



UNIVERSITY OF
CAMBRIDGE

Department of Computer
Science and Technology

Investigating the feasibility of using CHERI-enabled Arm Morello boards for cloud-based trusted execution

Kian Cross

Jesus College

June 2023

Submitted in partial fulfillment of the requirements for the
Master of Philosophy in Advanced Computer Science

Total page count: 67

Main chapters (excluding front matter, references, and appendices): 49 pages (pp. 7–55)

Main chapters word count: 14 932

The above word count was generated using the following command:

```
texcount -0 -sum=1,1,1,0,0,1,1 -merge -q main.tex
```

Declaration

I, Kian Cross of Jesus College, being a candidate for the Master of Philosophy in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: 

Date: 2nd of June 2023

Abstract

As the usage of cloud computing services continues to grow, the need for secure processing environments to complement existing secure storage and transport services is becoming increasingly important. This project explores the possibility of using hardware enhanced RISC instructions (CHERI) to build cloud-based trusted execution environments (TEEs).

To achieve particular security guarantees, existing TEEs often employ specialised hardware to enforce mutual distrust between applications and higher-privileged software components. After exploring concepts relating to trust and trusted execution, this project examines these related systems, focusing on their policy objectives, implementation mechanisms, and limitations.

Following this, I explore the trusted execution policy space available under the Arm Morello board, a recently released multicore, superscalar, experimental ARMv8-A-derived instantiation of CHERI. I further consider how minor modifications to this mechanism could support more ambitious policy objectives.

This culminates in the design of TERIOS, a cloud-based multi-tenant TEE platform developed for the Arm Morello board. Consisting of six small components partitioned along privilege boundaries to enable least-privilege compartmentalisation, TERIOS provides partial mutual distrust between TEEs and privileged code.

An experimental prototype implementing a subset of TERIOS is developed in assembly and C, targeting baremetal Morello. This single-address-space, pure-capability prototype primarily serves to exhibit the limits of Morello boards for achieving robust trusted execution policies. Specifically, it demonstrates the impossibility of ensuring total opacity of TEEs from privileged software due to the absence of hardware-facilitated context switches and memory allocation.

This work concludes by finding that there are valid reasons for reapplying CHERI to facilitate TEEs and that when deployed in the cloud, there are compelling arguments for relaxing the threat model concerning hardware attacks. However, the TEE security assurances that CHERI currently provides are limited. Achieving more robust policy objectives would necessitate extending CHERI or integrating existing hardware. Without developing and prototyping these enhancements, the feasibility and performance impacts remain uncertain. These challenges offer novel insights into the potential of CHERI-enabled hardware for trusted computing.

Acknowledgements

I am grateful to Prof. Jon Crowcroft, Dr. Carlos Molina-Jimenez, and Dann Toliver for their support and guidance during this project. Jon provided valuable oversight, and I have appreciated his grounding reminders that research can be useful even if it does not fully achieve its original aims. Carlos has spent significant time providing feedback and has facilitated my attendance at insightful workshops, for which I am grateful. Dann's input on the project's direction, particularly on subtle policy considerations, was especially important. All three have welcomed me into the CAMB project, enabling me to experience an active research environment. Finally, I appreciate those who have dedicated their time and effort to proofreading this work.

Contents

1	Introduction	7
1.1	Research environment	8
1.2	Motivations and aims	8
1.3	Contributions	9
2	Background	10
2.1	Definitions	10
2.1.1	Trust and trustworthiness	10
2.1.2	Threat model	11
2.1.3	Trusted computing base	11
2.1.4	Root of trust	12
2.1.5	Attestation	12
2.1.6	Trusted execution environment	14
2.1.7	Confidential computing environment	15
2.2	Commercial usage of trusted execution services	15
2.3	Capability hardware-enhanced RISC instructions	17
2.3.1	Fundamental CHERI concepts	17
2.3.2	Arm Morello	19
3	Analysis of related systems	21
3.1	Hardware assisted	21
3.1.1	Intel SGX (SCONE and Gramime)	22
3.1.2	Arm TrustZone (OP-TEE)	24
3.1.3	CheriOS	25
3.1.4	CHERI-TrEE	27
3.2	TCB minimisation	28
3.2.1	seL4	28
3.2.2	The CHERIoT RTOS	28
4	Design	30
4.1	Requirements	30
4.2	Threat model	31
4.2.1	Exclusion of physical attacks	32
4.3	TERIOS architecture	33
4.3.1	Attester	33
4.3.2	Capability manager	37
4.3.3	Heap allocator	37

4.3.4	TEE switcher	38
4.3.5	TEE scheduler	38
4.3.6	TEE manager	38
4.3.7	Execution environment	39
4.4	Evaluation	39
5	Prototype implementation	42
5.1	Development environment	43
5.1.1	Morello software stack	43
5.1.2	Code compilation and execution	44
5.1.3	Arm Development Studio	45
5.2	Implementation details	46
5.2.1	Memory management unit	47
5.2.2	Generic interrupt controller	48
5.2.3	Exception handling and context switching	48
5.2.4	Dynamic relocations	48
5.3	Evaluation	51
6	Conclusions	53
6.1	Reflections	53
6.2	Limitations and future work	54
6.3	Final remarks	54
	Bibliography	56
	Appendix A Software artefact	67



Chapter 1

Introduction

Data is frequently encrypted in transit [36, 55, 66, 76, 169] and at rest [35, 42, 155]. Such mechanisms allow information to leave trusted environments; for example, to traverse the Internet or for storage at a third-party backup site. However, other than in some limited instances [62, 78, 119, 136], encrypted data cannot be operated upon. As such, the processing of this information has traditionally been restricted to trusted environments.

With the rise of cloud computing [141], those who consider the cloud provider’s infrastructure to be untrusted are deprived of the many benefits that such services can offer, including reduced capital investment and rapid scalability [2, 147]. If users could protect data whilst *in use*, they might be able to rely on cloud processing services as they do cloud storage services. This idea is known as *confidential computing*.

The commercial motivations behind cloud-based confidential computing offerings are articulated clearly by IBM, a stakeholder in the domain [162]:

The primary goal of confidential computing is to provide greater assurance to leaders that their data in the cloud is protected and confidential, and to encourage them to move more of their sensitive data and computing workloads to public cloud services [84].

The market for confidential computing is estimated to grow by between six and 26 times over the next five years [59]. As a result, the technical solutions to enable such services — or problems associated with them — will become increasingly important.

1.1 Research environment

Facilitating confidential computing requires specialised hardware [49, 139], some of which has been found to contain critical issues [1, 45, 51]. Implemented in Arm Morello prototype boards at the start of 2022 [71], capability hardware enhanced RISC instructions (CHERI) [176] extends RISC [137, 159] to support granular memory protection and efficient software compartmentalisation [176, 178]. Cloud attestables on Morello boards (CAMB) is a project at the University of Cambridge, started in September 2022, aimed at utilising these features to provide an alternative hardware environment for cloud-based trusted execution [118]. This project has been undertaken in conjunction with CAMB, and given the shorter time frame, focuses on a subset of their aims.

1.2 Motivations and aims

This work investigates the hypothesis that ‘developers can use CHERI-enabled Arm Morello boards to implement cloud-based trusted execution environments’. Considerations of this statement can generally be categorised as ones of either *policy* or *mechanism*. Trusted execution environments typically require a method to enforce mutual distrust between applications and higher-privileged components of the system. Often, this is achieved using specialised hardware, which facilitates secure execution environments, combined with complementary orchestration software. These aspects are the ones of *mechanism*. Considerations, such as the degree to which mutual distrust is required — a subset of the overall threat model — are ones of *policy*. Both are closely related; the policy goals might be constrained if the available mechanisms are fixed. In the context of trusted execution, this work explores the policy space available under the Morello boards and how small changes to this mechanism might enable more ambitious policy goals.

Exploring these policy considerations is essential. The additional security offered by traditional trusted execution environments comes at a performance cost [27, 40, 56, 72, 177, 182]. The policy space is vast, and there is possibly a point within it — equating to a relaxed threat model which still fulfils practical requirements — with a smaller performance penalty. Moreover, reusing a multipurpose technology, such as CHERI, might offer benefits compared to relying on additional specialised hardware.

Within the broader CAMB project, this work could be considered a feasibility analysis or exploratory study, which can be further built upon. Indeed, the primary outcome of this project is the exploration, justification, design, and prototype of an alternative environment for trusted computing based on the Arm Morello board platform. This constitutes a novel contribution to the field, with the consideration of CHERI-enabled hardware for trusted computing still in the early stages.

1.3 Contributions

The following contributions have been made through this work:

- Fundamental concepts relating to trust and trusted execution are defined and explored. By analysing real-world cloud-based trusted execution services, this work extracts essential requirements, facilitating a policy discussion grounded in genuine needs and practical considerations (§ 2).
- A comprehensive analysis of influential existing systems, encompassing the entire trusted-execution policy space, is conducted. This examination distinguishes between the policy and mechanism aspects of these systems, offering a critical evaluation and providing a better understanding of where a CHERI-based system could fit within the landscape of cloud-based TEEs (§ 3).
- Drawing upon the previous review, a design for a CHERI-based system — TERIOS — is outlined and evaluated. TERIOS is a cloud-based multi-tenant TEE platform developed for the Arm Morello board, providing partial mutual distrust between TEEs and privileged code. With a focus on justifying the threat model of TERIOS with respect to real-life requirements, this work addresses an aspect not fully considered by existing CHERI-based systems (§ 4).
- An experimental prototype, implementing a subset of TERIOS, is developed to support the design process, uncover practical challenges, and expose limitations of the Morello hardware. In addition, this single-address-space, pure-capability prototype offers valuable documentation and guidance for establishing an environment for baremetal development and debugging on the Morello platform (§ 5).

In the final chapter of this work, limitations are outlined, future work is presented, and conclusions are drawn (§ 6).



Chapter 2

Background

This chapter provides the background material necessary to understand the hypothesis presented in § 1.2. First, relevant terms are defined and discussed, particularly relating to what ‘trusted execution’ really means (§ 2.1). Next, commercial expectations and motivations for utilising trusted execution services are outlined (§ 2.2). Finally, a short introduction to CHERI and Arm Morello is presented (§ 2.3).

2.1 Definitions

A crucial part of the hypothesis is the phrase ‘cloud-based trusted execution environments’. Hence, it is essential to understand what it means to ‘trust’ a cloud-based environment. Accordingly, this section starts with a discussion around trust (§ 2.1.1 and § 2.1.2) and how this relates to components of a computer system (§ 2.1.3 and § 2.1.4). The remainder of the section presents definitions of other relevant terms used throughout the work.

2.1.1 Trust and trustworthiness

At its core, *trust* is a belief that an entity is reliable and honest; it is a choice. *Trustworthiness* is the degree to which an entity can be trusted. It is a relative term that depends on the perceptions and expectations of those who place their trust in the entity. Because trust and trustworthiness are inherently subjective and imprecise concepts, they are challenging to reason about and quantify. Moreover, they are influenced by complex social and economic factors (e.g., mar-

ket positioning, reputation, customer service, product quality, company leadership, jurisdiction, geopolitical circumstances etc.) and are considerations of human behaviour rather than strictly technical.

However, there are objective factors that can help assess the trustworthiness of an entity. For instance, certifications from recognised bodies can indicate compliance with standardised security specifications [92, 93, 94]. A track record of trustworthy behaviour can also bolster an entity's trustworthiness. In the case of a computing platform, comprehensive software testing and a large customer base can increase confidence in its reliability. Some may also prefer entities with a smaller attack surface, measured by the size of their software stack, as this can reduce the number of potential vulnerabilities.

The level of trustworthiness required for different entities varies depending on the potential consequences of a breach of trust. For example, a company processing sensitive medical information must place more trust in the computing platform than a music streaming platform might require. This is because a breach of trust resulting in the leak of confidential medical data could have far more severe consequences, both for the individuals affected and for the company responsible. In such cases, breaches could lead to significant legal and financial liabilities, including compensation for damages, loss of reputation, and regulatory fines [77, 129].

2.1.2 Threat model

A *threat model* is a systematic approach for documenting and analysing potential threats to a system, focusing on anticipating attack goals [161, p.26]. The model identifies the system's entry points — its interfaces with the outside world — and describes which of these are accessible to adversaries and which are 'out of scope'. As such, the threat model captures the capabilities of potential attackers and the possible attack vectors they may use, providing a structured representation of how adversaries may interact with the system.

A threat model can be developed before the implementation of a system as part of the specification or afterwards as part of the documentation. A system is deemed secure if it is resilient to exploitation with respect to a specific threat model, which should be considered adequate by the entity responsible for assessing the system's security. The threat model usually incorporates notions of trust, where certain parts of the system are considered trustworthy and assumed to be resistant to exploitation via a particular entry point. Thus, determining the adequacy of a threat model depends primarily on whether the trust assumptions made are realistic and appropriate for the system's use case. Many of the considerations outlined in § 2.1.1 also apply here.

2.1.3 Trusted computing base

The *trusted computing base* (TCB) of an application P is the subset of the system on which P operates — including hardware, firmware, and software — whose failure could result in P being compromised [108]. In this work, it is argued that the definition of the TCB should

be expanded to include non-tangible aspects of computer systems, such as code deployment processes, physical access restrictions, and employee vetting procedures. These can be encompassed by a single component representing the operating entity (e.g., the cloud provider or hardware vendor). By broadening the TCB to include these elements, a more comprehensive and nuanced understanding of the trustworthiness of a system can be provided.

The threat model determines which system components are part of the TCB. Examples of the TCB for various computer systems are shown in Figure 2.1 and Figure 2.2.

The compiler and other aspects of the compilation toolchain are easily overlooked when considering the trusted components of a system. The toolchain could introduce accidental or malicious vulnerabilities unless the resulting machine code is verified. In practice, only closely related trusted components are typically considered, although, in theory, the trust chain could be examined and extended extensively.

2.1.4 Root of trust

The *root of trust* is the foundational component from which the security of the remainder of the system is derived. It corresponds to the blue leaf nodes shown in Figure 2.1. The figures distinguish between the conceptual entity that manufactures or operates the root of trust (e.g., a company) and the actual component of the computer system. Other entities trust the former, whilst the latter is trusted or relied upon by other hardware or software.

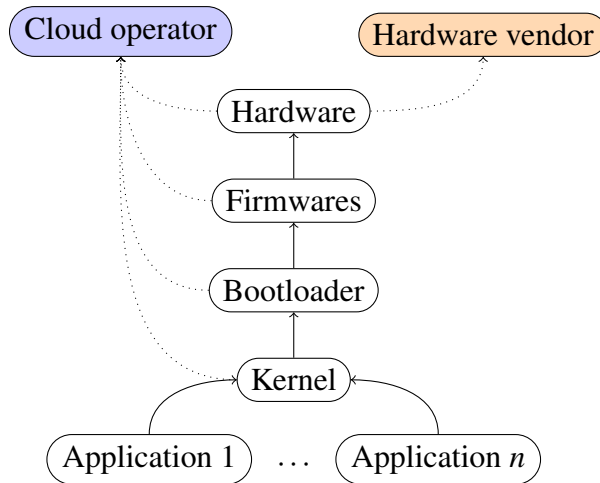
In some cases, multiple entities may appear to act as the root of trust. For example, Figure 2.1a illustrates that there is a cloud operator, hardware vendor, possibly closed-source firmware vendors (not shown), and perhaps even the kernel developer if a closed-source operating system (OS) is used. However, typically only one entity, such as the cloud operator, subsumes the responsibility of verifying vendors, procuring reliable hardware, and inspecting open-source code; end-users consider this entity the root of trust.

The specific threat model determines the adequacy of a root of trust. For instance, standard hardware may provide sufficient protection mechanisms for non-sensitive applications, but may be insufficient for exfiltration-sensitive computations.

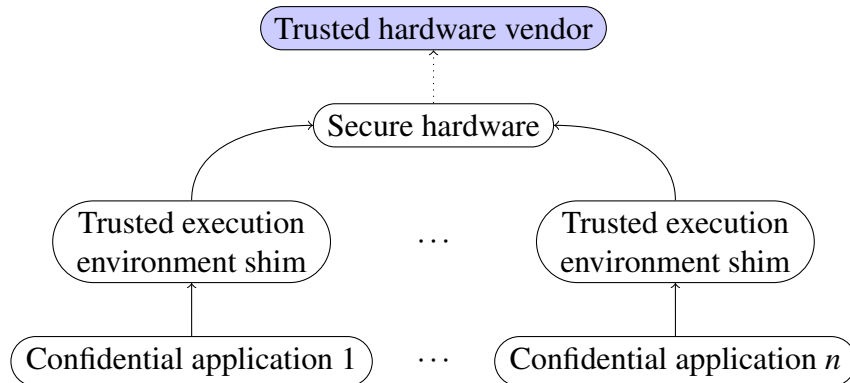
2.1.5 Attestation

The process of *attestation* involves a trusted *verifier* assessing the trustworthiness of *evidence* provided by a potentially untrusted *attester* to determine the state of a local or remote system with some degree of confidence [38, 164]. By performing attestation, it is possible to verify a system's integrity and security properties before sensitive data is exchanged. In this work, the entity relying on the verification of the evidence is considered the same as the verifier, although this is not strictly necessary [38].

Software-based attestation comes in various forms [96, 135, 152, 153, 156, 158, 179], but



(a) Illustration of the trust relationships for an application running on a typical operating system within a cloud data centre. Each application relies on a large TCB, where bugs or malicious code could potentially compromise its secure operation. To ensure the system's security, both the cloud operator and hardware vendor must be trusted to implement and operate the various components of the system correctly and without malice. In reality, trust relationships are more nuanced and complex, with dependencies on entities such as closed-source firmware vendors being just one example.



(b) Illustration of the trust relationships for a confidential application running within a trusted execution environment (TEE). The TEE provides a secure execution environment for the application, with the TCB consisting of a small set of components, including an inspectable shim. The secure hardware serves as the root of trust and can remotely attest that the software provided by the client is being executed on the hardware. Unlike traditional cloud deployments, the cloud operator does not need to be trusted, as the TEE guarantees the application's confidentiality and integrity.

Figure 2.1: These examples show directed graphs illustrating the trusted computing bases (TCBs) of two different systems. Each node represents a system component, and an arrow from a node P to Q indicates that Q is in the TCB of P . The nodes reachable from a node P constitute P 's TCB. Dotted lines represent conceptual trust relationships that do not correspond to physical components of the system. The blue leaf node of the TCB is known as the root of trust, which serves as the foundation for the security of the entire system.

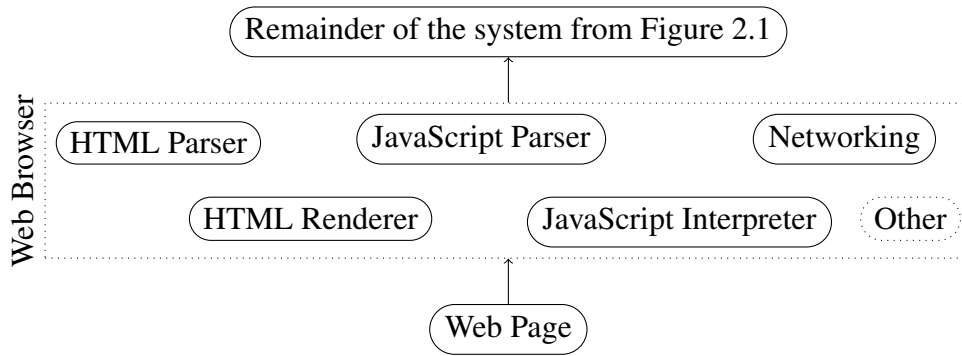


Figure 2.2: The trusted computing base (TCB) of a web page, illustrated here, consists of the web browser and all system components below it in the hardware and software stack.

it is susceptible to issues such as man-in-the-middle attacks, which are difficult to mitigate without introducing restrictive and impractical limitations [8]. As an alternative, specialised hardware components can be used to provide evidence for attestation. These components are trusted (e.g., manufactured correctly and free from vulnerabilities) [53, 160] and form part of the application’s TCB; in fact, they often serve as the root of trust.

In the cases considered in this work, the hardware component takes a measurement of the code and data state [165] — a hash — which serves as the evidence. The verifier can use this to ensure it matches the expected code and data state. This method is suitable for *local attestation*, where the presence of this hardware and the overall physical security of the system can be observed by the verifier. However, when the system is remote and there is no way to verify that the code is being executed on the expected hardware configuration, *remote attestation* must be used. This process is similar to local attestation, but the measurement is cryptographically signed by the hardware, making it unforgeable [154, 164].

2.1.6 Trusted execution environment

A *trusted execution environment* (TEE) is a secure computing environment which provides data and code *integrity* and data *confidentiality* [164]. Integrity prevents unauthorised entities from adding, deleting, or altering data and code whilst in use within the TEE. Confidentiality prevents unauthorised entities from viewing data in use within the TEE. In this work, TEEs are assumed programmable, meaning they can execute arbitrary code. The term *trusted execution* is used to describe the action performed by a TEE, whilst *trusted execution service* refers to a commercial solution, typically cloud-based, which provides some form of TEE.

The guarantees provided by the term ‘prevent’ and the scope of what is considered an ‘unauthorised entity’ depend on the specific threat model. An unauthorised entity may typically include other userspace programmes, the kernel, and potentially anyone with access to the hardware. However, this work adopts an abstract definition of a TEE, which depends entirely on the given

threat model and is without specific reference to any particular entity or system.¹

2.1.7 Confidential computing environment

The Confidential Computing Consortium defines *confidential computing* as ‘the protection of data in use by performing the computation in a hardware-based, attested trusted execution environment’ [164].² In this work, the term *confidential computing environment* will refer to a system that provides such protection. This definition strengthens the concept of a TEE by specifying it as hardware-based, implying more robust security properties. Confidential computing environments represent a specific type of TEE.

2.2 Commercial usage of trusted execution services

Numerous cloud providers, such as Google Cloud Platform [67], Microsoft Azure [114], IBM Cloud [83], and Amazon Web Services (AWS) [4, 41, 143], offer confidential computing services. In order to propose viable alternatives for the underlying technologies, it is essential to understand why users are drawn to these services and what they expect from them.

A fictional proposal is presented next to illustrate better the process of migrating from an on-premises computing infrastructure to a cloud-based computing infrastructure. This concrete scenario is synthesised from various sources [37, 82, 84, 115], which provide general motivations for such a transition. This example captures only some of the considerations; the factors influencing decision-makers are often situation-dependent and may vary from case to case.

Medabase is a company that manages and processes patients’ medical information on behalf of multiple healthcare providers. Their computing infrastructure consists of several Intel-based servers housed in a small data centre adjacent to their headquarters. Maintaining this data centre requires round-the-clock staffing, diverse expertise, contracts with numerous external companies, and complex contingency plans.

The head of IT has been advocating for a migration to a cloud-based computing platform, citing benefits such as cost savings, increased reliability, and scalability to match the company’s growth. However, the executives are highly risk-averse, expressing concerns that cloud provider employees could alter the code on their servers to exfiltrate sensitive patient data. A data breach would result in a catastrophic loss of trust, costly lawsuits, and an exodus of clients, causing severe damage to the company. Despite the head of IT’s assurances of legal processes in place to deter and punish such behaviour, the executives remain unconvinced, insisting on a *preventative* solution rather than a *reactive* one, with legal remedies only offering the latter.

¹The flexibility of this definition might suggest the possibility of labelling any computer system as a TEE by using unorthodox and weak meanings of ‘prevent’ and ‘unauthorised entity’. However, this is not helpful. To maintain some pragmatism, the meaning of a TEE is imprecisely restricted to imply a more secure environment than offered by typical operating systems, without necessarily requiring specialised hardware [146].

²A cloud-based attested TEE — with some particular properties — is what the CAMB project refers to as an *attestable*. However, this work will avoid using the term, instead favouring the definitions given in this section.

Eventually, the head of IT discovers Cloudcomp, a cloud computing platform which offers a confidential computing service using Intel Software Guard Extensions (SGX), an attested TEE. He presents this option to the executives, explaining that just as they trust Intel to provide hardware for their on-premise data centre, they can continue to rely on *only* Intel in the cloud. This proposal resonates with the executives, as it eliminates the need to trust any additional entities. Moreover, Intel SGX offers a *preventative* solution to data exfiltration rather than a reactive measure. Consequently, the company begins its transition to a cloud-based computing infrastructure.

This example captures the following points: (1) the typical drivers for adopting cloud-based computing infrastructure apply, such as reduced capital costs, potential reductions in marginal costs, and increased scalability [2, 147]; (2) decision-makers are concerned about the privacy of their users' sensitive data; (3) decision-makers are seeking technical solutions which proactively prevent data breaches rather than reactive solutions such as legal recourse; (4) maintaining the same root of trust across the migration is beneficial; (5) the goal is not to eliminate all risk of trust being breached but rather to reduce it to a tolerable level that aligns with other risks; and (6) the decision to migrate is as much a business decision as a technical one, which can sometimes be difficult for technical professionals to grasp and accept.

Many of these observations are supported by market research. A study conducted by Futurum in collaboration with IBM revealed that 80% of surveyed enterprises did not believe compliance with industry certifications alone was sufficient to prevent data breaches. Additionally, 82% expressed concerns about their cloud provider being able to access their data, with 93% preferring that the provider cannot access it under any circumstances [125].³

Inevitably, a company must place trust in various entities and processes, including employees, physical security measures, hardware providers, and cloud-computing providers. Relying on both the cloud company and hardware provider — which are often, but not always, separate entities — introduces the concept of *distributed trust* [53]. Given the dependence on more than one *trust domain*, both entities would need to behave in a manner which breaches trust for negative consequences to arise.

Nonetheless, the situation is not so straightforward. For instance, AWS offers confidential computing services on in-house designed and manufactured silicon [41] — part of their Nitro system [5] — challenging the notion of distributed trust. There may be a perception that hardware is generally less susceptible to attacks compared to software, a sentiment echoed in an AWS blog post that states '[AWS] engineered the [system] with a hardware-based root of trust', which 'provides a significantly higher level of trust than can be achieved with traditional hardware or virtualisation systems' [41]. Moreover, the blog highlights that '[AWS] have been...moving more and more virtualisation functions to dedicated hardware and firmware' [41], potentially enhancing operational security by reducing the number of individuals with deployment access. However, although the hardware is designed in-house, the team is part

³This study was a commercially commissioned market research project. Unfortunately, the methodology was not disclosed, so the results should be interpreted cautiously.

of a wholly owned subsidiary, which may offer a degree of operational independence that reassures users. This showcases a feasible position in the policy spectrum where there is no need for a trusted third party.

2.3 Capability hardware-enhanced RISC instructions

Between 59% and 70% of operating system vulnerabilities are caused by programmes accessing data that they are not supposed to [54, 116, 163], a problem known as *memory safety*. Whilst various software-based mechanisms have been developed to mitigate this [28, 46, 50, 151, 157], none have adequately solved the issue. It is argued that this is because software-based memory safety mechanisms are underpinned by hardware not designed with this type of security in mind [64].

This prompted the development of CHERI — which stands for ‘capability hardware enhanced RISC instructions’ — an architectural extension to enable granular memory protection and efficient software compartmentalisation [128, 178]. This work explores the trusted execution policy space facilitated by the CHERI mechanism.

To this end, a brief introduction to CHERI and its features is provided (§ 2.3.1), omitting microarchitectural details. This is followed by a short description of the Arm Morello project (§ 2.3.2), which has implemented CHERI in a prototype board and serves as a valuable platform for experimentation and evaluation of the CHERI architecture.

2.3.1 Fundamental CHERI concepts

Capabilities

The key innovation of CHERI is the introduction of architectural *capabilities*, which replace traditional pointers⁴ and restrict the access of programmes to designated regions of memory. Capabilities also attach semantics to regions of memory, which the hardware can enforce as permissions. Together, these features prevent unintentional memory accesses and prevent pointers from being used for unintended purposes, such as executing instructions from memory intended for data storage.

Capabilities are analogous to pointers but with additional metadata, often called ‘fat pointers’. As shown in Figure 2.3, a capability on a 64-bit system includes a full-precision address, along with compressed upper and lower bounds that define the region of memory being referenced. These bounds are compressed using a floating-point representation, which trades accuracy for space efficiency when dealing with larger addresses. Capabilities also include permissions, which allow or restrict memory access (such as permitting reads, writes, and executes, as well as other software-defined privileges [172]).

⁴CHERI offers two compilation modes: *hybrid*-capability code and *pure*-capability code [172]. Standard pointers and capabilities can be used within the same programme in the hybrid mode, whilst the pure mode only supports capabilities. This work focuses exclusively on pure-capability code.

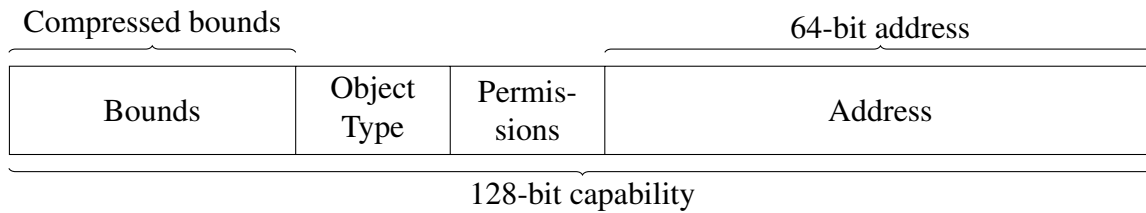


Figure 2.3: This figure depicts a *capability*, which replaces traditional pointers. On a 64-bit system, the capability includes a full-width 64-bit address, compressed bounds to restrict memory access, permissions that attach semantic information to memory, and an object type used for sealing the capability. Representation of the tag bit is omitted, as it is typically stored separately from the capability [97, 171]. This figure is adapted from a technical report by Watson et al. [172].

Provenance validity and capability monotonicity

One of the fundamental concepts of CHERI is *provenance* validity and *capability monotonicity* [175]. Provenance refers to the requirement that a capability must be derived from another valid capability, whilst monotonicity states that a capability cannot have wider bounds or greater permissions than its parent. This idea is illustrated, with examples of invalid behaviour, in Figure 2.4.

To preserve *integrity*, a tag bit which indicates the validity of a capability is cleared whenever any byte manipulation of the capability is performed. An invalid capability is non-dereferenceable, and a valid capability cannot be derived from an invalid capability. This ensures that the provenance and monotonicity properties are not violated [174].

As a consequence of these properties, it can be formally proven that during the execution of arbitrary code, the regions of memory and permissions represented by the set of *reachable capabilities* cannot be increased [127, 172]. This is known as the *protection domain*. Reachable capabilities include those currently in registers, those that can be loaded from memory accessible by capabilities in registers, and any that can be derived from these capabilities.

Controlled non-monotonicity

There is an exception to this principle of *reachable capability monotonicity*, known as *controlled non-monotonicity*. Two mechanisms exist to allow this: exception handling and the invocation of sealed capabilities [172]. Both mechanisms enable switching the protection domain. In the case of exception handling, control is transferred to a known and protected handler, which may have access to higher-privileged capabilities, thereby non-monotonically increasing the reachable capability space.

Sealing a capability marks it as immutable and non-dereferenceable, meaning that no other capability may be derived from it [174]. The object type field of an unsealed capability is set to -1 .⁵ When a capability is sealed, its object type is set to the address of another *authorising*

⁵This holds for the abstract CHERI protection model, although specific implementations may opt for alternative values, such as zero [12].

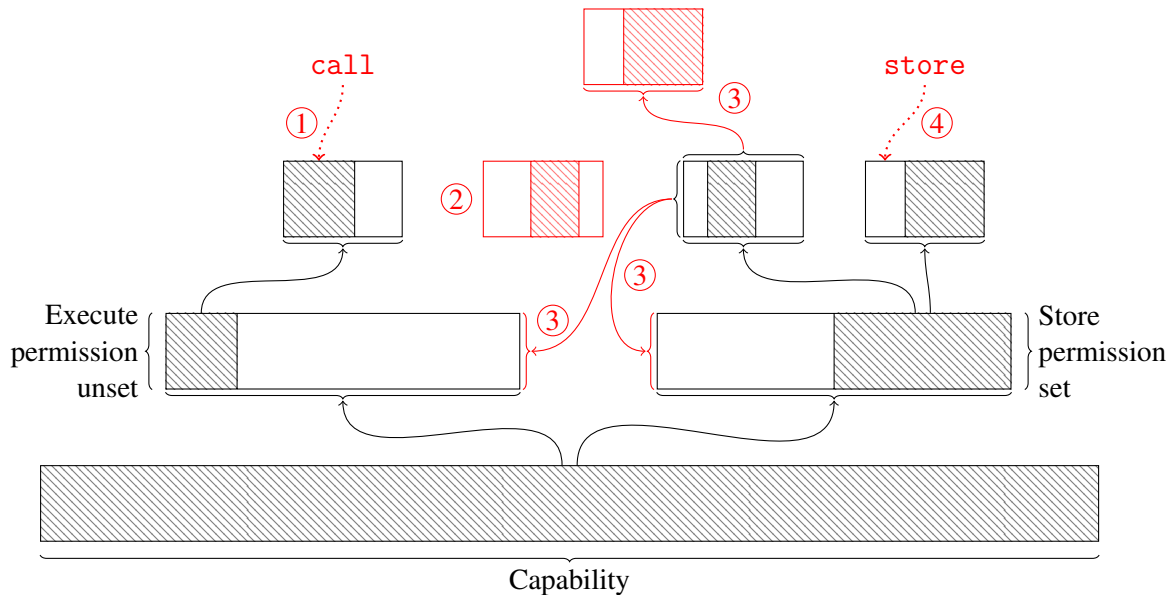


Figure 2.4: This figure illustrates the construction of new capabilities, with shaded regions denoting the bounds of the derived capabilities and arrows indicating the derivation process. Invalid behaviours are highlighted in red. ① Invoking a `call` instruction with a capability derived from another capability without the execute permission violates *monotonicity* and results in an invalid invocation due to inadequate *permissions*. ② To maintain *integrity*, a capability must be derived from a valid capability, known as having valid *provenance*. For example, constructing capabilities through byte manipulation is prohibited. ③ *Monotonicity* prevents the derivation of child capabilities with wider bounds or greater permissions than their parents. ④ Capabilities cannot be used to access memory beyond their bounds.

capability.⁶ The bounds of an authorising capability indicate the region of capabilities it is permitted to seal. Likewise, an authorising capability can unseal a capability, provided it has the unseal permission. Alternatively, an instruction can be used to unseal a code capability and jump to it atomically, or similarly to invoke a pair of code and data capabilities with matching object-type fields. These latter methods induce a domain transition, enabling a non-monotonic increase of the reachable capability space, which is comparable to an exception; however, the exception level⁷ remains unchanged.

2.3.2 Arm Morello

CHERI is an abstract protection model which has been implemented in numerous instruction set architectures (ISAs), including the experimental Arm Morello architecture [12], which applies CHERI to the ARMv8-A [11] architecture [174]. The Morello project aims to evaluate CHERI, with the hope of providing evidence for its further adoption and development [171]. To this end, in early 2022, Arm released a prototype system-on-chip (SoC) based on the Mo-

⁶Certain implementations may attribute special meanings to other object type values, which do not necessarily correspond to the addresses of authorising capabilities [12].

⁷Exception levels in the Arm architecture, equivalent to protection rings, range from EL0 (the lowest software execution privilege) to EL3 (the highest). Each level permits varying degrees of access and control over the processor [21]. The configurability at each level is typically restrictable by higher levels. For instance, a higher privilege level can prevent a lower one from modifying its virtual memory mapping.

rello architecture [71], which offers a multicore, superscalar platform for experimentation and evaluation.

Some specific aspects of the ARMv8-A architecture are covered in § 3.1.2 and § 5; however, it is beyond the scope of this work to provide a comprehensive summary. Similarly, Morello — which, for example, extends ARMv8-A with a capability stack pointer (CSP) and capability program counter (PCC) — is not formally covered but will be explained in context as needed. The citations in the previous paragraph may be used as a source of additional information for both.

Analysis of related systems

Numerous systems aim to improve security, with many also facilitating trusted execution. In this section, several of these systems will be examined, with a focus on both their implementation methods and policy objectives. Investigating the current policy landscape aims to determine where a system based on the CHERI-enabled Arm Morello architecture could fit. This discussion is divided into two sections: systems utilising specialised hardware for enhanced security (§ 3.1) and those seeking to improve security by reducing the size of the trusted computing base (§ 3.2). A summary is provided in Table 3.1.

3.1 Hardware assisted

Specialised hardware is crucial for improving system security and provides a foundation for implementing various systems. For instance, Intel SGX forms the basis of systems like SCONE (§ 3.1.1), whilst Arm TrustZone enables solutions such as OP-TEE (§ 3.1.2). Furthermore, numerous systems have been developed using CHERI (§ 3.1.3 and § 3.1.4). In the following sections, the hardware underlying these systems is explored.

Table 3.1: Comparison of various systems designed to enhance security through hardware mechanisms, trusted computing base minimisation, or a combination of both.

System	Mutual distrust	Minimised TCB	Hardware only TCB	Remote attestation	Encrypted memory ¹	Formally verified ²
Intel SGX	✓	✓	✓	✓	✓	✗
Arm TrustZone	✗	✓	✗	✗	✗	✗
CheriOS	✓ ³	✓	✗	✗	✗	✗
CHERI-TrEE	✓ ³	✓	✗	✗	✗	✗
seL4	✗	✓	✗	✗	✗	✓
CHERIOT RTOS	✗	✓	✗	✗	✗	✗

¹A tick denotes systems that, in practice, guarantee or implement encrypted memory. Systems that are merely compatible with encrypted memory but do not mandate or directly enable it are not marked as such.

²Only systems with published and reproducible formal verification methodologies are marked with a tick.

³This holds true, with the exception of a trusted interrupt handler which can access registers.

3.1.1 Intel SGX (SCONE and Gramime)

SCONE is a ‘secure container mechanism for Docker’ [27], which leverages Intel SGX’s [49] hardware-based isolation to protect unmodified applications and their associated data. The secure C library within SCONE facilitates the encryption and decryption of input and output in an SGX enclave. This enables developers to run applications with enhanced confidentiality, even when the underlying operating system may be compromised. Gramine, a similar system to SCONE, also offers comparable functionality [166, 167].

Threat model

The Intel SGX threat model presumes a powerful adversary capable of controlling the entire software stack, such as through a compromised operating system (e.g., dropping packets and delivering malicious data via system calls, as demonstrated in Iago attacks [43]). This adversary is also assumed to have physical access to the hardware and the ability to carry out certain hardware-based attacks, which are discussed further towards the end of this section. The trusted hardware is assumed to be free of vulnerabilities and manufactured correctly, positioning Intel as a trusted entity and, consequently, a single point of failure.

Implementation

Enclaves. A fundamental concept within Intel SGX is the enclave, a secure memory region within an application’s address space [111]. Enclaves establish a type of TEE that is specifically designed to withstand the threat model outlined in the previous section, facilitating mutual distrust between code protected within an enclave and code outside. The enclave page cache serves as a protected region of memory used to store an enclave’s pages and other SGX structures [111]. The CPU enforces access controls on this cache to ensure that only the code executing within the enclave can access its memory. This provides isolation from other processes and system software, such as the operating system and hypervisor [111]. In addition, the

memory encryption engine [49] allows the cache to be stored on untrusted DRAM [89, § 36.5.1] by dynamically encrypting and decrypting data during memory read or write operations [7].

Attestation. Intel SGX supports remote attestation, enabling an enclave to demonstrate its identity and integrity to a remote party. Remote attestation is based on a signed data structure known as a *quote*, which is derived from a *report* containing information about the enclave (such as its measurement, which is a hash of the loaded code and data) [79]. The report is signed with an attestation key [7] obtained during the platform’s provisioning process [154]. Provisioning can take place at any time and occurs within a special provisioning enclave, which implements a signature scheme called enhanced privacy ID [154]. This scheme establishes its authenticity to Intel’s online provisioning servers using keys derived from two other on-SoC keys embedded at different stages of manufacturing [154]. Upon successful authentication, the provisioning servers supply an attestation key, which is encrypted and stored for subsequent use in attestation.

Context switching. During a standard context switch, the operating system typically saves the current thread’s register file and loads the new thread’s register file. However, this exposes the confidential registers to the OS, which is considered untrusted in the Intel SGX threat model. To address this issue, the SGX hardware provides facilities for secure context switching between code within enclaves and code outside of an enclave. An enclave is entered using the EENTER instruction, accompanied by an SGX enclave control structure (SECS). Execution proceeds until completion or an exception occurs (e.g., a timer interrupt). Upon such an exception, the current execution context is securely saved to the state save area (SSA) by the hardware, any confidential execution context is cleared, and control is transferred to a designated asynchronous exit pointer (AEP), which can invoke the appropriate OS exception handler [88]. Execution within an enclave can be resumed using the ERESUME instruction, along with a SECS, which loads the execution context from the SSA and continues execution within the enclave [49, § 5.4.4].

Discussion

Intel SGX has been found to possess multiple issues. One of the most significant is its vulnerability to side-channel attacks, such as Spectre [101] and Meltdown [106], which have compromised many of the security guarantees offered by SGX. In addition, numerous other cache-based attacks have been exposed [39, 52, 117, 150], further undermining it.

Various hardware attacks are also known [45, 60, 122]. Whilst the threat model does include some hardware attacks, the specifics of which are considered ‘in scope’ do not appear to have been thoroughly documented. DRAM and DRAM bus attacks are certainly included [90, p. 172], whilst other bus attacks, CPU chip attacks [90, p. 165], and power analysis attacks [90, p. 115] appear not to be [49, § 6.6.2]. Intel’s approach [68, 69, 98, 110] focuses not on making them impossible, but rather on making them prohibitively costly and technically challenging [49, § 6.6.2]. Consequently, the question arises: costly and challenging in relation to what and to whom? This is discussed further in § 4.2.

It is argued [144] that these problems, at least in part, resulted in the discontinuation of SGX from the recent generations of the Intel Core processors [86, 87]. However, development has continued in server-class Intel Xeon processors. Nevertheless, limitations [95], such as reduced performance [56, 182], persist.

3.1.2 Arm TrustZone (OP-TEE)

OP-TEE utilises Arm TrustZone¹ to create a secure and isolated environment for executing sensitive applications — such as those requiring secure storage [132] or cryptographic operations [133] — alongside an untrusted OS [131]. An OP-TEE application comprises two components: one that operates in the *secure world* and another in the *normal world*. Communication between the two is facilitated through an API implemented using Arm’s secure monitor call (SMC) [25] calling convention [134].

The secure world accommodates sensitive computations, whilst the normal world usually contains a standard operating system, such as Linux. Typically, applications within the normal world make synchronous calls to the secure world, positioning the normal world as the primary driver of secure world execution. However, this is not a strict limitation, and more intricate configurations — such as a secure operating system within the secure world — are feasible [20].

Threat model

The threat model presumes an attacker may gain full control over the software operating in the normal world [126]. However, Arm TrustZone is not designed to provide a complete security solution. As stated by Arm, the strategy should be to establish ‘a hardware architecture that extends the security infrastructure throughout the system design’ [20]. Consequently, many physical attacks fall outside the scope of the threat model. However, this does not preclude the possibility of protection against such attacks by other mechanisms.

Implementation

Execution boundary. A hardware-enforced execution boundary separates the secure and normal worlds, with various aspects of the hardware extended to maintain this separation. The advanced extensible interface bus incorporates a non-secure bit, which directs read and write operations towards secure or non-secure memory, as appropriate [10, p. 82]. This bit is further propagated to modified versions of the cache controller, direct memory access controller, and other peripheral buses [126]. The TrustZone address space controller and TrustZone memory adapter enable memory regions to be marked as secure or non-secure. Additionally, the generic interrupt controller (GIC) is adapted to allow prioritisation of secure interrupts, preventing denial-of-service attacks from non-secure sources [139].

¹Whilst OP-TEE is primarily designed for Arm TrustZone, it also offers compatibility with other isolation technologies [91, 131].

Context switching. An SMC, a hardware interrupt, or an abort signal can trigger the transition between the secure and normal worlds. Given the appropriate interrupt routing, these triggers can be managed by a handler at EL3 in the secure world, commonly called the secure monitor. If an SMC initiates the transition, the secure monitor directs the call to the appropriate secure application code. Once the secure application code has completed execution, the secure monitor oversees the transition back to the normal world, returning any results [126].

Secure boot. An attacker with physical access to the device could overwrite secure world software stored in flash memory, thereby breaching the system's security, which assumes that only the normal world can be compromised [20]. To counter this, TrustZone-enabled processors implement a secure boot process in which each boot level verifies the integrity of the next before proceeding. The root of trust is typically a unique public key stored on-SoC. This key is used to authenticate a secure boot loader — signed with the private key counterpart — stored in the on-SoC read-only memory (ROM). Each layer embeds the public key of the subsequent layer, utilising it to verify and subsequently load said layer. This sequence continues up to the point of loading the OS.

Discussion

Arm TrustZone is susceptible to various software and hardware vulnerabilities [47, 73, 105, 139, 180, 181]. Moreover, it lacks support for remote attestation, with the amount of memory that can be marked as secure also being limited [126]. Finally, one of the most substantial shortcomings of TrustZone is its monolithic structure, with secure code and data sharing the same protected environment. Only conventional separation techniques can provide further segmentation within this secure block.

3.1.3 CheriOS

CheriOS is a single-address space OS implemented using CHERI [58]. The system is mutually distrusting: the OS does not trust applications, and likewise, applications do not trust the OS. Whilst Esswood, the developer of CheriOS, has not framed their work in such a way, CheriOS could be characterised as an OS with the single purpose of managing TEEs (i.e., each application is considered a TEE).

Threat model

The work on CheriOS primarily focuses on software-layer security measures rather than hardware-layer defences. Consequently, certain types of threats, notably hardware-based attacks, fall outside the scope of the work. At the software level, the OS and applications are mutually distrusting, although there exists a small TCB — a nanokernel — which is inherently trusted. This mutual distrust extends only to assumptions about confidentiality and integrity; applications can not guarantee that the OS will not starve them of resources. The OS does not

reciprocate this trust towards applications. Correctness is assumed only for this nanokernel and the hardware (although specific external components, such as the compiler, are also presumed to be correct) [58].

Implementation

A nanokernel operates immediately above the hardware level, functioning as a hypervisor. This component is the only trusted component of the system, and as suggested by Esswood, it would likely be ‘delivered in firmware alongside CHERI CPUs’ [58]. The nanokernel’s responsibilities include managing *reservations*, which denote the authority to have allocated a particular region of memory (notably, not the authority to access said memory). This allows the OS to reference a memory region without accessing it, enabling mutual distrust. The reservation is then transferred to an application, where it is *taken*, or exchanged, by the nanokernel for an actual capability. In addition, the nanokernel provisions *foundations*, which are measured initial programme states for which the nanokernel guarantees the starting capability graph is disconnected from the remainder of the system’s capability graph [58].

The nanokernel guarantees that reservations correspond to regions of memory for which there is no existing capability (other than in the nanokernel itself). This is achieved in the following manner:

1. Upon revocation, the concerned region of the virtual address space is unmapped from the physical page within the memory management unit (MMU). These virtual addresses are not reused until the entire virtual address space becomes ‘dirty’. Subsequent attempts to access the physical memory using any remaining capabilities will result in a page fault.
2. When the entire virtual address space has been used, a sweep of the physical memory is performed to invalidate any remaining capabilities before the virtual address space is reused.

Through these measures, the nanokernel can guarantee that any allocated memory is unreferenced by dangling capabilities.

When exceptions occur, handlers within the CheriOS nanokernel are triggered. First, the execution context is saved, with a sealed capability used to reference it. Subsequently, a handler within the microkernel is invoked, which can use this sealed capability to reference the thread without being able to access the confidential programme state.

Discussion

CheriOS presents an innovative system closely aligned with the goals of this project. However, the specific requirements of trusted execution in a cloud environment are not central to its design. Consequently, the threat model lacks a thorough justification, and there is limited evaluation concerning the potential requirements of a cloud tenant. Furthermore, remote attestation — a crucial aspect of a cloud-based system — is not extensively discussed in the work.

3.1.4 CHERI-TrEE

CHERI-TrEE is a TEE implemented using (a possibly extended version of) CHERI, specifically designed for embedded systems which lack virtual memory [168]. Much like CheriOS (§ 3.1.3), the system’s architecture enables mutual distrust, but CHERI-TrEE is explicitly framed within the context of trusted execution. The proposed ISA extensions are formally specified in the Sail specification language [70], and the complete system is implemented on both an open-source RISC-V [170] processor extended with CHERI features, as well as on the Arm Morello platform.

Threat model

The CHERI-TrEE work focuses on the architectural and software levels rather than the micro-architectural level. Consequently, certain attack vectors are not considered, specifically those involving hardware attacks such as bus snooping. By proposing to extend CHERI, CHERI-TrEE can avoid needing a software TCB. Alternatively, a small hypervisor emulating these instructions may be used. As such, the threat model spans from only trusted hardware to a combination of trusted hardware and a small amount of trusted software, the latter model being comparable to CheriOS (§ 3.1.3). Like CheriOS, the mutual distrust between the OS and applications concerns confidentiality and integrity; applications cannot ensure that the OS will not starve them of resources. However, the OS makes no such assumptions about applications. Subject to this caveat, all trusted components are presumed to be correct, whilst the remainder of the software remains untrusted.

Implementation

Abstractly, CHERI-TrEE extends the ISA with an `EInit` instruction,² which takes a code and data capability, conducts a memory sweep to confirm unique ownership of each memory region, and seals the capabilities, ready to be invoked as an object-capability pair. Symbolic encryption and signing capabilities are also constructed for inter-TEE communication [168].

A trusted interrupt handler services exceptions, storing the thread’s registers in trusted memory. After clearing confidential registers, it invokes an untrusted handler and, upon completion, resumes execution of the thread [168].

Discussion

CHERI-TrEE targets low-end embedded devices, specifically those without virtual memory, resulting in a single-address space system. This design is probably insufficient for a multi-tenant cloud-based system. Furthermore, the threat model, particularly for the Arm Morello

²Whilst this section presents the implementation in terms of ISA extensions, this could alternatively be written as ‘CHERI-TrEE extends the trusted hypervisor with an `EInit` function’, reflecting instead the method of implementation on the Arm Morello platform.

implementation which has a software TCB, lacks a thorough justification. Lastly, the system does not currently support multi-threading, which is severely limiting.

3.2 TCB minimisation

Some systems enhance security without relying on specialised hardware. Instead, they minimise the size of their trusted computing base, consequently reducing the probability of an exploitable vulnerability in the trusted code. The following sections briefly outline two such systems, one of which is also CHERI-based (§ 3.1.3).

3.2.1 seL4

seL4 is a microkernel and hypervisor designed to have a small TCB.³ It has been proven functionally correct based on a formal specification written in higher-order logic. Additionally, it has been demonstrated that the resulting binary is a correct translation of the C code, providing assurance that the compiler has not introduced any malicious or unintentional vulnerabilities. Verified security properties include confidentiality and integrity, which prevent data from being accessed without explicit permission, and availability, guaranteeing that authorised access to a resource cannot be denied by other entities [75].

The formal guarantees provided for seL4 are based on the assumptions that the hardware behaves as expected, the theorem prover (Isabelle [138]) is correct, and the specification truly reflects the desired behaviour. Unfortunately, whilst the first two assumptions are reasonable, the latter is more challenging to justify and is the most likely source of potential weaknesses. Indeed, the current formal specification does not account for covert timing channels, which are commonly exploited in cache side-channel attacks [75].

3.2.2 The CHERIoT RTOS

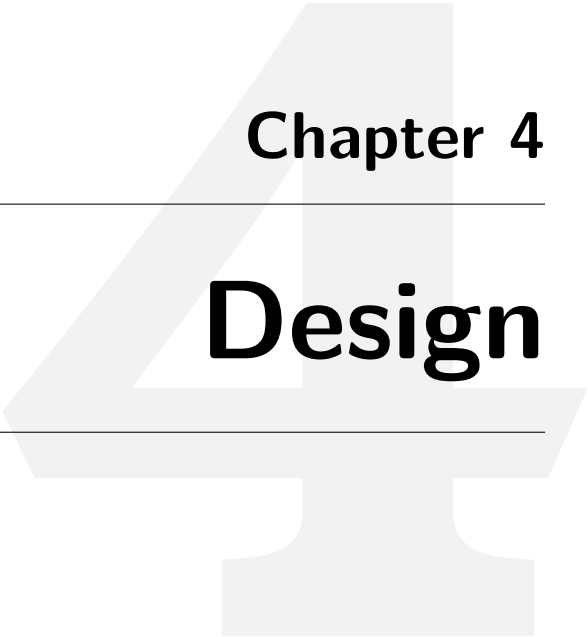
CHERI_{IoT} is an instantiation of CHERI extending RISC-V for small embedded cores, particularly Internet of things (IoT) platforms [3]. For the purposes of this discussion, it is not the architectural specification of CHERIoT that is of interest, but rather the real-time operating system (RTOS) that has been developed on top of it.

The RTOS comprises four components: a loader, a switcher, a heap allocator, and a scheduler. The loader derives capabilities from the root capability for use throughout the system. Upon completion of this task, it erases its own existence, including code, data, and registers, before handing control over to the scheduler. Employing a priority round-robin algorithm, the scheduler determines the execution order of threads, invoking the switcher to facilitate context switches. The switcher also handles interrupts and, alongside the loader, is one of only

³seL4 is also a capability-based system [75]; however, unlike CHERI, its capabilities are not implemented at the architectural level.

two components with access to a thread's register file and stack. Other components receive a sealed capability which can be used to reference a thread's register file, which is, of course, non-dereferencible [3].

Consequently, in the CHERIoT RTOS, the only fully trusted components are the loader — which is short-lived — and the switcher. This further reduces the size of the TCB.



Chapter 4

Design

Having provided a critical overview of various systems in the preceding chapter, I now present TERIOS, a cloud-based multi-tenant TEE platform designed for Arm Morello.

There are numerous strategies for enhancing the security of a system, and there is not necessarily a singular correct choice. However, the design of TERIOS is driven by two primary aims: (1) to operate on an Arm Morello board without any hardware modifications, fulfilling modest policy objectives; and (2) given some small hardware additions, be capable of achieving significantly more ambitious policy goals.

TERIOS consists of six small components partitioned along privilege boundaries to enable least-privilege compartmentalisation. Each component performs a particular duty, such as context switching, memory allocation, or scheduling, with the overall design intending to minimise the TCB. There is a strong emphasis on the justification of these design decisions, a detail that previous work has somewhat lacked. This reasoning is crucial because TERIOS's trustworthiness partially stems from a persuasive argument demonstrating its robust design.

To this end, the subsequent sections outline TERIOS's requirements (§ 4.1), define the threat model (§ 4.2), present the proposed design (§ 4.3), and conclude with a brief evaluation (§ 4.4).

4.1 Requirements

The following points broadly outline TERIOS's requirements:

- R.1) Have a clearly defined and justifiable threat model and design (§ 4.2 and § 4.3).
- R.2) Provide an execution environment that is more secure than traditional operating systems (e.g., Linux, MacOS, Windows), in line with the threat model (§ 4.2). More specifically:
 - R.2.1) The operation of TEEs should be opaque, prohibiting both observation and manipulation of data and computations by other TEEs and privileged code.
 - R.2.2) TEEs should be prevented from starving the system of resources.
 - R.2.3) Privileged code should only have access to the resources necessary for its operation.
 - R.2.4) The TCB should be minimised, facilitating the verification of functional correctness, either manually or through formal methods.
- R.3) Have a mechanism to remotely attest that TEEs are indeed executing on the anticipated system.
- R.4) Enable the operation of an arbitrary number of mutually distrusting TEEs (e.g., belonging to different cloud tenants) on a single board.
- R.5) In alignment with the overall objectives of this project (i.e., conducting a feasibility analysis), illustrate the extent to which CHERI and Morello boards are suitable for facilitating cloud-based TEEs.

4.2 Threat model

Many of the systems explored, such as Intel SGX (§ 3.1.1), had poorly documented threat models, necessitating an accumulation of information from various sources, including potentially outdated patents [49]. The vagueness surrounding their threat models exposes them to criticism when vulnerabilities emerge: critics might contend that they have compromised the system's security, despite the system never being designed to prevent such attacks. Therefore, the points below aim to clearly delineate TERIOS's threat model:

- Applications are deemed entirely untrustworthy. They might be compromised, contain vulnerabilities, or attempt to starve the system of resources.
- Each component of the privileged code is trusted by applications and other units of the privileged code to the minimum extent necessary to perform its required functions. Essentially, each component operates under the principle of least privilege [148]. For instance, if the scheduler becomes compromised, it may deprive a TEE of CPU time, but it cannot access the TEE's memory. This point is somewhat imprecise, and so the specific degree of trust required for each component is further elaborated upon in § 4.3.
- Microarchitectural attacks, including rowhammer [100] and cache side-channel attacks [101, 106], are acknowledged as critical security threats. However, they are regarded as outside

the scope of this system's threat model and left for resolution via other mechanisms.

- Attacks relying on physical access to the hardware are excluded from the threat model, with justifications presented in the following section (§ 4.2.1).

4.2.1 Exclusion of physical attacks

The following arguments (to be considered separately and cumulatively) are made to support the exclusion of physical attacks from TERIOS's threat model:

- As seen in the previous chapter, measures to mitigate attacks caused by physical access to hardware are not impenetrable. Whilst such preventative measures might be sufficient for digital rights management on devices located in consumer homes, given reasonable resources and expertise, it is possible to breach them. Large cloud providers, such as Microsoft Azure, Amazon Web Services, and the Google Cloud Platform, are operated by companies with a market capitalisation of almost five trillion USD [103]. These companies also have access to top-tier technical talent. As a result, they possess the capabilities to execute such attacks. Consequently, it is argued here that defending against a malicious and concerted effort by the cloud entity to breach these security measures — at least in their current form — is futile.
- Data centres operated by cloud providers are highly secure, with stringent access controls in place [6, 65, 113], making it challenging for a rogue employee to execute a physical attack. Even so, the potential scale of such an attack would be limited due to the vast number of geographically distributed servers managed by cloud operators. Furthermore, employees with physical access to the hardware, such as maintenance or security staff, are unlikely to possess knowledge of the specific hardware used by particular tenants. This limitation necessitates that any attack be untargeted, significantly reducing the incentive and usefulness of carrying one out. Given the limited scope, untargeted approach, and challenging nature of a physical attack, the cost-benefit of implementing countermeasures is unclear. Whilst the involvement of multiple employees could increase the impact of an attack, coordinating such an effort would be exceedingly difficult to keep covert. Additionally, distributed trust mechanisms within the cloud entity [53] provide some mitigation against such coordinated efforts.
- Arguably, the physical attack surface is smaller than the software attack surface. In practical terms, the likelihood of a non-malicious vulnerability emerging and compromising the system is significantly lower, even when only software attacks are included in the threat model. Executing large-scale malicious hardware attacks would likely require targeting the supply chain (e.g., during the design phase) [145], where the window of opportunity is relatively narrow. In contrast, comparable software attacks can be initiated over an indefinite period, presenting a more persistent threat.

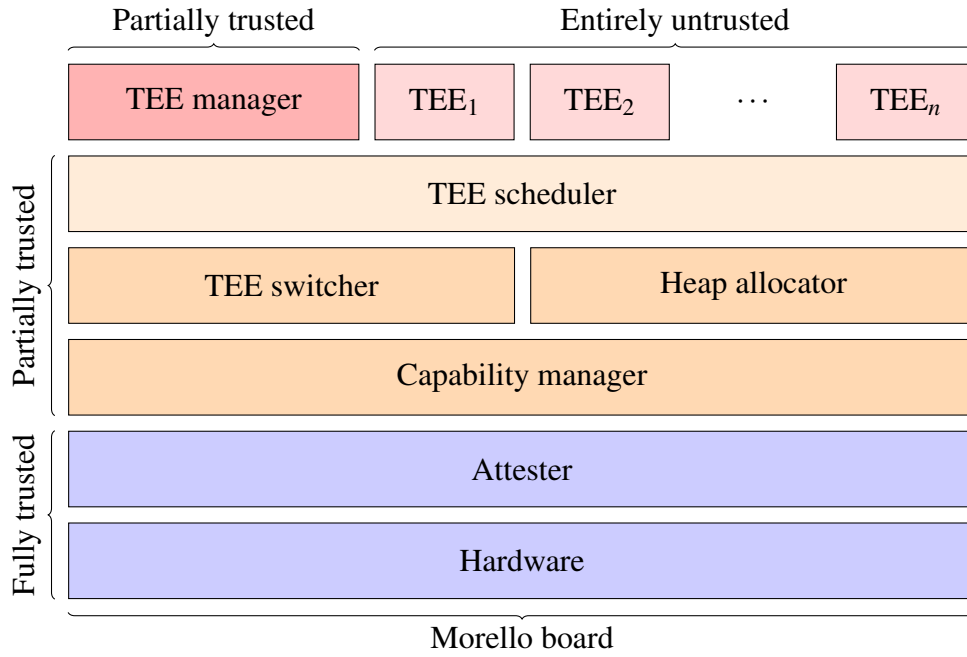


Figure 4.1: This figure illustrates the architecture of TERIOS. The *attester* issues an attestation proving the state of the booted system. Subsequently, the *capability manager* partitions the root capability and delegates access to the derived capabilities. The trusted execution environment (TEE) *switcher* manages the transitions between TEEs whilst also handling interrupts and routing them as necessary. A *heap allocator* is utilised to securely allocate memory, ensuring exclusive ownership and zeroing it upon release. The *scheduler* is responsible for determining the next TEE to execute, also allocating time to the manager. The *manager*, in turn, handles networked or local requests (e.g., from other TEEs) to create new TEEs, subject to resource policies. Colours loosely denote different levels of trust between components.

4.3 TERIOS architecture

This section describes the architecture of TERIOS, with each subsection devoted to a principal component, as outlined in Figure 4.1.

4.3.1 Attester

A significant issue during the design was the absence of a robust facility for remote attestation on the Morello hardware. Without this, how can users be sure that the secure system they believe they are engaging with is indeed the one in operation? One could argue that given the Morello hardware is experimental and not intended for production use [22], this issue can be disregarded. Indeed, it is proposed later that an effective solution would be the addition of hardware, such as a trusted platform module (TPM). Nonetheless, examining the scope of what could be achieved by utilising the available resources is interesting.

This proposal introduces a trusted software layer, the *attester*, which runs immediately after boot (or shortly thereafter, following firmware initialisation), as illustrated in Figure 4.2. This software is the only bootable entity, a restriction enforced by a read-only secure boot key installed by the original equipment manufacturer (OEM). Contained within this software is a

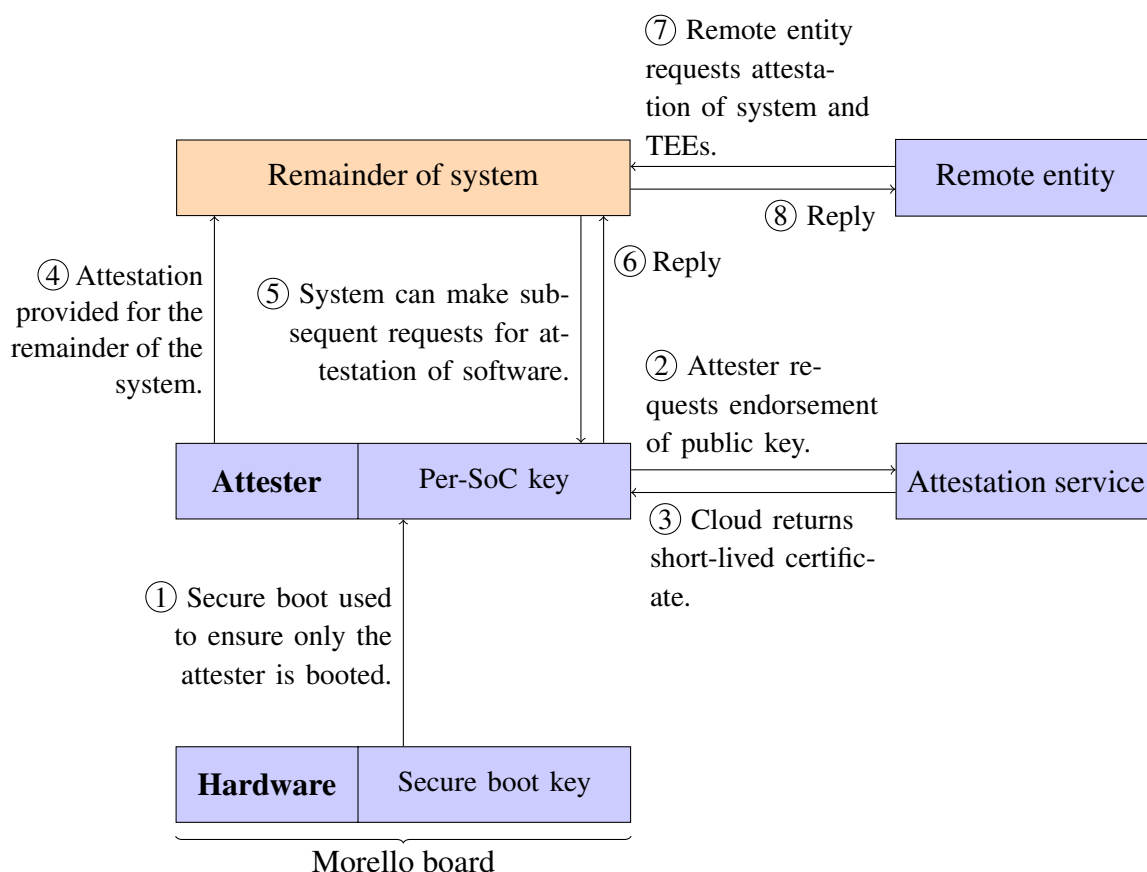


Figure 4.2: This figure illustrates a weak software-based attestation mechanism. The secure boot process guarantees that only the attester is booted, with each attester embedded with a unique key pair. The attester solicits a cloud service to endorse the public key, which replies with a short-lived signed certificate. The remainder of the system is booted and attested using the private key, enabling remote entities to verify validity using the attestation and certificate. Subsequent requests can be made for the attestation of other dynamically loaded software components. Some intermediate boot steps between the hardware and the attester have been omitted.

unique, per-device key pair. The public part of this key is sent to an OEM¹ cloud attestation service for endorsement, exchanged for a short-lived signed certificate. Such certificates are only issued if the public key resides on the attestation service's whitelist (added post-SoC manufacture). This key is removed from the whitelist if compromised (e.g., in the event of a cloud provider breach). The remainder of the system can further attest TEEs by either making down calls to the attester to endorse a particular hash, or by devising its own attestation service.² Remote entities can verify attestations using the signed certificate. This process accomplishes the following:

- The secure boot process guarantees the integrity of the attester, which is necessary given that the attester's role is to measure and attest the remainder of the system accurately. For example, if the attester were modified, it could falsely measure the system's state.
- If long-term certificates were pre-issued with the board, remote parties must check against a potentially extensive and sensitive blacklist to confirm the certificate's validity. Instead, by issuing certificates with a short expiry time, remote parties can be confident that the OEM attestation service remains satisfied that the board is secure,³ given it has recently issued a new certificate.
- The short expiry periods of the certificates help limit potential damage if exfiltrated.

Discussion

This mechanism has significant limitations. For instance, the cloud provider could extract and execute the attester on non-Morello boards or extract the keys from the attester to obtain a certificate from the attestation service. Despite the approaches to disincentivise and respond to such actions, it remains a technically weak solution.

A more robust alternative requires the addition of a TPM to the hardware [23, 142]. The TPM needs to be identifiable and integrated into the SoC, as it must attest not only to the software's state but also that the software is indeed on a Morello board. A TPM that is not integrated into the SoC could be attached to any device, thus compromising the attestation of the hardware. This rules out any attestation methods involving off-SoC devices connected via the network, USB, PCI, or the FPGA links.

The TPM serves the role of both a root of trust for reporting and storage. However, an additional root of trust for measurement (RTM) is required, necessitating at least a microcode modification to the CPU [57]. As depicted in Figure 4.3, the TPM, in conjunction with the RTM, facilitates

¹The attestation service provider does not necessarily need to be the OEM; it could be any trusted third party.

²Given that the state of the remainder of the system is attested and, as such, guaranteed to behave in the expected way (e.g., correctly obtain a TEE's hash), one may question why the hash itself cannot be used to attest the state of a TEE. The challenge lies in the potential for a man-in-the-middle attack to falsely return such a hash to the requester. The signed attestation proves that the information originated from the claimed system.

³This determination can be made through various means, including on-site inspection of hardware or compliance with contractual agreements. Certain poor behaviour patterns from a board owner (e.g., a cloud provider) might give reasons for the OEM to stop issuing them with attestation certificates.

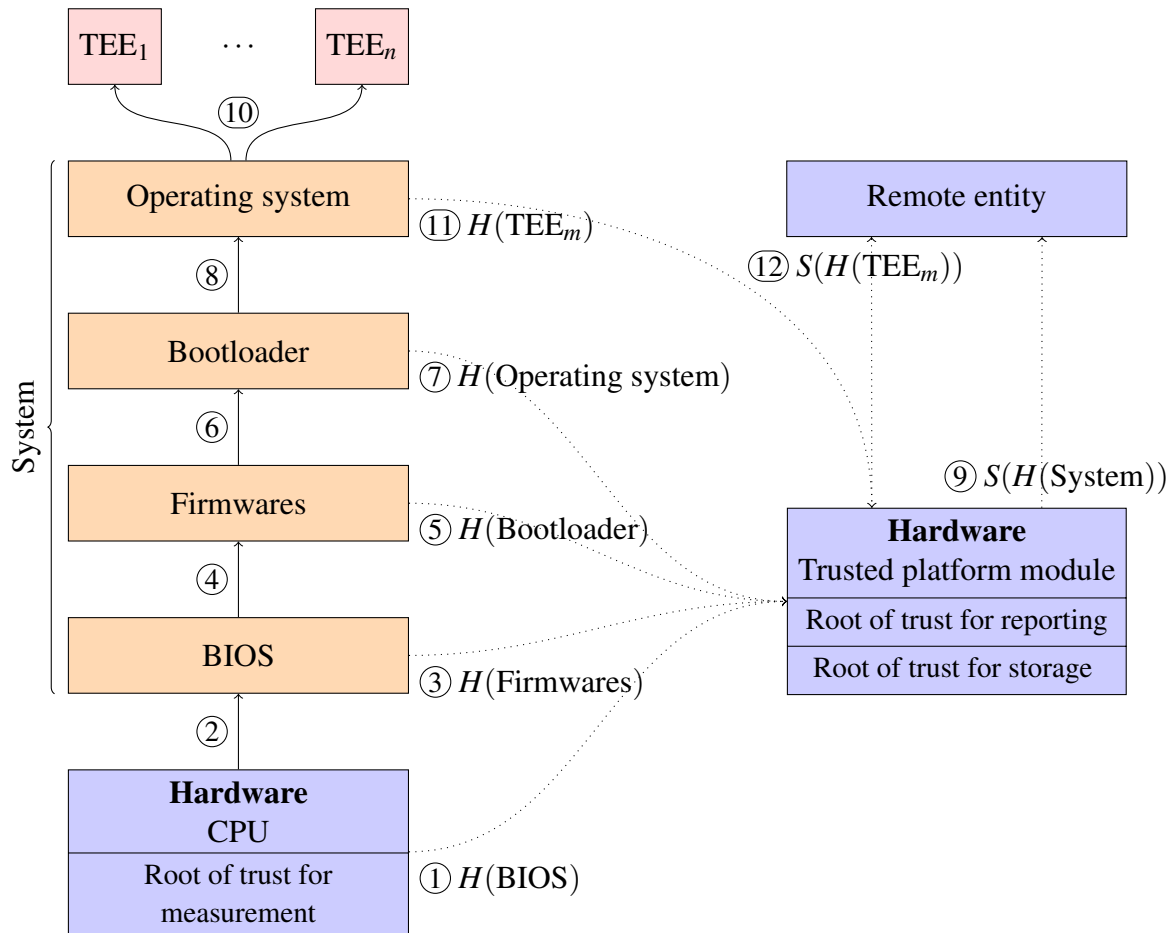


Figure 4.3: This figure depicts a hardware-based attestation mechanism. Started by the root of trust for measurement, each software component is measured using a hash function (H). These measurements are stored in read-only registers within the trusted platform module (TPM), which is the root of trust for storage and reporting. Through the operating system, a remote party can request these measurements and an associated signature (S). The operating system can provide further attestations for dynamically loaded software by measuring said software and obtaining a signature from the TPM. Remote entities can verify this information by comparing the hash to a known good value and verifying the signature.

an attested, measured boot process.

However, there remain limitations. The RTM only measures the initial software layer, after which each subsequent layer measures the next. Consequently, for a remote entity to independently validate an attestation, it would need to inspect the source code of each layer, compile it with an identical configuration to produce the same binary, and then perform a measurement. This process is impractical and fragile, particularly considering some components may be proprietary. The more probable solution is to depend on a trusted third party which has obtained and published known good values of the measurements. Further mitigation of this trust dependency trends towards an Intel-SGX-like solution, and it is not immediately clear how CHERI can be adjusted to facilitate this without significant hardware modifications. It is thus concluded that the TPM addition marks the upper bound of what would be attainable within the policy spectrum enabled under the Morello board mechanism.

4.3.2 Capability manager

The Morello board boots with a root capability covering the entire virtual address space. The role of the *capability manager* is to partition this capability and drop access to any unneeded resources before delegating access to the derived capabilities. Proper compartmentalisation practices ensure that capability access is granted solely to the intended units of code. Dropping any unneeded capabilities (e.g., to unused peripherals) further improves the system's security. Additionally, this component of TERIOS ensures that all memory, which may contain confidential information left in memory from a sudden reboot, is zeroed.

The capability manager must be entirely trusted, given that it maintains a capability to the entire address space. Alternatively, it could exist ephemerally: delegating capabilities before self-destructing (i.e., zeroing itself). Although this component would still demand trust, the trust level would be reduced by shrinking the attack window. This was the approach taken by the CHERI_{IoT} RTOS (§ 3.2.2). Instead, TERIOS permits unlimited calls to the capability manager, allowing other components to discard the delegated capability when not in use, further reinforcing the principle of least privilege. This is a calculated tradeoff within the design.

4.3.3 Heap allocator

The *heap allocator* securely allocates memory, ensuring exclusive ownership and zeroing it upon release. Like CheriOS (§ 3.1.3), it must employ a strategy to safeguard against unauthorised access from retained capabilities in released memory. Specifically, virtual addresses should not be reused until the entire address space is expended. After this, a memory sweep is required to zero any dangling capabilities (taking care also to check register files).

Potential optimisations exploiting characteristics of cloud applications might be possible. Since the pages mapped to a TEE's address space are tracked, on TEE termination, the remaining pages could be zeroed. Any regions of the virtual address space previously used by the TEE can

then be reused, with the guarantee that no dangling capabilities remain. As many TEEs might be short-lived, address space reuse in this manner is likely achievable, potentially eliminating the need for any slow [168] memory sweeps.

4.3.4 TEE switcher

Handling transitions between TEEs is the responsibility of the *TEE switcher*. This component is highly trusted, as it interacts directly with the confidential registers of TEEs, saving them to memory before restoring the execution context of the subsequent TEE scheduled to run. As well as using CHERI for TEE memory isolation, traditional mechanisms should be utilised, such as unmapping physical pages associated with other TEEs from the virtual address space.

This component also handles interrupts and exceptions, responding accordingly, for example, by executing a context switch. Furthermore, it routes these interrupts (or queues the resultant events) to the relevant components and TEEs as required.

4.3.5 TEE scheduler

The *TEE scheduler*, invoked either by the TEE switcher on a timer interrupt or at the end of a TEE's execution, determines the next TEE to execute (or opts to continue executing the current TEE). It also must schedule the execution of the TEE manager (§ 4.3.6).

This design adopts a simple round-robin scheduling approach, but more intricate mechanisms integrating priorities could be considered. This is a standard scheduling problem; thus, existing solutions used in cloud computing could be applied.

4.3.6 TEE manager

The *TEE manager* manages networked and local requests relating to the creation and termination of TEEs. It also monitors execution time and resource usage to facilitate accurate billing. In addition, the manager can authorise or deny the creation of new TEEs based on predefined policies (e.g., a particular tenant may have reached their resource allocation limit).

The TEE manager maintains a degree of trust in that it can halt existing TEEs and refuse to create new ones. However, it cannot access any confidential code or data within a TEE.

This component and any other part of the cloud provider's network or billing infrastructure could gather side-channel information by monitoring the volume of ingress and egress traffic per TEE. Correlating these could expose sensitive information. A possible solution is to make TERIOS data oblivious [130], for example, by padding TEE outputs to a fixed length or a function of the input size (over a given unit of time) [81]. A similar concern extends to processing-time channels, particularly with short-lived TEEs. This issue can be mitigated similarly by quantising the processing time [81].

4.3.7 Execution environment

A significant design decision relates to the execution environment inside a TEE. A custom, minimal system call interface could be exposed, allowing a variety of richer environments to target the TEE, such as WebAssembly [74] or various library operating systems [34, 140, 167]. Alternatively, such an environment could act as the primary interface. This design proposes the former for flexibility and simplicity, anticipating that the latter systems can be ported to this interface. The specific interface is not delineated here; however, a small subset of POSIX [85] is proposed for preliminary validation of the design.

It is important to highlight that all applications execute within a TEE; there is no non-secure environment. Thus, the system can be framed as a secure co-processor accessible over the network. However, given that TEEs can spawn child TEEs,⁴ they can define varying degrees of internal distrust, effectively recreating the boundary that exists between non-secure and secure execution environments. These child TEEs are implemented and behave entirely as conventional TEEs, the only difference being that they are owned by another TEE (and thus can be terminated only by that TEE or its respective ancestors).

4.4 Evaluation

TERIOS reasonably fulfils the initial requirements, with a complete evaluation with respect to these provided in Table 4.1.

This work has not considered any formal verification methods during the design phase. However, the minimal functionality makes TERIOS amenable to manual functional verification. Furthermore, given the run-time assurances provided by CHERI, a compelling case can be made for omitting costly formal verification methods entirely.

The most significant limitation lies in the attestation properties, which are severely constrained. Even with improvements possible through minor hardware modifications, it remains challenging to argue that an Intel-SGX-like solution would not be more desirable.

Despite falling outside the threat model of TERIOS, the potential for micro-architectural attacks must be taken into account before production use. Current research efforts are examining how to handle cache side-channel attacks in the context of CHERI [61, 173]. Furthermore, cloud operators already employ techniques to mitigate attacks such as rowhammer [112], which could be integrated into TERIOS.

A potential design approach not explored in this work involves the use of linear capabilities, which permit only a single reference to an object. This could enable further elements of the privileged code to be distrusted (e.g., the heap allocator once the capability for the allocated memory has been passed to the TEE). However, I did not pursue this idea further, considering

⁴Ideally, these child TEEs should reside locally, but could conceivably be transparently offloaded to a remote machine, subject to the parent's preferences.

that linear capabilities are not a part of mainline CHERI or Morello. Nevertheless, preliminary investigations have been carried out by Lippeveldts [107] and Watson et al. [174, § D.7].

Lastly, it would be beneficial to consider modifications to TERIOS that align it with the GlobalPlatform TEE protection profile specification [63], yielding a standardised design.

Table 4.1: An evaluation of TERIOS with respect to the original requirements (§ 4.1).

Requirement	Achieved?	Justification
R.1	✓	The threat model and design are outlined and justified in § 4.2 and § 4.3, respectively.
R.2	✓	Fulfilled due to the successful accomplishment of the majority of the sub-requirements.
R.2.1	Partially	Whilst TEEs remain concealed from most privileged code, with only the switcher, heap allocator, and capability manager able to view confidential data and code, they are not entirely opaque. This partial opaqueness stems from hardware limitations, with no mechanism for hardware-facilitated context switches or memory allocation.
R.2.2	✓	The TEE manager, heap allocator, and scheduler work in combination to prevent TEEs from starving TERIOS of resources.
R.2.3	✓	The privileged code is designed to be compartmentalised according to the principle of least privilege. TERIOS has been modularised along privilege boundaries to facilitate this.
R.2.4	✓	TERIOS is designed to be small, featuring a minimised system call interface, with additional functionality delegated to a non-privileged library OS.
R.3	Partially	The proposed attestation solution, which does not require hardware modifications, is vulnerable to several attacks from a dishonest cloud provider. Minor hardware modifications can enhance the attestation robustness, although verification challenges persist, especially with proprietary software in the stack. An additional trusted third party is likely to be required.
R.4	✓	Ignoring resource exhaustion, TERIOS supports an arbitrary number of TEEs from multiple tenants, with CPU and memory resources fairly allocated by the switcher, scheduler, and heap allocator.
R.5	✓	TERIOS demonstrates the extent to which CHERI and Morrello boards can be used to facilitate cloud-based TEEs.

Chapter 5

Prototype implementation

With the design outlined, this chapter describes the implementation of TERIOS’s experimental prototype.¹ The prototype served as a tool to validate design concepts, guiding necessary adjustments and exposing practical challenges. In addition, it enabled experimentation with the design. The primary purpose was not to produce a production-ready system but to ensure that the design was grounded in the architectural reality of the Morello SoC and demonstrate hardware limitations.

The prototype provides an illustrative example of a baremetal nanokernel operating across EL0, EL1, and EL2, scheduling multiple untrusted tasks² which all share CPU time. Context switches are initiated by a timer interrupt, routed to the exception handler by the GIC. Exceptions are also used to implement system calls from EL0, which invoke the nanokernel executing at EL1.

In this section, I begin by describing the comprehensive development environment constructed for this project (§ 5.1). Following this, I discuss the specifics of the implementation (§ 5.2) before providing a brief evaluation (§ 5.3).

¹The associated software artefact and documentation can be found in Appendix A.

²Each task can be viewed as a limited TEE.

5.1 Development environment

This project involved the development of baremetal code, which following the execution of low-level firmwares, interacts directly with the hardware without an underlying operating system. Establishing such a development environment was challenging and time-consuming, requiring a cross-compilation toolchain, a non-host target simulator, and a compatible debugger. Although documentation was available, it was often fragmented, outdated, and incomplete. Consequently, considerable reverse engineering and trial-and-error was necessary. This section provides further details in relation to this.³

5.1.1 Morello software stack

Firmwares

The Morello software stack comprises various firmwares, operating systems, and tools. The user guide [121] supplied the necessary information for downloading and building this software. First, host dependencies were installed, and a script was used to download the latest source code. The initial download took several hours and resulted in 130 GB of content; however, I subsequently learnt that much of this, including the Android, Ubuntu, Debian, and Busy-Box distributions, was unnecessary for the project's requirements. Following this, downloaded programmes were executed to verify the installation of all required dependencies and to build the software stack.

The build process was carried out using the `./build-scripts/build-all.sh` script, with the `-f none` option specified to build only the packages and firmware components required for baremetal execution. These components include the system control processor (SCP) and manageability control processor (MCP) firmwares, responsible for tasks such as power management, clock management, and voltage regulation [24]. Depending on the use case, Trusted Firmware-A (TF-A) may also be required, which is also available as part of the provided software stack. TF-A is a reference implementation of secure world software [26] and can be embedded with custom payloads [120]. Its usage is explored later in this section.

Morello fixed virtual platform

At this stage, the only way to execute baremetal software would be directly on a Morello SoC. This presents two issues. First, I have only remote (i.e., SSH) access to a Morello SoC running CheriBSD. Second, even with physical access to a board, using it for development would be inefficient. An alternative is to use Arm's fixed virtual platform (FVP), which virtualises the Morello SoC on a non-target host, such as x86-64.⁴ Download and installation of the FVP is

³The description provided in this section is non-exhaustive. For more detailed information, refer to the `README.md` file in the accompanying software artefact (Appendix A).

⁴Development was carried out on an x86-64 system equipped with a quad-core Intel i7 1.90 GHz processor, 16 GB of RAM, and running under the Ubuntu 20.04.1 LTS operating system.

Table 5.1: Options to be passed to the assembler, C compiler, and linker for various compilation modes.

	<code>--target</code>	<code>--march</code>	<code>--mabi</code>
Non-Morello instructions	<code>aarch64-none-elf</code>	Unset	Unset
Hybrid-capability mode	<code>aarch64-none-elf</code>	<code>morello+a64c</code>	Unset
Pure-capability mode	<code>aarch64-none-elf</code>	<code>morello+c64</code>	<code>purecap</code>

trivial, following the instructions provided by Arm [17].

5.1.2 Code compilation and execution

Toolchain

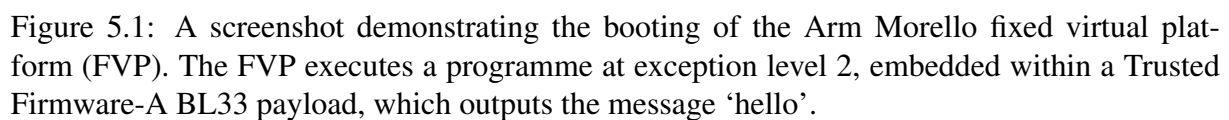
To compile code for baremetal execution, the baremetal toolchain is required. Obtaining this is a simple process that involves downloading the pre-built binaries [99] and extracting them from the archive file. The toolchain binaries, including the `clang` compiler, are located in the `bin` directory. This toolchain can be used in the standard manner, applying the options specified in Table 5.1.

There are two primary options for baremetal execution: the first is post-SCP firmware boot at EL3, and the second is running as a TF-A BL33 payload at EL2 [120]. Both options are feasible when executing within the FVP. However, additional board setup is necessary for the SoC, which is performed by TF-A [29, 48]. Utilising TF-A avoids the need to duplicate this configuration and ensures that the resulting system is portable to an actual SoC rather than being restricted to the FVP.

C standard library

Using the C standard library (`libc`) can be advantageous for many reasons. The baremetal toolchain includes a version of Newlib [123], a C standard library adapted for baremetal Morello. However, by default, it assumes execution starts at EL3, making it incompatible as part of a TF-A payload. Modifications were made to Newlib in November 2022 to enable EL2 execution [30]. However, the documentation indicates that the toolchain (and presumably also Newlib) cannot currently be built from source [32], and no pre-built binaries with these changes were accessible during development.

Given the importance of access to the C standard library — particularly for performing dynamic relocation — I made the decision to execute post-SCP firmware boot at EL3, sacrificing portability to the SoC. However, in late March 2023, binaries containing the modified version of Newlib were made available. Consequently, I ported the existing work to execute at EL2 as a TF-A payload. By providing the `--config morello-baremetal-purecap-el2.cfg` option to the linker, the relevant EL2 components of Newlib (e.g., `crt0.o` and EL2 CPU initialisation) are linked.

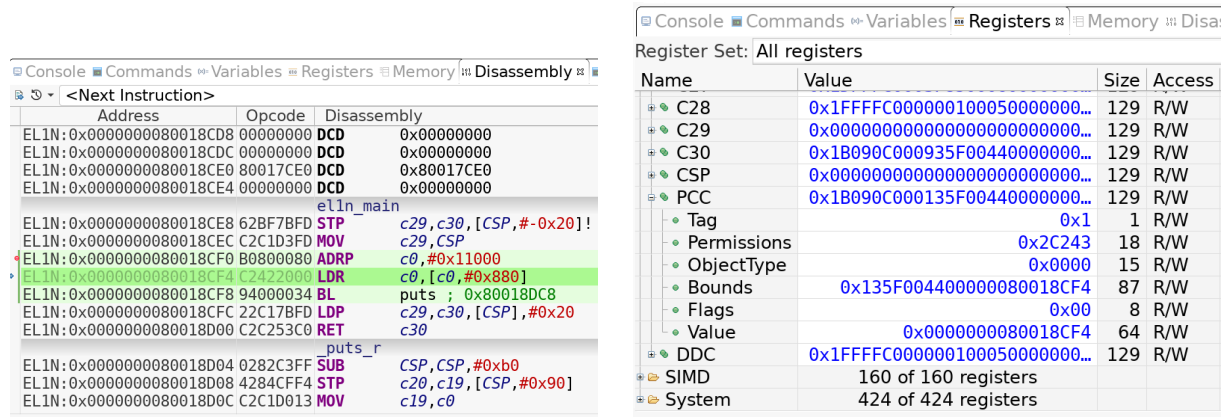


Executing within the FVP

For post-SCP firmware boot at EL3, the binary is ready to be loaded into the FVP. If it is necessary to embed the binary within the TF-A firmware, the `make -C "bsp/arm-tf"` command can be utilised. The resulting image is located at `bsp/arm-tf/build/morello/release/fip.bin` and is ready to be loaded into the FVP. Figure 5.1 demonstrates the booted FVP running a simple programme.

5.1.3 Arm Development Studio

After installing the development studio, it needs to be configured to work with the Morello



(a) The disassembler allows for step-by-step analysis of code running on the Morello fixed virtual platform (FVP).

(b) During execution, registers can be inspected, with the fields of capability registers being displayed separately for added convenience.

Figure 5.2: The Arm Development Studio allows binaries to be compiled and loaded onto the FVP, enabling efficient debugging through breakpoints and register inspection.

toolchain to compile projects. The final step involves connecting the debugger to the FVP. To accomplish this, the FVP is started with the `--cadi-server` option, running a programme in an infinite loop (known as branch-to-self). The debugger can then connect to the FVP and load the compiled binary. Subsequently, registers can be inspected during execution, the binary disassembled, and instructions stepped through, as illustrated in Figure 5.2.

When debugging, an issue I encountered was the loss of debug symbol information when changing exception levels, which led to issues such as non-functional assembly-to-source-code translation and disappearing labels in the disassembly. After a thorough investigation, I discovered that the debugger discards debug information for an unknown reason when the exception level changes. This was resolved by utilising the debugger's `add-symbol-file` command to reload the debug information from the debug binary.

Throughout development, exceptions frequently occurred. The `ESR_ELx` and `ELR_ELx` registers (where `x` corresponds to the current exception level) proved to be particularly valuable and could be analysed in the debugger following an exception. The `ESR_ELx` register stores error information, detailing the precise cause of an exception, whilst the `ELR_ELx` register holds the address of the instruction that triggered the exception.

5.2 Implementation details

The TERIOS prototype comprises 1113 lines⁵ of assembly and C code, which I have programmed. As much as possible is in C, avoiding the need for manual and tedious construction of capabilities in assembly.

As linear and detailed descriptions of code are often unilluminating and unengaging, the com-

⁵Calculated using `find src -name "*.c" -o -name "*.h" -o -name "*.S" | xargs wc -l`.

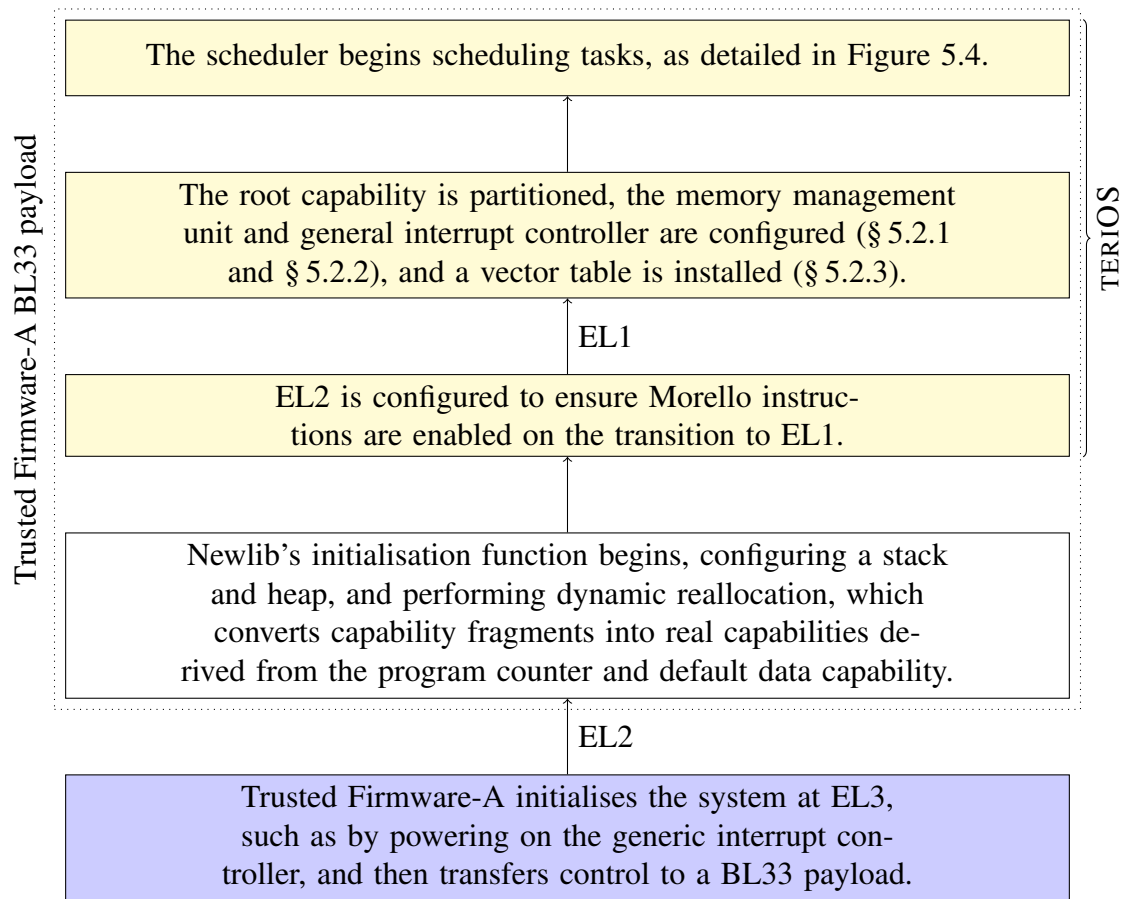


Figure 5.3: An outline of the principal components of the prototype.

mented source code (Appendix A) is deferred to for a detailed understanding.⁶ A broad overview of the implementation is illustrated in Figure 5.3, whilst the remainder of this section outlines any particularly interesting aspects and challenges I encountered during development.

5.2.1 Memory management unit

Ideally, the default flat mapping from the virtual address space to the physical address space would be utilised for simplicity during the prototyping phase. However, due to the application of the Device-nGnRnE memory attribute on this mapping [11, R_{WFZPW}], unaligned accesses trigger an alignment fault [11, § G5.11.4]. This presents a challenge, as the pre-compiled `libc` is not configured to disable unaligned accesses.

Consequently, the MMU must be configured. The 1 GB region starting at 0x0 — which includes essential peripherals such as the UART and the general interrupt controller — and the 2 GB of DRAM commencing at 0x80000000 are mapped; virtual and physical addresses are equal. A challenge arises, however, as this memory needs to be accessible at EL0. Even with ‘privileged access never’ (PAN) [11, § A2.4.1] disabled, memory concurrently mapped to both EL0 and EL1 is forcibly marked non-executable at EL1 by the hardware.

⁶For a quick overview without delving into the details, refer to `src/el1n/{configure_mmu.c,configure_gic.c,scheduler.c,timer_interrupt_handler.S}`.

To overcome this, the first gigabyte of DRAM starting at 0x80000000 is halved using a level two table, with the first segment mapped to EL1 and the second to EL0. A link script is used to ensure all EL0 code is allocated to sections beginning at 0xA0000000. The second gigabyte of DRAM, starting at 0xC0000000, is shared between EL0 and EL1, but this is used exclusively for the stack, so non-executability at EL1 is acceptable. Capabilities are utilised to maintain separation.

5.2.2 Generic interrupt controller

For timer interrupts to be enabled, the GIC must be configured to route relevant exceptions. Trusted Firmware-A handles the power-up of the GIC, simplifying this configuration task. The Morello SoC is equipped with the GIC-600, a version not backwards compatible with GICv2 [14, §, 3.9], which implies that affinity routing is always active [14, §, 3.9]. Note that because execution occurs in a non-secure state, the mapping of the distributor control register differs slightly [14, §, 3.9].

The system only requires interrupts from the EL1 physical timer, identified by an interrupt ID of 30 [9, §, 3.4]. The redistributor's offset for software-generated interrupts is obtained [18, §, 12.10] and the 30th bit [19, §, 5.3.1] of `IGROUPR0`, `IGRPMODR0`, and `ISENABLER0` is set. This configures EL1 physical timer interrupts to be routed to group one by the redistributor. These interrupts are handled in the next section (§ 5.2.3).

5.2.3 Exception handling and context switching

A vector table is initialised to handle both synchronous and interrupt request (IRQ) exceptions. Its address is placed into the system register `CVBAR_EL1`. These IRQ exceptions, triggered by the EL1 timer, signal that a context switch is necessary, which is performed as outlined in Figure 5.4.

Synchronous exceptions, which can be triggered for numerous reasons, are disregarded except when caused by the `SVC` instruction, which initiates a supervisor call [11, §, C6.2.365]. The lower exception level assigns an ID, which facilitates the routing of the `SVC` to the appropriate handler. Currently, a single `SVC` has been implemented, enabling printing to the UART.

The hardware automatically disables nested interrupts upon an exception, raising all bits of the `DAIF` exception mask [15, §, 10.5]. Upon performing an `ERET`, this mask is cleared, thus re-enabling interrupts. As a consequence, interrupt handlers need not be reentrant. This is crucial, given that functions such as `puts`, used in `SVC` handlers, also lack reentrancy.

5.2.4 Dynamic relocations

One of the most challenging issues involved acquiring an unsealed capability to executable code across distinct compilation units. The dynamic relocation performed by Newlib seals executable capabilities [124]. These sentries can be invoked using instructions such as `BL` but cannot be

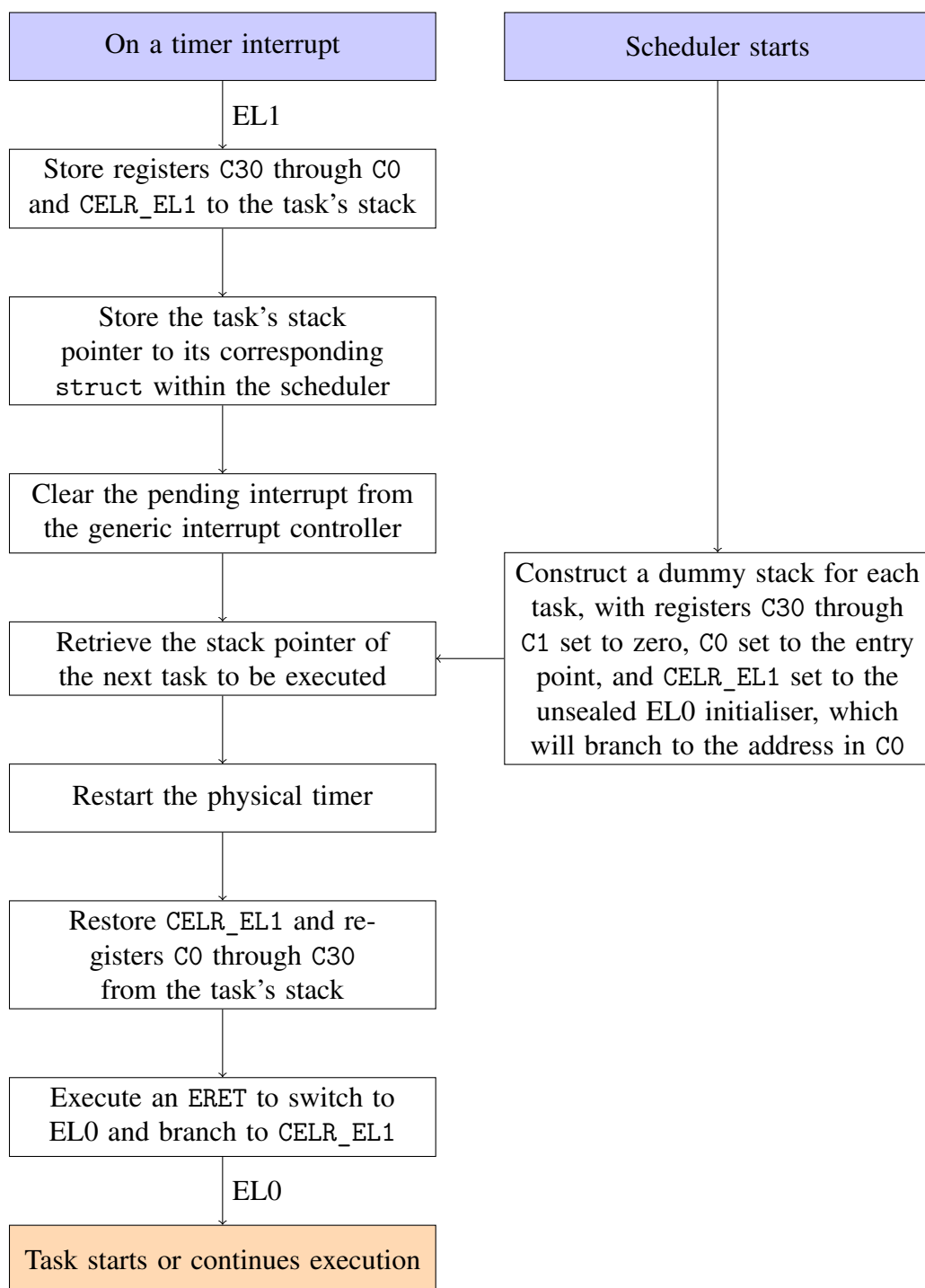


Figure 5.4: An illustration of a context switch initiated by either a timer interrupt or the scheduler being started.

```

1  .global el1n_get_init
2  .type   el1n_get_init, "function"
3  el1n_get_init:
4      adr c0, el1n_init
5      ret
6
7  el1n_init:
8      bl  el1n_main
9      b   .

```

Listing 5.1: An example illustrating a global function returning an address to a local function. Taken from `src/el1n/init.S` (Appendix A).

```

1  static void* get_el0n_init(void *unseal_sentry) {
2      void *init = cheri_unseal(&el0n_init, unseal_sentry);
3
4      uintptr_t init_addr = cheri_address_get(init);
5
6      // Align address to a 4-byte boundary.
7      init_addr = (init_addr + 3) & ~((uintptr_t) 3);
8
9      init = cheri_address_set(init, init_addr);
10
11     return init;
12 }

```

Listing 5.2: A demonstration of the process required to unseal and realign a sentry. Taken from `src/el1n/scheduler.c` (Appendix A).

used within the exception link registers or the vector table base register, as when needed, the capability is copied directly to the program counter, which must be unsealed.

In most instances, the solution is a function that returns an address a few instructions ahead within its own compilation unit, as illustrated in Listing 5.1. However, this strategy fails when retrieving the EL0 entry, as the required code must reside in the `0xA0000000` segment, thus rendering it non-executable from EL1. The workaround involves manually unsealing the capability, as demonstrated in Listing 5.2.⁷

This approach, however, introduces a secondary issue: the address used for sentry invocations is misaligned by a byte, and the capability cannot be realigned downward due to its bounds. Consequently, alignment must occur upward to the next 4-byte boundary. This results in the

⁷Retrospectively, I wonder if forcing the `el0n_init` code into its own section using a link script and then obtaining the start address of this section at link time for use in the exception link register may have been a more desirable approach.

Figure 5.5: A screenshot showcasing the FVP executing TERIOS, with the interleaved output from tasks displayed. Despite the shortest possible timer interval, hundreds of messages are printed during each task’s CPU time slice. Hence, the screenshot can only display the output from two tasks. Task 1 was started before task 3 (and indeed, there is undisplayed output from task 1 previously in the console).

function’s first instruction being skipped, which should, therefore, always be a NOP.

5.3 Evaluation

The prototype, when compared to the design, is of a relatively narrower scope. It is not intended to be production-ready and should be considered experimental. It was designed to meet requirements R.2.3, R.2.4, and R.4, whilst also demonstrating the reasons behind the partial fulfilment of R.2.1. As for the non-functional requirements, it satisfies R.1, R.2, and R.5. Further work is required to implement the software attestation outlined in § 4.3.1 and specified in R.3.

The TERIOS prototype was evaluated using three statically-linked tasks, each emitting a unique message 500 times. This is displayed in Figure 5.5. The interleaved outputs from these tasks demonstrate that the CPU is shared between them and that the SVC call is functioning correctly. Various attempts to access unpermitted regions of memory resulted in capability faults, as expected. Further analysis was limited, as outlined in the following limitations and areas of future work:

- This evaluation has been brief and predominantly qualitative, focusing on how well the prototype aligned with the initial goals. The evaluation did not make comparisons with numerical objectives, such as throughput, latency, or memory overhead. This approach was necessary due to the prototype’s limited scope and also partially due to work being carried out on the FVP rather than the SoC, making performance benchmarks unrepresentative. Future work should prioritise obtaining numerical data, such as the maximum

number of TEEs each board can accommodate, the amount of memory each TEE requires, and whether performance diminishes as the number of TEEs increases.

- The current experimental prototype highlights the hardware limitations that make satisfying R.2.1 particularly challenging. The implementation provides a foundation for future efforts to build upon, but substantial additional work remains to implement TERIOS. Until such work is completed, the design of TERIOS remains only partially validated. Specifically, the following areas could be of interest:
 - Develop the software attestation described in § 4.3.1.
 - Implement the orchestrator, as detailed in § 4.3.6, to facilitate dynamic task loading.
 - Expand the SVC calls to facilitate porting a library operating system to execute within a task (as discussed in § 4.3.7). For instance, a crucial SVC to develop is one to allocate heap memory, which would subsequently necessitate the implementation of the heap allocator (§ 4.3.3).
- The use of Newlib increases the TCB. Investigating if the functionality being used from Newlib can be refactored into a standalone, smaller library would be beneficial.
- Additional testing on the Morello SoC is necessary, as the TERIOS prototype has been exclusively evaluated on the FVP.
- Further efforts should explore the integration of Arm TrustZone features into the implementation, assessing the prospective security improvements it may offer.

In conclusion, the TERIOS prototype provides a substantial foundation for future work, though it still requires additional refinement and development to validate the design.

Chapter 6

Conclusions

Using the hypothesis from § 1 as a foundation, this investigation aimed to see if CHERI’s memory safety attributes could be leveraged to provide robust security properties. This approach is feasible, albeit with certain constraints related to the current Morello hardware.

In this chapter, reflections about the work are shared (§ 6.1), limitations of the project and potential future directions of work are discussed (§ 6.2), and conclusions are drawn (§ 6.3).

6.1 Reflections

Submitted in November 2022, the original project proposal — encompassing a subset of CAMB’s objectives — aimed to implement cloud-based enclaves using CHERI. At that time, it was unclear to the CAMB researchers and me that CHERI was not well-suited for this purpose. Consequently, the project’s direction shifted significantly, focusing instead on the trusted execution policy space enabled by CHERI.

Working within a nascent project environment has exposed me to the uncertainties and ambiguities inherent in the early stages of research. Coupled with the need to demonstrate progress for the purposes of assessment, this has been a challenging project, but I have gained invaluable research experience.

6.2 Limitations and future work

Evaluations, discussions, and limitations relating to the design and prototype have been explored previously in § 4.4 and § 5.3, respectively. The points below focus on the overarching limitations of this project:

- Whilst an extensive review of several systems was provided in § 3, it does not qualify as a methodological literature review. Several additional works, such as Haven [33], InkTag [80], Flicker [109], Sego [102], Overshadow [44], MiniBox [104], and CAP-VMs [149] would be invaluable additions, but could not be included due to space constraints.
- Complementing this project with market research could provide valuable insights. It would be intriguing to understand the priorities of business and IT leaders, their opinions on the proposed design of TERIOS, and their sentiments towards the range of existing TEEs. Furthermore, exploring which aspects of TEEs appeal to them most would be beneficial. For example, are they more drawn to systems with distributed trust, such as Microsoft Azure in combination with Intel SGX, or do they find a single trusted entity like AWS's Nitro sufficient? Are they attracted to objective and rational security enhancements, or does marketing rhetoric which gives the perception of heightened security hold equal importance? Until such work is completed, the commercial viability of the design remains unvalidated.

6.3 Final remarks

This project started by thoroughly examining key concepts relating to trust and trusted execution, specifically within the context of cloud computing. This philosophical discussion exposed the complexities involved in understanding why consumers place their trust in various entities. Following this, the project analysed a range of similar systems, all of which aimed to improve security. Building on this analysis, TERIOS, a CHERI-based system was presented, with an emphasis on the justification of design decisions and a precise threat model. In particular, a robust argument for the exclusion of hardware-based attacks from the threat model was presented. The baremetal, single-address-space, pure-capability prototype of TERIOS is a solid foundation for such a system. Furthermore, it facilitated the distillation and authoring of up-to-date documentation for configuring a baremetal Morello debugging and development environment.

There are good arguments for reapplying CHERI to facilitate TEEs. Moreover, when implemented in a cloud setting, there are persuasive reasons for relaxing the threat model concerning hardware attacks. Yet, the trusted execution security assurances CHERI offers are limited, particularly when combined with the restricted facilities for remote attestation on the Morello hardware. Achieving more robust policy objectives (e.g., R.2.1 and R.3) might necessitate extending CHERI or integrating existing Arm hardware into the Morello boards. However, the

feasibility and performance impact of these enhancements remains uncertain without further development and prototyping. For instance, does the high performance of today's computers require multiple specialised technologies, each optimised for specific use cases, rather than a 'one-size-fits-all' mechanism? If so, attempting to position CHERI as a universal solution to diverse problems might inhibit its excellence in memory safety.

In summary, CHERI holds potential for use within cloud-based TEEs, possibly as part of an enhanced Arm board. However, further work beyond this project on prototyping and possible hardware extensions is required.

Bibliography

- [1] Fritz Alder et al. ‘Faulty Point Unit: ABI Poisoning Attacks on Trusted Execution Environments’. In: *Digital Threats* 3.2 (Feb. 2022).
- [2] Abdulaziz Aljabre. ‘Cloud Computing for Increased Business Value’. In: *International Journal of Business and Social Science* 3.1 (Jan. 2012).
- [3] Saar Amar et al. *CHERIoT: Rethinking security for low-cost embedded systems*. Tech. rep. MSR-TR-2023-6. Microsoft, Feb. 2023.
- [4] Amazon Web Services. *AWS Nitro Enclaves*. URL: <https://aws.amazon.com/ec2/nitro/nitro-enclaves/> (visited on 17/04/2023).
- [5] Amazon Web Services. *AWS Nitro System*. URL: <https://aws.amazon.com/ec2/nitro/> (visited on 17/04/2023).
- [6] Amazon Web Services. *Perimeter Layer*. URL: <https://aws.amazon.com/compliance/data-center/perimeter-layer/> (visited on 08/05/2023).
- [7] Ittai Anati et al. *Innovative Technology for CPU Based Attestation and Sealing*. Tech. rep. 659614. Intel Corp, 14th Aug. 2013.
- [8] Sigurd Frej Joel Jørgensen Ankergård, Edlira Dushku and Nicola Dragoni. ‘State-of-the-Art Software-Based Remote Attestation: Opportunities and Open Issues for Internet of Things’. In: *Sensors* 21.5 (Feb. 2021), p. 1598.
- [9] Arm. *AArch64 Programmer’s Guides Generic Timer*. Manual ARM062-1010708621-30. 13th Aug. 2019.
- [10] Arm. *AMBA AXI Protocol*. Specification ARM IHI 0022. Mar. 2023.
- [11] Arm. *Arm Architecture Reference Manual for A-profile architecture*. Manual. 19th Aug. 2022.
- [12] Arm. *Arm Architecture Reference Manual Supplement Morello for A-profile Architecture*. Manual. 17th Jan. 2022.
- [13] Arm. *ARM Compiler toolchain Developing Software for ARM Processors. What is semi-hosting?* 29th Feb. 2012. URL: <https://developer.arm.com/documentation/dui0471/g/Bgbjjgij> (visited on 28/04/2023).
- [14] Arm. *Arm CoreLink GIC-600 Generic Interrupt Controller*. Manual 100336_0106_00_en. 8th Feb. 2019.
- [15] Arm. *ARM Cortex-A Series. Programmers Guide for ARMv8-A*. Tech. rep. ARM DEN0024A ID050815. 24th Mar. 2015.

- [16] Arm. *Arm Development Studio Morello Edition*. 18th Jan. 2022. URL: <https://developer.arm.com/downloads/-/morello-development-tools-downloads> (visited on 22/03/2023).
- [17] Arm. *Arm Ecosystem FVPs*. URL: <https://developer.arm.com/downloads/-/arm-ecosystem-fvps> (visited on 09/02/2023).
- [18] Arm. *Arm Generic Interrupt Controller Architecture Specification. GIC architecture version 3 and version 4*. Specification Arm IHI 0069H ID020922. Jan. 2022.
- [19] Arm. *Arm Morello System Development Platform*. Manual 102278_0001_04_en. 15th Apr. 2022.
- [20] Arm. *ARM Security Technology. Building a Secure System using TrustZone Technology*. Tech. rep. Apr. 2009.
- [21] Arm. *Learn the architecture — Introducing the Arm architecture*. Tech. rep. 102404_0201_01_en. 23rd Feb. 2023.
- [22] Arm. *Morello Prototype Architecture Overview. Summary and next steps*. URL: <https://developer.arm.com/documentation/den0133/0100/Summary-and-next-steps> (visited on 21/05/2023).
- [23] Arm. *Platform Security Boot Guide*. Tech. rep. DEN 0072. 30th July 2020.
- [24] Arm. *SCP-firmware - version 2.12*. URL: <https://github.com/ARM-software/SCP-firmware> (visited on 22/04/2023).
- [25] Arm. *SMC Calling Convention*. Tech. rep. ARM DEN 0028E. May 2022.
- [26] Arm. *Trusted Firmware-A Documentation*. URL: <https://trustedfirmware-a.readthedocs.io/en/latest/> (visited on 12/04/2023).
- [27] Sergei Arnautov et al. ‘SCONE: Secure Linux Containers with Intel SGX’. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 689–703.
- [28] Todd M. Austin, Scott E. Breach and Gurindar S. Sohi. ‘Efficient Detection of All Pointer and Array Access Errors’. In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. PLDI ’94*. Orlando, Florida, USA: Association for Computing Machinery, 1994, pp. 290–301.
- [29] Silviu Baranga. Slack message. 10th June 2022. URL: https://cheri-cpu.slack.com/archives/C012UPAE86Q/p1654874495295649?thread_ts=1654843775.157279 (visited on 22/04/2023).
- [30] Silviu Baranga. *[libgloss] Add support for bootcode at EL2*. 16th Nov. 2022. URL: <https://git.morello-project.org/morello/newlib/-/commit/0f8400db8a40fc7133b2a768ef19faab1af14ea3> (visited on 03/04/2023).
- [31] Silviu Baranga. *[Morello] Add initial support for not using semihosting*. 18th Nov. 2022. URL: <https://git.morello-project.org/morello/newlib/-/commit/07db1bab7976e809bb92854df85618f3d850292> (visited on 04/04/2023).
- [32] Silviu Baranga. *LLVM toolchain with Morello support*. Arm. URL: <https://git.morello-project.org/morello/docs/-/blob/morello/mainline/toolchain-readme.rst> (visited on 15/02/2023).
- [33] Andrew Baumann, Marcus Peinado and Galen Hunt. ‘Shielding Applications from an Untrusted Cloud with Haven’. In: *ACM Trans. Comput. Syst.* 33.3 (Aug. 2015).

- [34] Andrew Baumann et al. ‘Composing OS Extensions Safely and Efficiently with Bascule’. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys ’13. Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 239–252.
- [35] Ken Beer and Ryan Holland. *Securing Data at Rest with Encryption*. Tech. rep. Amazon Web Services, Nov. 2013.
- [36] T. Berger. ‘Analysis of current VPN technologies’. In: *First International Conference on Availability, Reliability and Security (ARES’06)*. 2006.
- [37] Nick Bhadange. *Confidential Computing in Health Care*. URL: <https://www.intel.com/content/dam/www/central-libraries/us/en/documents/digital-transformation-confidential-computing-in-healthcare.pdf> (visited on 17/04/2023).
- [38] Henk Birkholz et al. *Remote ATtestation procedureS (RATS) Architecture*. RFC 9334. Jan. 2023.
- [39] Ferdinand Brasser et al. ‘Software Grand Exposure: SGX Cache Attacks Are Practical’. In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, Aug. 2017.
- [40] Stefan Brenner et al. ‘SecureKeeper: Confidential ZooKeeper Using Intel SGX’. In: *Proceedings of the 17th International Middleware Conference*. Middleware ’16. Trento, Italy: Association for Computing Machinery, 2016.
- [41] David Brown. *Confidential computing: an AWS perspective*. Amazon Web Services. 24th Aug. 2021. URL: <https://aws.amazon.com/blogs/security/confidential-computing-an-aws-perspective/> (visited on 08/02/2023).
- [42] Eoghan Casey and Gerasimos J. Stellatos. ‘The Impact of Full Disk Encryption on Digital Forensics’. In: *SIGOPS Oper. Syst. Rev.* 42.3 (Apr. 2008), pp. 93–98.
- [43] Stephen Checkoway and Hovav Shacham. ‘Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface’. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 253–264.
- [44] Xiaoxin Chen et al. ‘Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems’. In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 2–13.
- [45] Zitai Chen et al. ‘VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface’. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 699–716.
- [46] B.V. Chess. ‘Improving computer security using extended static checking’. In: *Proceedings 2002 IEEE Symposium on Security and Privacy*. 2002, pp. 160–173.
- [47] Yung Ryn Choe et al. ‘BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments.’ In: (Dec. 2016).
- [48] Jessica Clarke. Slack message. 15th Dec. 2022. URL: https://cheri-cpu.slack.com/archives/C012UPAE86Q/p16711132812777149?thread_ts=1671120342.195449 (visited on 22/04/2023).

- [49] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Tech. rep. Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2016.
- [50] C. Cowan et al. ‘Buffer overflows: attacks and defenses for the vulnerability of the decade’. In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX’00*. Vol. 2. 2000, 119–129 vol.2.
- [51] Jon Crowcroft. ‘Threats to SGX’. Unpublished.
- [52] Fergus Dall et al. ‘Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks’. In: (2018).
- [53] Emma Dauterman et al. ‘Reflections on Trusting Distributed Trust’. In: *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. HotNets ’22. Austin, Texas: Association for Computing Machinery, 2022, pp. 38–45.
- [54] Kevin Deus et al. *Data Driven Security Hardening in Android*. Google Security Blog. 29th Jan. 2021. URL: <https://security.googleblog.com/2021/01/data-driven-security-hardening-in.html> (visited on 12/11/2022).
- [55] Roger Dingledine, Nick Mathewson and Paul Syverson. ‘Tor: The Second-Generation Onion Router’. In: *13th USENIX Security Symposium (USENIX Security 04)*. San Diego, CA: USENIX Association, Aug. 2004.
- [56] Tu Dinh Ngoc et al. ‘Everything You Should Know About Intel SGX Performance on Virtualized Systems’. In: *Proc. ACM Meas. Anal. Comput. Syst.* 3.1 (Mar. 2019).
- [57] ebrary. *Platform Boot Integrity*. URL: https://ebrary.net/24529/computer-science/platform_boot_integrity.
- [58] Lawrence G. Esswood. *CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor*. Tech. rep. UCAM-CL-TR-961. University of Cambridge, Computer Laboratory, Sept. 2021.
- [59] Everest Group. *Confidential Computing — The Next Frontier in Data Security*. Oct. 2021. URL: https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/Everest_Group_-_Confidential_Computing_-_The_Next_Frontier_in_Data_Security_-_2021-10-19.pdf (visited on 08/02/2023).
- [60] Shufan Fei et al. ‘Security Vulnerabilities of SGX and Countermeasures: A Survey’. In: *ACM Comput. Surv.* 54.6 (July 2021).
- [61] Franz A. Fuchs et al. ‘CHERI Compartments. Toward Transient-Execution Attack Mitigations on CHERI Compartments’. CHERI Technical Workshop 2023. 31st Mar. 2023.
- [62] Craig Gentry. ‘A Fully Homomorphic Encryption Scheme’. AAI3382729. PhD thesis. Stanford, CA, USA, 2009.
- [63] GlobalPlatform Device Committee. *TEE Protection Profile*. Tech. rep. GPD_SPE_021. Nov. 2016.
- [64] John Goodacre. *Why A Fundamental Change in Cybersecurity Is Required*. Info Security Group. 16th Feb. 2022. URL: <https://www.infosecurity-magazine.com/opinions/fundamental-change-cybersecurity/> (visited on 13/11/2022).
- [65] Google. *Data and Security*. URL: <https://www.google.com/about/datacenters/data-security/> (visited on 08/05/2023).
- [66] Google. *HTTPS encryption on the web*. URL: <https://transparencyreport.google.com/https/overview> (visited on 29/03/2023).
- [67] Google Cloud. *Confidential Computing*. URL: <https://cloud.google.com/confidential-computing> (visited on 17/04/2023).

- [68] Kevin C. Gotze, Gregory M. Iovino and Jiangtao Li. ‘Secure provisioning of secret keys during integrated circuit manufacturing’. U.S. pat. 9742563. Intel Corp. 22nd Aug. 2017.
- [69] Kevin C. Gotze, Jiangtao Li and Gregory M. Iovino. ‘Fuse attestation to secure the provisioning of secret keys during integrated circuit manufacturing’. U.S. pat. 8885819. Intel Corp. 11th Nov. 2014.
- [70] Kathryn E. Gray et al. *The Sail instruction-set semantics specification language*. Tech. rep. University of Cambridge, 15th Mar. 2017.
- [71] Richard Grisenthwaite. ‘Arm Morello Evaluation Platform — Validating CHERI-based Security in a High-performance System’. In: *2022 IEEE Hot Chips 34 Symposium (HCS)*. 2022.
- [72] Le Guan et al. ‘TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone’. In: *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys ’17. Niagara Falls, New York, USA: Association for Computing Machinery, 2017, pp. 488–501.
- [73] Roberto Guanciale et al. ‘Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures’. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 38–55.
- [74] Andreas Haas et al. ‘Bringing the Web up to Speed with WebAssembly’. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 185–200.
- [75] Gernot Heiser. *The seL4 Microkernel. An Introduction*. Whitepaper. 10th June 2020.
- [76] Scott Helme. *Alexa Top 1 Million Analysis — February 2019*. 11th Mar. 2019. URL: <https://scotthelme.co.uk/alexa-top-1-million-analysis-february-2019/> (visited on 29/03/2023).
- [77] Thomas A. Hemphill and Phil Longstreet. ‘Financial data breaches in the U.S. retail economy: Restoring confidence in information technology security standards’. In: *Technology in Society* 44 (2016), pp. 30–38.
- [78] Kevin John Henry. ‘The Theory and Applications of Homomorphic Cryptography’. MA thesis. University of Waterloo, 2008.
- [79] Matthew Hoekstra et al. ‘Using Innovative Instructions to Create Trustworthy Software Solutions’. In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP ’13. Tel-Aviv, Israel: Association for Computing Machinery, 2013.
- [80] Owen S. Hofmann et al. ‘InkTag: Secure Applications on an Untrusted Operating System’. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 265–278.
- [81] Tyler Hunt et al. ‘Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data’. In: *ACM Trans. Comput. Syst.* 35.4 (Dec. 2018).
- [82] IBM. *Confidential computing for total privacy assurance*. URL: <https://www.ibm.com/cloud/smarterpapers/confidential-computing-for-total-privacy-assurance/> (visited on 17/04/2023).
- [83] IBM. *Confidential computing on IBM Cloud*. URL: <https://www.ibm.com/uk-en/cloud/confidential-computing> (visited on 17/04/2023).

- [84] IBM. *What is confidential computing?* URL: <https://www.ibm.com/topics/confidential-computing> (visited on 07/02/2023).
- [85] ‘IEEE Standard for Information Technology — Portable Operating System Interface (POSIX). Base Specifications, Issue 7’. In: *IEEE*. IEEE Std 1003.1-2017 (2018).
- [86] Intel. *11th Generation Intel Core Processor Desktop*. Datasheet. Mar. 2021.
- [87] Intel. *12th Generation Intel Core Processors*. Datasheet. Oct. 2021.
- [88] Intel. *Exception Handling in Intel Software Guard Extensions (Intel SGX) Applications*. White Paper. 2017.
- [89] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Manual 325462-079US. Mar. 2023.
- [90] Intel. *Intel Software Guard Extensions (Intel SGX)*. 332680-002. June 2015.
- [91] Intel. *OP-TEE* for Intel Architecture*. URL: <https://www.intel.com/content/www/us/en/developer/topic-technology/open/op-tee/overview.html> (visited on 09/05/2023).
- [92] *ISO/IEC 27001:2022(en) Information security, cybersecurity and privacy protection — Information security management systems — Requirements*. Standard. International Organization for Standardization, Oct. 2022.
- [93] *ISO/IEC 27017:2015(en) Information technology — Security techniques — Code of practice for information security controls based on ISO/IEC 27002 for cloud services*. Standard. International Organization for Standardization, Dec. 2015.
- [94] *ISO/IEC 27018:2019(en) Information technology — Security techniques — Code of practice for protection of personally identifiable information (PII) in public clouds acting as PII processors*. Standard. International Organization for Standardization, Jan. 2019.
- [95] Yeongjin Jang et al. ‘SGX-Bomb: Locking Down the Processor via Rowhammer Attack’. In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. SysTEX’17. Shanghai, China: Association for Computing Machinery, 2017.
- [96] Xinyu Jin et al. ‘Unpredictable Software-based Attestation Solution for node compromise detection in mobile WSN’. In: *2010 IEEE Globecom Workshops*. 2010, pp. 2059–2064.
- [97] Alexandre Joannou et al. ‘Efficient Tagged Memory’. In: *2017 IEEE International Conference on Computer Design (ICCD)*. 2017, pp. 641–648.
- [98] Simon P. Johnson et al. ‘Technique for supporting multiple secure enclaves’. U.S. pat. 9904632. Intel Corp. 27th Feb. 2018.
- [99] Yury Khrustalev. *LLVM Toolchain for Morello*. Arm. URL: <https://git.morello-project.org/morello/llvm-project-releases> (visited on 15/02/2023).
- [100] Yoongu Kim et al. ‘Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors’. In: *Proceeding of the 41st Annual International Symposium on Computer Architecture*. ISCA ’14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 361–372.
- [101] Paul Kocher et al. ‘Spectre Attacks: Exploiting Speculative Execution’. In: *Commun. ACM* 63.7 (June 2020), pp. 93–101.

- [102] Youngjin Kwon et al. ‘Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services’. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’16. Atlanta, Georgia, USA: Association for Computing Machinery, 2016, pp. 277–290.
- [103] *Largest Companies by Market Cap*. URL: <https://companiesmarketcap.com/> (visited on 08/05/2023).
- [104] Yanlin Li et al. ‘MiniBox: A Two-Way Sandbox for x86 Native Code’. In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC’14. Philadelphia, PA: USENIX Association, 2014, pp. 409–420.
- [105] Moritz Lipp et al. ‘ARMageddon: Cache Attacks on Mobile Devices’. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 549–564.
- [106] Moritz Lipp et al. ‘Meltdown: Reading kernel memory from user space’. In: *Communications of the ACM* 63.6 (2020), pp. 46–56.
- [107] Aaron Lippeveldts. ‘Linear Capabilities for CHERI: An Exploration of the Design Space’. In: *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH Companion 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 47–48.
- [108] Andrew Martin. *The ten-page introduction to Trusted Computing*. Tech. rep. CS-RR-08-11. Oxford University Computing Laboratory, Nov. 2008.
- [109] Jonathan M. McCune et al. ‘Flicker: An Execution Infrastructure for Tcb Minimization’. In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. Eurosys ’08. Glasgow, Scotland UK: Association for Computing Machinery, 2008, pp. 315–328.
- [110] Francis X. McKeen et al. ‘Method and apparatus to provide secure application execution’. U.S. pat. 10885202. Intel Corp. 5th Jan. 2021.
- [111] Frank McKeen et al. ‘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. Tel-Aviv, Israel: Association for Computing Machinery, 2013.
- [112] Paul McLellan. *HOT CHIPS: The AWS Nitro Project*. 2nd Oct. 2019. URL: https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/the-aws-nitro-project (visited on 01/05/2023).
- [113] Microsoft. *Azure facilities, premises, and physical security*. 13th Mar. 2023. URL: <https://learn.microsoft.com/en-us/azure/security/fundamentals/physical-security> (visited on 08/05/2023).
- [114] Microsoft Azure. *Azure confidential computing*. URL: <https://azure.microsoft.com/en-gb/solutions/confidential-compute/> (visited on 12/11/2022).
- [115] Microsoft Azure. *Confidential computing on a healthcare platform*. URL: <https://learn.microsoft.com/en-gb/azure/architecture/example-scenario/confidential/healthcare-inference> (visited on 17/04/2023).
- [116] Matt Miller. *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape*. BlueHat IL 2019. Microsoft Security Response Center (MSRC), 7th Feb. 2019.

- [117] Ahmad Moghimi, Gorka Irazoqui and Thomas Eisenbarth. ‘Cachezoom: How SGX amplifies the power of cache attacks’. In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2017, pp. 69–90.
- [118] Carlos Molina-Jimenez. *CAMB (Cloud Attestables on Morello Boards)*. 2022. URL: <https://www.cl.cam.ac.uk/research/srg/projects/camb/> (visited on 30/03/2023).
- [119] Ciara Moore et al. ‘Practical homomorphic encryption: A survey’. In: *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2014, pp. 2792–2795.
- [120] Morello Project. *Running a standalone baremetal application*. URL: <https://git.morello-project.org/morello/docs/-/blob/morello/mainline/standalone-baremetal-readme.rst> (visited on 08/02/2023).
- [121] Morello Project. *User Guide*. URL: <https://git.morello-project.org/morello/docs/-/blob/morello/mainline/user-guide.rst> (visited on 08/02/2023).
- [122] Kit Murdock et al. ‘Plundervolt: Software-based Fault Injection Attacks against Intel SGX’. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1466–1482.
- [123] *Newlib*. URL: <https://git.morello-project.org/morello/newlib> (visited on 19/03/2023).
- [124] *Newlib. relocs.c*. URL: <https://git.morello-project.org/morello/newlib/-/blob/morello/master/libgloss/aarch64/relocs.c>.
- [125] Daniel Newman et al. *Confidential Computing: The Future of Data Security and Digital Trust*. Futurum Research. Oct. 2021. URL: <https://www.ibm.com/downloads/cas/M5LDGM8L> (visited on 17/04/2023).
- [126] Bernard Ngabonziza et al. ‘TrustZone Explained: Architectural Features and Use Cases’. In: *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. 2016, pp. 445–451.
- [127] Kyndylan Nienhuis et al. ‘Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process’. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1003–1020.
- [128] Robert M. Norton. *Hardware support for compartmentalisation*. Tech. rep. UCAM-CL-TR-887. University of Cambridge, Computer Laboratory, May 2016.
- [129] Ralph O’Brien. ‘Privacy and security: The new European data protection regulation and it’s data breach notification requirements’. In: *Business Information Review* 33.2 (2016), pp. 81–84.
- [130] Olga Ohrimenko et al. ‘Oblivious Multi-Party Machine Learning on Trusted Processors’. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 619–636.
- [131] *OP-TEE Documentation. About OP-TEE*. URL: <https://optee.readthedocs.io/en/latest/index.html> (visited on 09/05/2023).
- [132] *OP-TEE Documentation. Cryptographic implementation*. URL: https://optee.readthedocs.io/en/latest/architecture/secure_storage.html (visited on 09/05/2023).
- [133] *OP-TEE Documentation. Secure storage*. URL: <https://optee.readthedocs.io/en/latest/architecture/crypto.html> (visited on 09/05/2023).
- [134] *OP-TEE Documentation. Core*. URL: <https://optee.readthedocs.io/en/latest/architecture/core.html> (visited on 09/05/2023).

- [135] Taejoon Park and K.G. Shin. ‘Soft tamper-proofing via program integrity verification in wireless sensor networks’. In: *IEEE Transactions on Mobile Computing* 4.3 (2005), pp. 297–309.
- [136] Namrata Patel, Parita Oza and Smita Agrawal. ‘Homomorphic Cryptography and Its Applications in Various Domains’. In: *International Conference on Innovative Computing and Communications*. Ed. by Siddhartha Bhattacharyya et al. Singapore: Springer Singapore, 2019, pp. 269–278.
- [137] David A. Patterson. ‘Reduced Instruction Set Computers’. In: *Commun. ACM* 28.1 (Jan. 1985), pp. 8–21.
- [138] L.C. Paulson. *Isabelle: A Generic Theorem Prover*. Lecture Notes in Computer Science. Springer, 1994.
- [139] Sandro Pinto and Nuno Santos. ‘Demystifying Arm TrustZone: A Comprehensive Survey’. In: *ACM Comput. Surv.* 51.6 (Jan. 2019).
- [140] Donald E. Porter et al. ‘Rethinking the Library OS from the Top Down’. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 291–304.
- [141] Ling Qian et al. ‘Cloud Computing: An Overview’. In: *Cloud Computing*. Ed. by Martin Gilje Jaatun, Gansen Zhao and Chunming Rong. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 626–631.
- [142] Miguel Miranda Quaresma. ‘TrustZone based Attestation in Secure Runtime Verification for Embedded Systems’. MA thesis. July 2020.
- [143] Arvind Raghu and J D Bean. *Confidential computing with AWS compute*. AWS re:Invent 2022. 2nd Dec. 2022.
- [144] Anil Rao. *Rising to the Challenge — Data Security with Intel Confidential Computing*. 20th Jan. 2022. URL: <https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141> (visited on 27/04/2023).
- [145] M. Rostami et al. ‘Hardware security: Threat models and metrics’. In: *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2013, pp. 819–823.
- [146] Mohamed Sabt, Mohammed Achemlal and Abdelmadjid Bouabdallah. ‘Trusted Execution Environment: What It is, and What It is Not’. In: *2015 IEEE Trustcom/Big-DataSE/ISPA*. Vol. 1. 2015, pp. 57–64.
- [147] Hukum Saini, Abhay Upadhyaya and Manish Kumar Khandelwal. ‘Benefits of Cloud Computing for Business Enterprises: A Review’. In: *IO: Productivity* (2019).
- [148] J.H. Saltzer and M.D. Schroeder. ‘The Protection of Information in Computer Systems’. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308.
- [149] Vasily A. Sartakov et al. ‘CAP-VMs: Capability-Based Isolation and Sharing in the Cloud’. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 597–612.
- [150] Michael Schwarz et al. ‘Malware guard extension: Using SGX to conceal cache attacks’. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2017, pp. 3–24.
- [151] Konstantin Serebryany et al. ‘AddressSanitizer: A Fast Address Sanity Checker’. In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 309–318.

- [152] A. Seshadri et al. ‘SWATT: softWare-based attestation for embedded devices’. In: *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004.* 2004, pp. 272–282.
- [153] Arvind Seshadri et al. ‘Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems’. In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles. SOSP ’05.* Brighton, United Kingdom: Association for Computing Machinery, 2005, pp. 1–16.
- [154] SGX 101. *Attestation*. URL: <https://sgx101.gitbook.io/sgx101/sgx-bootstrap/attestation> (visited on 25/02/2023).
- [155] Muhammad Yasir Shabir et al. ‘Analysis of classical encryption techniques in cloud computing’. In: *Tsinghua Science and Technology* 21.1 (2016), pp. 102–113.
- [156] Mark Shaneck et al. ‘Remote Software-Based Attestation for Wireless Sensors’. In: *Security and Privacy in Ad-hoc and Sensor Networks*. Ed. by Refik Molva, Gene Tsudik and Dirk Westhoff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 27–41.
- [157] R. Shetty et al. ‘HeapMon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection’. In: *IBM Journal of Research and Development* 50.2.3 (2006), pp. 261–275.
- [158] Diomidis Spinellis. ‘Reflection as a Mechanism for Software Integrity Verification’. In: *ACM Trans. Inf. Syst. Secur.* 3.1 (Feb. 2000), pp. 51–62.
- [159] W. Stallings. ‘Reduced instruction set computer architecture’. In: *Proceedings of the IEEE* 76.1 (1988), pp. 38–55.
- [160] Nik Sultana, Deborah Shands and Vinod Yegneswaran. ‘A Case for Remote Attestation in Programmable Dataplanes’. In: *Proceedings of the 21st ACM Workshop on Hot Topics in Networks. HotNets ’22.* Austin, Texas: Association for Computing Machinery, 2022, pp. 122–129.
- [161] F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft professional. Microsoft Press, 2004.
- [162] Synergy Research Group. *Cloud Spending Growth Rate Slows But Q4 Still Up By \$10 Billion from 2021; Microsoft Gains Market Share*. 6th Feb. 2023. URL: <https://www.srgresearch.com/articles/cloud-spending-growth-rate-slows-but-q4-still-up-by-10-billion-from-2021-microsoft-gains-market-share> (visited on 29/03/2023).
- [163] The Chromium Projects. *Memory safety*. 2020. URL: <https://www.chromium.org/Home/chromium-security/memory-safety/> (visited on 12/11/2022).
- [164] The Confidential Computing Consortium. *A Technical Analysis of Confidential Computing*. Tech. rep. v1.3. Nov. 2022.
- [165] The Confidential Computing Consortium. *Common Terminology for Confidential Computing*. Tech. rep. Dec. 2022.
- [166] Chia-che Tsai, Donald E. Porter and Mona Vij. ‘Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX’. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 645–658.
- [167] Chia-Che Tsai et al. ‘Cooperation and Security Isolation of Library OSes for Multi-Process Applications’. In: *Proceedings of the Ninth European Conference on Computer Systems. EuroSys ’14.* Amsterdam, The Netherlands: Association for Computing Machinery, 2014.

- [168] Thomas Van Strydonck. ‘Formal Reasoning about Hardware Capability Architectures’. Devriese, Dominique (supervisor), Piessens, Frank (cosupervisor). PhD thesis. Distributed, Secure Software (DistriNet), Leuven (Arenberg), Faculty of Engineering Science, Science, Engineering and Technology Group, June 2022.
- [169] W3Techs. *Usage statistics of Default protocol https for websites*. URL: <https://w3techs.com/technologies/details/ce-httpsdefault> (visited on 29/03/2023).
- [170] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Manual UCB/EECS-2014-54. EECS Department, University of California, Berkeley, May 2014.
- [171] Robert N. M. Watson. *The Arm Morello Board*. University of Cambridge, Computer Laboratory. URL: <https://www.cl.cam.ac.uk/research/security/ctsrds/cheri/cheri-morello.html> (visited on 17/11/2022).
- [172] Robert N. M. Watson et al. *An Introduction to CHERI*. Tech. rep. UCAM-CL-TR-941. University of Cambridge, Computer Laboratory, Sept. 2019.
- [173] Robert N. M. Watson et al. *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks*. Tech. rep. UCAM-CL-TR-916. University of Cambridge, Computer Laboratory, Feb. 2018.
- [174] Robert N. M. Watson et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. Tech. rep. UCAM-CL-TR-951. University of Cambridge, Computer Laboratory, Oct. 2020.
- [175] Robert N. M. Watson et al. *CHERI C/C++ Programming Guide*. Tech. rep. UCAM-CL-TR-947. University of Cambridge, Computer Laboratory, June 2020.
- [176] Robert N. M. Watson et al. ‘CHERI: A hybrid capability-system architecture for scalable software compartmentalization’. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 20–37.
- [177] Ofir Weisse, Valeria Bertacco and Todd Austin. ‘Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves’. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA ’17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 81–93.
- [178] Jonathan Woodruff et al. ‘The CHERI capability model: Revisiting RISC in an age of risk’. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE. 2014, pp. 457–468.
- [179] Xinyu Yang et al. ‘Towards a Low-Cost Remote Memory Attestation for the Smart Grid’. In: *Sensors* 15.8 (2015), pp. 20799–20824.
- [180] Ning Zhang et al. ‘CacheKit: Evading Memory Introspection Using Cache Incoherence’. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2016, pp. 337–352.
- [181] Ning Zhang et al. *TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices*. Cryptology ePrint Archive, Paper 2016/980. 2016.
- [182] ChongChong Zhao et al. ‘On the Performance of Intel SGX’. In: *2016 13th Web Information Systems and Applications Conference (WISA)*. 2016, pp. 184–187.

Appendix A

Software artefact

This project report includes a supplementary software artefact, which can be accessed either as a compressed archive file or via GitHub:

- <https://github.com/kiancross/terios>