, 2020.

[175] Y. Nakatsuka, E. Ozturk, A. Paverd, and G. Tsudik. CACTI: Captcha Avoidance via Client-side TEE Integration. In M. Bailey and R. Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2561–2578. USENIX Association, 2021.

[176] Y. Nakatsuka, E. Ozturk, A. Paverd, and G. Tsudik. CACTI: Captcha avoidance via client-side TEE integration. In *30th USENIX Security Symposium*, 2021.

[177] Y. Nakatsuka, A. Paverd, and G. Tsudik. PDoT: Private DNS-over-TLS with TEE Support. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 489–499, New York, NY, USA, 2019. Association for Computing Machinery.

[178] Y. Nakatsuka, A. Paverd, and G. Tsudik. PDoT: Private DNS-over-TLS with TEE Support. *Digital Threats*, 2(1), feb 2021.

[179] Node.js. Node JS. `https://nodejs.org/en/`. [Online] Accessed: 2021-05-06.

[180] NovaCrypto/BIP32. Bitcoin Improvement Protocol 32. `https://github.com/NovaCrypto/BIP32`. [Online] Accessed: 2021-05-06.

[181] NPM UUID. NPM UUID. `https://www.npmjs.com/package/uuid`. [Online] Accessed: 2021-05-06.

[182] OpenVZ. CRIU – A project to implement checkpoint/restore functionality for Linux. `https://criu.org/`. [Online] Accessed: 2021-05-12.

[183] J. D. Osborn and D. C. Challener. Trusted platform module evolution. *Johns Hopkins APL Technical Digest (Applied Physics Laboratory)*, 32(2):536–543, 2013.

[184] X. Ouyang, B. Tian, Q. Li, J.-y. Zhang, Z.-M. Hu, and Y. Xin. A novel framework of defense system against DoS attacks in wireless sensor networks. In *2011 7th International Conference on Wireless Communications, Networking and Mobile Computing*, pages 1–5. IEEE, 2011.

[185] J. Park, S. Park, B. B. Kang, and K. Kim. eMotion: An SGX extension for migrating enclaves. *Computers & Security*, 80:173–185, 2019.

[186] J. Park, S. Park, J. Oh, and J.-J. Won. Toward Live Migration of SGX-Enabled Virtual Machines. In *2016 IEEE World Congress on Services (SERVICES)*, 2016.

[187] Paul Calvano. An Analysis of Cookie Sizes on the Web. `https://paulcalvano.com/2020-07-13-an-analysis-of-cookie-sizes-on-the-web/`. [Online] Accessed: 2021-12-10.

[188] A. Paverd, A. Martin, and I. Brown. Modelling and automatically analysing privacy properties for honest-but-curious adversaries. *Tech. Rep*, 2014.

[189] J. Pavur and C. Knerr. GDPArrrrr: Using Privacy Laws to Steal Identities. *CoRR*, abs/1912.00731, 2019.

[190] T. Peng, C. Leckie, and K. Ramamohanarao. Survey of network-based defense mechanisms countering the DoS and DDoS problems. *ACM Computing Surveys (CSUR)*, 39(1):3–es, 2007.

[191] P. Perlegos. *DoS defense in structured peer-to-peer networks*. Computer Science Division, University of California, 2004.

[192] A. Popov, M. Nystroem, D. Balfanz, and J. Hodges. The Token Binding Protocol Version 1.0. RFC 8471, Oct. 2018.

[193] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A Secure Database Using SGX. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 264–278. IEEE Computer Society, 2018.

[194] produck. svg captcha. `https://github.com/produck/svg-captcha`. [Online] Accessed: 2020-05-21.

[195] D. Project. DNSCrypt. `https://dnscrypt.info/`. [Online] Accessed: 2019-05-29.

[196] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. Zanella-Beguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. Cryptology ePrint Archive, Report 2019/757, 2019.

[197] Prowebscraper. Top 10 Captcha Solving Services Compared. `https://prowebscraper.com/blog/top-10-captcha-solving-services-compared/`. [Online] Accessed: 2020-05-22.

[198] G. B. Purdy, G. J. Simmons, and J. Studier. A software protection scheme. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 99–103. IEEE Computer Society, 1982.

[199] J. Randmets. An Overview of Vulnerabilities and Mitigations of Intel SGX Applications. `https://cyber.ee/research/reports/D-2-116-An-Overview-of-Vulnerabilities-and-Mitigations-of-Intel-SGX-Applications.pdf`, 2021. [Online] Accessed: 2023-01-05.

[200] E. Rescorla, H. Tschofenig, and N. Modadugu. The datagram transport layer security (DTLS) protocol version 1.3. RFC 9147, Apr. 2022.

[201] S. A. Rotondo. Trusted computing platform alliance. In H. C. A. van Tilborg and S. Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Ed*, page 1332. Springer, 2011.

[202] C. Rudolph. Covert Identity Information in Direct Anonymous Attestation (DAA). In H. S. Venter, M. M. Eloff, L. Labuschagne, J. H. P. Eloff, and R. von Solms, editors, *New Approaches for Security, Privacy and Trust in Complex Environments, Proceedings of the IFIP TC-11 22nd International Information Security Conference (SEC 2007), 14-16 May 2007, Sandton, South Africa*, volume 232 of *IFIP*, pages 443–448. Springer, 2007.

[203] sammyne/bip32. Bitcoin Improvement Protocol 32. `https://github.com/sammyne/bip32`. [Online] Accessed: 2021-05-06.

[204] M. Sanghavi and S. Doshi. Progressive captcha, Apr. 30 2009. US Patent App. 11/929,716.

[205] S. Sasy, S. Gorbunov, and C. W. Fletcher. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[206] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 753–768, New York, NY, USA, 2019. Association for Computing Machinery.

[207] M. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: eradicating controlled-channel attacks against enclave programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[208] H. Shulman and Haya. Pretty Bad Privacy: Pitfalls of DNS Encryption. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, WPES '14, pages 191–200, New York, New York, USA, 2014. ACM Press.

[209] G. J. Simmons. How to (selectively) broadcast A secret. In *1985 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 22-24, 1985*, pages 108–115. IEEE Computer Society, 1985.

[210] simov. native-messaging - npm. `https://www.npmjs.com/package/native-messaging`. [Online] Accessed: 2021-05-06.

[211] SoloKeys. SoloKeys. `https://solokeys.com/`. [Online] Accessed: 2021-12-10.

[212] C. Soriente, G. Karame, W. Li, and S. Fedorov. ReplicaTEE: Enabling Seamless Replication of SGX Enclaves in the Cloud. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 158–171, 2019.

[213] Sprout. VICEROY Source Code. `https://github.com/sprout-uci/VICEROY`. [Online] Accessed: 2021-05-06.

[214] StrongLoop. Express - Node.js web application framework. `https://expressjs.com/`. [Online] Accessed: 2021-05-06.

[215] Y. Swami. Intel SGX Remote Attestation is not sufficient. `https://www.blackhat.com/docs/us-17/thursday/us-17-Swami-SGX-Remote-Attestation-Is-Not-Sufficient-wp.pdf`, 2017. [Online] Accessed: 2023-03-04.

[216] S. Tamrakar, J. Liu, A. Paverd, J.-E. Ekberg, B. Pinkas, and N. Asokan. The circle game: Scalable private membership test using trusted hardware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, 2017.

[217] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 1–14, New York, New York, USA, 2018. ACM Press.

[218] Trezor. Trezor Firmware. `https://github.com/trezor/trezor-firmware/`. [Online] Accessed: 2021-12-10.

[219] T. Urban, M. Degeling, T. Holz, and N. Pohlmann. "Your hashed IP address: Ubuntu." perspectives on transparency tools for online advertising. In *35th Annual Computer Security Applications Conference*, 2019.

[220] T. Urban, D. Tatang, M. Degeling, T. Holz, and N. Pohlmann. A study on subject data access in online advertising after the GDPR. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, 2019.

[221] S. van Schaik, A. Kwong, and D. Genkin. SGAxe: How SGX Fails in Practice. `https://sgaxe.com/files/SGAxe.pdf`, 2020. [Online] Accessed: 2023-03-04.

[222] M. Veeningen, B. d. Weger, and N. Zannone. Modeling identity-related properties and their privacy strength. In *International Workshop on Formal Aspects in Security and Trust*, 2010.

[223] M. Veeningen, B. d. Weger, and N. Zannone. Formal privacy analysis of communication protocols for identity management. In *International Conference on Information Systems Security*, 2011.

[224] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. In E. Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 294–311, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[225] L. Von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *International Conference on the Theory and Applications of Cryptographic Techniques*, 2003.

[226] J. Z. Wang, R. Datta, and J. Li. Image-based CAPTCHA generation system, Apr. 19 2011. US Patent 7,929,805.

[227] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2421–2434, New York, NY, USA, 2017. Association for Computing Machinery.

[228] S. Weiser and M. 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*, pages 261–268, 2017.

[229] S. R. White and L. Comerford. ABYSS: A trusted architecture for software protection. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 27-29, 1987*, pages 38–51. IEEE Computer Society, 1987.

[230] S. R. White, S. H. Weingart, W. C. Arnold, and E. R. Palmer. Introduction to the citadel architecture: Security in physically exposed environments. Technical report, Technical Report RC16672, Distributed security systems group, IBM Thomas J ..., 1991.

[231] P. Wuille. Hierarchical Deterministic Wallets (BIP 0032). `https://github.com/bitcoin/bips/blob/master/bip-0033.mediawiki`. [Online] Accessed: 2021-05-06.

[232] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, May 2015.

[233] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[234] J. Yan and A. S. El Ahmad. A Low-cost Attack on a Microsoft CAPTCHA. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 543–554, 2008.

[235] Yubico. Yubikey. `https://www.yubico.com/`. [Online] Accessed: 2023-01-06.

[236] S. Zaitsev. JSMN JSON Parser. `https://github.com/zserge/jsmn`. [Online] Accessed: 2020-02-13.

[237] Z. Zhang, X. Ding, G. Tsudik, J. Cui, and Z. Li. Presence Attestation: The Missing Link in Dynamic Trust Bootstrapping. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 89–102, New York, NY, USA, 2017. Association for Computing Machinery.

[238] Z. Zhang, X. Ding, G. Tsudik, J. Cui, and Z. Li. Presence attestation: The missing link in dynamic trust bootstrapping. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[239] F. Zhao, Y. Hori, and K. Sakurai. Analysis of Privacy Disclosure in DNS Query. In *2007 International Conference on Multimedia and Ubiquitous Engineering*, MUE '07, pages 952–957. IEEE, 2007.

[240] F. Zhao, Y. Hori, and K. Sakurai. Two-Servers PIR Based DNS Query Scheme with Privacy-Preserving. In *The 2007 International Conference on Intelligent Pervasive Computing*, IPC '07, pages 299–302. IEEE, Oct 2007.

[241] L. Zhu, Z. Hu, J. Heidemann, D. Wessels, A. Mankin, and N. Somaiya. Connection-Oriented DNS to Improve Privacy and Security. In *2015 IEEE Symposium on Security and Privacy*, pages 171–186. IEEE, May 2015.

[242] S. Zimmeck and K. Alicki. Standardizing and implementing do not sell. In J. Ligatti, X. Ou, W. Lueks, and P. Syverson, editors, *WPES'20: Proceedings of the 19th Workshop on Privacy in the Electronic Society, Virtual Event, USA, November 9, 2020*, pages 15–20. ACM, 2020.

Appendix A

VICEROY Tamarin model

A.1 Formal Protocol Specification

Listing A.1 presents the formal model of the VICEROY protocol and the security properties verified using the Tamarin prover [160]. For details of the Tamarin syntax and conventions, please refer to the Tamarin prover website [29].

```
theory VICEROY
begin

builtins: signing, hashing

// Public key infrastructure
rule Register_pk:
  [ Fr(~skA) ]
  --[Unique($A)]->
  [ !Sk($A, ~skA), !Pk($A, pk(~skA)), Out(pk(~skA)) ]

rule Reveal_sk:
  [ !Sk(A, skA) ] --[ RevSk(A) ]-> [ Out(skA) ]
```

```
14
15
16 // Derivable asymmetric keys
17 rule Register_derived_key:
18   [ !Sk($A, skA), Fr(~dskA) ]
19   -->
20   [ !Dsk($A, ~dskA), !Dpk($A, pk(~dskA)), Out(pk(~dskA)) ]
21
22 rule Reveal_dsk:
23   [ !Dsk(A, dskA) ] --[ RevDsk(A) ]-> [ Out(dskA) ]
24
25
26 /* We formalize the following protocol
27
28 Trusted device (T)  |  Client device (C)  |  Web server (S)
29
30 [Optional] Trusted device generates master private key sk(t) and sends device
       public key pk(t/i) to the client's device [not included in the model since
        it is not mandatory to use a separate trusted device].
31
32 When the client begins interacting with a website, the website issues the
       client with a cookie.
33 0. S -> C: cookie
34
35 The client derives a fresh VCR public key pk(t/i/j) from the device public key
        and sends it to the server along with the cookie.
36 1. C -> S: cookie, pk(t/i/j)
37
38 The server returns a cookie wrapper, which is a signature over the cookie and
       the provided VCR public key.
39 2. S -> C: sign_{sk(S)}{h(cookie, pk(t/i/j))}
40
41 When issuing a VCR, the user signs the request and cookie using the
```

```
    corresponding private key on their trusted device [not included in this
    model], and then sends the cookie, request, public key, wrapper, and
    signature to the server.
3. C -> S: cookie, request, pk(t/i), sign_{sk(S)}{h(cookie, pk(t/i/j))}, sign_
    {sk(t/i/j)}{h(request, cookie)}

If all the signatures can be verified, the server accepts the VCR and performs
     the requested operation. */

rule S_0:
    [ Fr(~cookie) ]
  --[ ]->
    [ Out( ~cookie ), !State_S_0($S, ~cookie) ]

rule C_1:
  let dpkC = pk(~dskC)
      m1 = <cookie, dpkC>
  in
    [ In(cookie), Fr(~dskC) ]
  --[ Request_wrapper( dpkC ) ]->
    [ Out( m1 ), State_C_1(~dskC, dpkC, cookie) ]

rule S_1:
  let m1 = <cookie, dpkC>
      m2 = sign(h(<cookie, dpkC>), skS)
  in
    [ !State_S_0($S, cookie), !Pk($S, pkS), !Sk($S, skS), In( m1 ) ]
  --[ Issue_wrapper($S, dpkC, <cookie>) ]->
    [ Out( m2 ), !State_S_1($S, pkS) ]

rule C_2:
  let request = <'op', ~nonce, pkS>
      m3 = <cookie, request, dpkC, wrapper, sign( h(<request, cookie>), dskC )
```

```
    >
  in
    [ State_C_1(dskC, dpkC, cookie), !Pk($S, pkS), In(wrapper), Fr(~nonce) ]
  --[ Eq( verify(wrapper, h(<cookie, dpkC>), pkS), true )
    , Issue_VCR(dskC, $S, <cookie, request>) ]->
    [ Out( m3 ) ]

rule S_2:
  let request = <'op', nonce, pkS>
      m3 = <cookie, request, dpkC, wrapper, client_sig>
  in
    [ !State_S_1($S, pkS), In( m3 ) ]
  --[ Eq( verify(wrapper, h(<cookie, dpkC>), pkS), true )
    , Eq( verify(client_sig, h(<request, cookie>), dpkC), true )
    , Unique(nonce)
    , Accept_VCR($S, dpkC, <cookie, request>) ]->
    [  ]

restriction Equality:
  "All x y #i. Eq(x,y) @i ==> x = y"

restriction Uniqueness:
   "All x #i #j. Unique(x) @ i & Unique(x) @ j ==> #i = #j"

/* Wrapper unforgeability: whenever the server accepts a VCR from a client,
    then that server had previously issued a cookie wrapper to that client for
    the same cookie, or the adversary performed a long-term key reveal on the
    server, or the adversary knows the client's derived private key. */
lemma wrapper_unforgeability:
  " All server dskC cookie request #i.
       Accept_VCR(server, pk(dskC), <cookie, request>) @ i
     ==>
         (Ex #j. Issue_wrapper(server, pk(dskC), <cookie>) @ j & j < i)
```

```
99          | (Ex #r. RevSk(server) @ r) | (Ex #r. KU(dskC) @ r) "
100
101 /* VCR unforgeability: whenever the server accepts a VCR, then the client with
        the corresponding private key issued that VCR, or the adversary performed
        a long-term key reveal on the server, or the adversary knows the client's
        derived private key. */
102 lemma VCR_unforgeability:
103   " All server dskC cookie request #i.
104         Accept_VCR(server, pk(dskC), <cookie, request>) @ i
105       ==>
106           (Ex #j. Issue_VCR(dskC, server, <cookie, request>) @ j & j < i)
107         | (Ex #r. RevSk(server) @ r) | (Ex #r. KU(dskC) @ r) "
108
109 /* Replay resistance: the server will not accept a VCR for the same cookie and
        request combination more than once, unless the adversary knows the client
       's derived private key. */
110 lemma replay_resistance:
111   " All server dskC cookie request #i #j.
112         Accept_VCR(server, pk(dskC), <cookie, request>) @ i &
113         Accept_VCR(server, pk(dskC), <cookie, request>) @ j
114       ==>
115           #i = #j | (Ex #r. KU(dskC) @ r) "
116
117 /* Consistency check: the server can accept a VCR without the adversary having
        performed a long-term key reveal on the server or knowing the client's
       derived private key. */
118 lemma accept_vcr_possible:
119   exists-trace
120   " Ex server dskC params #i.
121       Accept_VCR(server, pk(dskC), params) @ i
122     & not (Ex #r. RevSk(server) @ r)
123     & not (Ex #r. KU(dskC) @ r) "
124
```

```
125 end
```

Listing A.1: Tamarin specification of the messages exchanged in VICEROY, and the corresponding security lemmas.