2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-6512-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/EuroSP57164.2023.00070

CHERI-TrEE: Flexible enclaves on capability machines

Thomas Van Strydonck*§, Job Noorman*, Jennifer Jackson†, Leonardo Alves Dias†
Robin Vanderstraeten‡, David Oswald†, Frank Piessens*, Dominique Devriese*
*KU Leuven †University of Birmingham ‡Vrije Universiteit Brussel

***Abstract*—This paper studies the integration of two successful hardware-supported security mechanisms: capabilities and enclaved execution. *Capabilities* are a powerful and flexible security mechanism for implementing fine-grained memory access control and compartmentalizing untrusted or buggy software components. Capabilities have a long history but have gained significant momentum recently, as evidenced by ARM's experimental Morello processor that supports the Capability Hardware Enhanced RISC Instructions (CHERI). *Enclaved execution* is a popular mechanism for dynamically creating Trusted Execution Environments (TEEs), called *enclaves*. Enclaves are isolated execution contexts that protect the integrity and confidentiality of software in the enclave (even against compromised system software) and that support attestation.**

Integrating capabilities and enclaved execution in a single processor is challenging because they overlap partially in their security objectives, and a clean integration should unify the way in which these overlapping objectives are achieved. In addition, it is not obvious how attestation should interact with capabilities. In this paper, we propose CHERI-TrEE: a novel design for a processor that cleanly integrates support for both capabilities and enclaved execution. CHERI-TrEE targets low-end embedded systems without virtual memory. We show that CHERI-TrEE is greater than the sum of its parts by showing how it naturally supports useful features that have traditionally been hard to support in enclaved execution, like dynamically growing and shrinking enclaves, non-contiguous and nested enclaves, sharing of memory between enclaves etc. We implement our proposal both in hardware on a RISC-V processor, as well as in a small software hypervisor on top of ARM Morello, and evaluate impact on performance and hardware resources.

***Index Terms*—enclaves, TEE, trusted execution, capability machines, CHERI, CHERI-RISC-V, ARM Morello**

1. Introduction

There is a wide variety of hardware-supported mechanisms to securely isolate software, each with its strengths and limitations. This paper studies the integration of two related but fundamentally different mechanisms: capabilities and enclaved execution.

Capabilities are unforgeable tokens of authority granting rights to system objects. They are a powerful security mechanism for implementing fine-grained access control and for compartmentalizing untrusted or buggy software components. *Capability machines* implement the concept of capabilities at the machine code level: they provide hardware support for capabilities by defining an instruction set architecture (ISA) that provides access to system memory only through *memory capabilities*, a kind of hardware-supported fat pointers. The ISA is designed to ensure that software can only create capabilities that represent a subset of the authority that the software already holds. Hence, capabilities are a secure basis for implementing memory access control and isolation. Next to memory capabilities, capability machines can support a wide variety of other kinds of capabilities, including, for example, *object capabilities* that can control access to software defined objects, or *sealing capabilities* that can symbolically encrypt or decrypt other capabilities. Capability machines have a long history [1], but have gained significant momentum over the last decade with, for instance, the development of the CHERI system [2], and with the ARM Morello project [3] that integrates the concepts of CHERI in the widely used ARM architecture.

Enclaved execution is a security mechanism that supports the run time creation of *enclaves*, execution environments for software components that are strongly isolated and that can attest their identity to other code running either locally on the same platform or remotely on other platforms. The idea of Enclaved Execution Systems (EES) is more than a decade old [4], and currently many implementations are available [5], [6], [7], [8], [9], [10], both in research prototypes, as well as in commercial systems like Intel's Software Guard Extensions (Intel SGX). We will therefore use the term *enclave* to designate a general *protected module*, rather than *SGX-like* enclaves specifically, as sometimes happens in the literature.

Problem statement. Assume the designer of an embedded system has decided to use a capability processor (we motivate the use of capabilities on embedded processors below). This paper then aims to find the best way to integrate enclaved execution into the capability processor design. This is challenging because (1) these mechanisms overlap partially in their security objectives (for instance, both mechanisms provide their own form of controlled invocation) and a clean integration should unify how these overlapping objectives are achieved, and because (2) it is not obvious how (remote or local) attestation should interact with capabilities.

To the best of our knowledge, this paper is the first to propose a design that cleanly integrates both mechanisms in a single processor ISA. Roughly speaking, our design proceeds as follows: enclaved execution is known to be a *complex* security mechanism [11], hence we first decom-

§. Corresponding author. thomas.vanstrydonck@kuleuven.be

DOI 10.1109/EuroSP57164.2023.00070

pose it into a set of maximally independent primitives. Next, we implement these primitives on top of the state-of-the-art CHERI capability machine [2], [12], maximally reusing existing CHERI mechanisms. For instance, enclave isolation reuses standard CHERI object capabilities. The new mechanisms we introduce are lightweight, orthogonal, and contribute features that do not yet exist in CHERI. They allow (1) obtaining guarantees about exclusive ownership of a memory region (i.e., no other code on the system has a reference to said memory), and (2) obtaining sealing capabilities (a private key used to symbolically encrypt or sign other capabilities) derived from an enclave's identity.

We demonstrate that the resulting system, CHERI-TrEE, can be used as an EES, allowing other code on the system to establish trust in an enclave. CHERI-TrEE is greater than the sum of its parts. Our reuse of existing CHERI features achieves economy of mechanism and reduces complexity and cost. At the same time, the resulting EES innately supports features that have traditionally been hard to accommodate in EESs: dynamically growing and shrinking enclaves, non-contiguous enclaves, nested enclaves, sharing of memory between enclaves, etc.

Target systems. Capabilities and enclaved execution have been studied for both low-end embedded systems and higher-end systems that support virtual memory or virtualization instructions. In this paper, we focus on low-end embedded systems without virtual memory. Although we demonstrate an implementation on top of ARM Morello, a high-end architecture, this proof-of-concept implementation does not use virtual memory and is to be interpreted (for the time being) as a proof of feasibility of a software-only implementation of CHERI-TrEE.

Note that there are already good arguments for applying CHERI-like primitives to embedded processors, independently of CHERI-TrEE. The benefits and costs of CHERI-like protection (compared to e.g., MPUs) on embedded systems have been studied extensively in the literature [13], [14], [15]. CHERI-TrEE is intended to additionally offer EES security primitives when CHERI protection is already used on an embedded system for its other benefits. As such, the costs of supporting CHERI-TrEE should be regarded separately from the costs incurred by adding support for capabilities to the processor.

Attacker model. As usual for EESs, we assume an attacker that can control *all* software present on the system, notably including privileged software such as the operating system. CHERI-TrEE can be implemented in either hardware, software, or a mixture thereof. Depending on this choice, a small software interrupt handler and a small hypervisor might be considered trusted. Other desirable enclave properties that are largely orthogonal to capabilities are left out of scope: availability guarantees, hardware-based attacks (cold-boot attacks, malicious DRAM, ...) or side-channel attacks (timing or cache side-channels, memory bus snooping, ...). Note that this assumption inherently limits the size of our TCB compared to systems with a stronger attacker model, that implement additional security features (e.g., memory encryption).

Contributions. To summarize, in this paper we contribute:

- A decomposition of enclaved execution into more basic primitives.
- The design of CHERI-TrEE, a capability-based ISA extension that supports these primitives and hence integrates capabilities and enclaved execution.
- A specification and security argument for classic enclaved execution operations (initialization, unloading, local attestation, secure communication) on top of CHERI-TrEE.
- An open-source implementation of CHERI-TrEE on top of RISC-V, including a Sail specification of the ISA and a hardware implementation on FPGA, as well as a proof-of-concept implementation on ARM Morello [3]. Our evaluations and benchmarks are open source. [1]
- An evaluation of the impact on performance and hardware resources of the extension on the considered platforms.

For reasons of anonymity, we cannot provide the implementation sources during the review process. Files can be made available to reviewers on request.

2. Background

We implement all EES primitives by extending CHERI-like capability machines [2]. We thus briefly recap the background on enclaved execution and capabilities.

2.1. Enclaved execution

Enclaved execution is a security mechanism that enables *secure remote computation* [16] with a small Trusted Computing Base (TCB). It supports the runtime creation of *enclaves*, strongly isolated software components that can attest their identity to other code running either locally on the same platform or remotely on other platforms. A number of variations on this idea have been designed and implemented by researchers [4], [6], [9], [8], [10], and Intel have commercially implemented the Software Guard Extensions (Intel SGX). We provide an overview of existing designs in Section 7.

The typical life cycle of an enclave goes as follows: First, untrusted code creates the enclave and initializes it from a static binary code image (e.g., a .dll file). After initialization, the enclave is supposed to be isolated from all other (non-TCB) software, and has an enclave identity based on the code image initially loaded. At this point, interaction with the enclave is possible: the enclave context (untrusted code and/or other enclaves) can call into the enclave, and the enclave can call out to its context. Such interactions can be authenticated: the context can get proof of the identity of an enclave (*attestation*), and similarly, the enclave can get proof of the identity of other enclaves calling in. Most EESs extend attestations to remote platforms: a remote party can get proof of the identity of an enclave it is interacting with. Finally, an enclave can be terminated. Care must be taken to ensure that enclave secrets do not leak on termination.

Existing EESs vary widely in the implementation details of this life cycle. There are differences in the way isolation is implemented, in the interaction with other isolation mechanisms (like virtual memory), and in how

1. See https://github.com/proteus-core/cheritree

termination is handled. However, all existing systems face challenges in securely handling natural features, including: nesting of enclaves, non-contiguous enclaves, dynamic enclave memory (de-)allocation, efficient memory sharing between enclaves, the number of supported enclaves, etc.

2.2. Capability machines and CHERI

Capability machines define an ISA that provides access to system memory only through *memory capabilities*, a kind of unforgeable hardware-supported fat pointers that carry metadata about the bounds over which they grant authority. The ISA is designed to enforce *monotonicity* of authority: software cannot forge capabilities that increase the access to memory that it initially possessed. Hence, capabilities can be used to enforce memory access control and isolation; properties we will rely on when designing secure enclaves in Section 3. The base platform on which we build our design is a variant of CHERI-RISC-V. The CHERI-RISC-V capability metadata relevant for this paper includes *base* and *length* fields, *otype*, *permissions* and an *address* field. A memory capability can be used in store and load instructions to access its address, provided the address is in the range $[base, base+length)$, and compliant with the permissions specified in the permissions field (e.g., read, write, execute). The ISA provides instructions to inspect and modify metadata fields, but only in ways that respect the aforementioned monotonicity of capability authority. For instance, bounds can be reduced but not expanded, e.g., the `CSetBoundsExact cd, cs1, rs2`[2] instruction takes an existing capability in `cs1` and a new bound length in `rs2`, which must be equal to or less than the length of `cs1`, and creates a new capability in `cd` with the specified bound length.[3]

CHERI supports other kinds of capabilities besides memory capabilities, for which the interpretation of the metadata and the possible uses vary. Notable for our purposes are so-called *sealed capabilities* [2], a type of CHERI capability that cannot be used in operations like stores and loads or have its fields altered; only reading fields is allowed. Sealed capabilities are sealed with a specific seal (i.e., a key), represented by a value in the otype field. The highest otype value represents an unsealed capability.

Capabilities can be sealed and unsealed through the `CSeal` and `CUnseal` instructions. The `CSeal cd, cs1, cs2` instruction takes unsealed capabilities in `cs1` and `cs2`, and seals `cs1` with otype equal to the address field of `cs2`, placing the result in `cd`. To avoid arbitrary capabilities being used to seal other capabilities, `cs2` must have the bespoke `Permit_Seal` permission bit set, which allows it to seal other capabilities with otypes within its bounds. The `CUnSeal cd, cs1, cs2` instruction is the dual of `CSeal`; `cs1` must now be sealed, and `cs2` is required to have the `Permit_UnSeal` permission bit set. The otype in the address field of `cs2` must match the otype of `cs1`. The unsealed result is placed in `cd`. Software cannot bypass the security guarantees offered by sealed capabilities: as capability authority must evolve monotonously, not even privileged software can set the `Permit_Seal/Unseal` bits or forge sealed capabilities.

The purpose of these capabilities in CHERI is dual. First, otypes can be used to implement efficient symbolic encryption and signing, using the aforementioned permission bits. A capability that has both `Permit_Seal` and `Permit_UnSeal` bits set can be used for symmetric encryption, because it can both encrypt (seal) and decrypt (unseal) messages (capabilities). If a capability carrying the `Permit_Seal` permission is made public and a `Permit_UnSeal` counterpart is kept private, we get public key encryption. The converse setup results in a digital signature scheme. These observations form the basis of our design for secure enclave communication in Section 3.2.

Secondly, sealed capabilities can be used to implement the aforementioned object capabilities that control access to software defined objects. The `CInvoke cs1, cs2` instruction enables this functionality and permits secure domain transitions. If both `cs1` and `cs2` contain capabilities with matching otypes and solely `cs1` has execute permission, then the two capabilities are atomically unsealed, and `cs1` is installed in the program counter register, continuing execution from the invoked domain. Together, `cs1` and `cs2` are said to constitute a *sealed pair*, with `cs1` the code and `cs2` the data capability. Such pairs can safely be passed to adversarial code, as both capabilities are sealed and can only be invoked together. To distinguish sealed pairs from other sealed capabilities, a permission `Permit_CInvoke` determines whether sealed capabilities can be invoked together. We will use sealed pairs to perform secure domain transitions into enclaves in Section 3.2.

3. The design of CHERI-TrEE

In this section, we propose decomposing the requirements for enclaved execution into orthogonal properties (Section 3.1). These properties serve as a guideline for the design of CHERI-TrEE, our capability-based EES. Section 3.2 considers the CHERI capability machine as a starting point and lists for each property whether it can be enforced through existing capability primitives or whether extensions have to be defined. With this analysis in hand, Section 3.3 discusses the security model and the operations offered by the TCB in a backend-agnostic way. Section 3.4 discusses implementation details that Section 3.3 brushed over, and Section 3.5 contains an abstract discussion of why our design ensures isolation and secure communication. Finally, Section 3.6 highlights how our bottom-up approach, maximally reusing the flexibility of architectural capabilities, allows for greater flexibility than the state of the art.

Capabilities and enclaved execution have been studied for systems that support virtual memory and systems with a single physical memory address space. This paper focuses on the latter kind of system. The combination with virtual memory and address translation brings additional challenges that are left for future work discussed in Section 6.

2. As in the RISC-V and CHERI specifications, we use `rs1` and `rs2` for source and `rd` for destination integer registers, while using `cs1`, `cs2`, and `cd` for capability registers throughout this paper.

3. Assuming the requested bounds can be represented in CHERI's compressed capability representation.

3.1. Decomposing enclaved execution into core properties

In our view, the core security properties required to build an EES can be decomposed into the five categories listed below. Some other properties (e.g., confidential deployment [9], secure storage) are also relevant, but either not essential to the construction of a *functional* EES, or outside our attacker model. The core properties are:

① **Exclusive access**: A mechanism to guarantee exclusive access to specific memory areas is required for the secure initialization of an enclave. In principle, the enclave can be allowed to share its uniquely owned memory once it has been securely initialized, but few enclaved execution systems support this. For example, Intel SGX relies on *Processor Reserved Memory* and the initialization process to guarantee exclusive access, while Sancus relies on program-counter based access control.

② **Controlled invocation**: To avoid reasoning about the correctness and security of many possible control-flow paths, enclaves can only be invoked at predefined *entry points*. Examples are *ecalls* in Intel SGX and *entry points* in Sancus. In most systems, entry points are declared at enclave creation time, but at least conceptually, new entry points could also be created dynamically.

③ **Enclave identities and attestation**: To enable multi-enclave/distributed applications, enclaves and third parties require a mechanism to establish trust in the correct initialization and execution of another enclave. This authentication process, called attestation, occurs either locally or remotely. It requires a notion of *enclave identity*, usually based on a cryptographic hash of the enclave code and metadata. Locally, the architecture provides a mechanism to communicate identities and authenticate enclaves to other code running on the same platform. Remote attestation requires cryptographic support, either in the form of a public key infrastructure (as in Intel SGX) or a symmetric key derivation scheme (as in Sancus).

④ **Secure communication**: A mechanism to securely communicate between two enclaves, both locally and remotely, is essential to achieve integrity and confidentiality. Locally, either CPU registers or shared memory can be used. Because enclaves might be deinitialized and replaced at any point, encrypting and signing messages cryptographically is required to ensure integrity and confidentiality. The efficiency of the involved cryptography is performance-critical. To avoid having to sign and encrypt messages, some EESs have built-in mechanisms to check liveness of the sender and receiver enclave. Here, time-of-check to time-of-use (TOCTOU) attacks can be an issue (e.g., for Sancus). For the remote case, the key distribution infrastructure of attestation can often be reused for secure communication with standard protocols.

⑤ **Secure interruptability**: When an enclave's execution is interrupted, its register state should not be accessible to untrusted code as this potentially breaks confidentiality guarantees offered by the enclave. Additionally, the enclave's register state must be correctly and securely restored after servicing the interrupt.

Because capabilities provide no protection during network transition, the design of remote attestation and remote secure communication would mostly reuse existing solutions. The remote aspects of enclaved execution are therefore left for future work and further discussed in Section 6.

3.2. Satisfying the security properties

We now discuss how we enforce these properties in our design that builds on CHERI, reusing CHERI primitives where possible (✓) and extending the TCB otherwise (✗). In the remainder of Section 3, the TCB is taken to be hardware-only (except for a small software interrupt handler). However, Section 4.4 illustrates that this is not required.

✗ **Exclusive access** at runtime is not supported out of the box in CHERI, so we add *memory sweep* functionality to the TCB. During a memory sweep, the TCB checks the whole memory and all architectural registers (including special registers) for the presence of capabilities that overlap with a given capability. To successfully complete enclave initialization, the unique ownership of an enclave's code and data sections needs to be verified through such a sweep. The memory sweep is a simple solution to exclusive access that can be optimized further and for which alternatives exist. We discuss these in more detail in Section 6.

✓ **Controlled invocation** can be implemented using the aforementioned sealed capability pairs [2]. Concretely, if the enclave only shares sealed capabilities to specific entry points with adversary code, then capability monotonicity ensures that the enclave cannot be otherwise entered. Alternatively, controlled invocation can be implemented through so-called *enter capabilities* [17], which intuitively combine both the code and data capabilities of a sealed pair into a single capability. This alternative could have simplified a few aspects of our design, but we did not implement it, as enter capabilities have only recently been added to CHERI.

✗ **Enclave identities and attestation** are not built into CHERI. The architecture hence has to use a cryptographic hash function to calculate an enclave's identity by hashing the code section. Additionally, to enable local attestation, an instruction needs to be added to securely look up this generated identity.

✓ **Secure communication** can be efficiently implemented locally through the symbolic encryption provided by sealed capabilities. Every enclave has a signing capability cap_sign with otype o_sign in its address and both `Permit_Seal` and `Permit_UnSeal` set. The enclave solely shares the `Permit_UnSeal` part of this capability with other code, so that it can exclusively authenticate its messages and other enclaves can verify them. Although it might seem unintuitive that a *capability* (rather than just the integer otype o_sign) is required to *verify* a signature, without this capability, recipients of signed values would have no way to remove the signature and access the payload underneath. Dually, the enclave has an encryption capability cap_enc with otype o_enc and shares only its `Permit_Seal` part so that only the enclave can decrypt messages encrypted with o_enc. We refer to the shared halves of both capabilities as an enclave's *public keys*, and to the full-authority versions as its *private keys*.

✗ **Secure interruptability** cannot be guaranteed in the presence of an untrusted interrupt handler: the confi-

dentiality and integrity of an enclave depend on an adversarial interrupt handler not being able to read security-critical capabilities (e.g., the capability for the data section or cap_sign) from its register state. We resolve this problem by installing a minimal, trusted interrupt handler that cannot be altered or bypassed by untrusted code. The handler performs the necessary register sanitization before passing control to the adversary and restores the register state on return. This solution is similar to the *security monitor* employed in Keystone [10], but our handler is more limited in scope (e.g., it does not need to manipulate memory protection state). Although we currently implement interrupt handling in software as a proof of concept, a Sancus-style hardware implementation is entirely feasible.

3.3. Fleshing out the design

Having studied the security of enclaves at a conceptual level in Sections 3.1 and 3.2, this section now discusses the modifications we made to the CHERI architecture to implement our design.

Our first addition to the TCB is the enclave table in Fig. 1, which connects enclave identities to their otypes. The table enables local attestation: a new operation performs a secure lookup of the enclave identity associated with an otype that was received from untrusted code.

Each enclave entry in the table contains three otypes: the aforementioned otypes o_enc and o_sign, that serve as otypes for the enclave's cap_enc and cap_sign capabilities, and an otype o_entry that will be used to create a sealed entry point from the enclave's unsealed code and data section. We allocate these otypes such that they are consecutive and 2-bit-aligned. This allows the TCB to store only the common prefix in the enclave table, which we call the enclave identifier (*eid*) and allows an enclave to represent ownership of all three otypes in a single capability cap_seals (cf. Fig. 1).

We ensure that otypes are spatially and temporally unique to avoid collisions with previously allocated enclaves and other usages of sealed capabilities. In case otypes are required for other purposes on the capability machine, the otype space should be split up.

We now study the operational aspects of our design in more detail. We discuss (de)initialization, local attestation, communication between enclaves, stand-alone memory sweeps and secure interrupts.

Initialization. Enclaves register themselves with the TCB by invoking an initialization operation, passing capabilities cap_code and cap_data as arguments. Initialization will perform the following steps:

1. Perform a memory sweep to check unique ownership of cap_code and cap_data.
2. Allocate fresh otypes for the enclave and store the capability cap_seals for them to the start of cap_data so that the enclave can set up its symbolic encryption upon invocation.
3. Use o_entry to seal cap_code and cap_data, transforming them into a sealed pair and ensuring controlled invocation.
4. Compute the enclave's identity I by hashing its code section.
5. Store this identity I in the TCB along with its *eid* to allow secure lookups. If the TCB is full, the instruction fails.

After successful completion, the enclave has been initialized and the TCB contains an appropriate entry (cf. Fig. 1, we explain the *Temporary* field below). Untrusted code can now invoke the enclave by executing `CInvoke` on cap_code and cap_data.

Upon first invocation, the enclave's code will initialize any necessary state and enable symbolic encryption by returning the public parts of its sealing and encryption keys. The enclave can create a so-called *fast entry point* at a different offset to skip initialization on subsequent invocations (taking care that the initial entry point does not reinitialize the enclave if it is invoked again).

Deinitialization. A TCB operation deinitialize allows an enclave to deregister itself. The enclave authenticates itself by providing its cap_seals capability (with both `Permit_Seal` and `Permit_UnSeal` permissions set). Thus, only the enclave itself or parties that were granted access to (subranges of) cap_seals can deinitialize it. After deinitialization, an enclave can clear sensitive data (including its seals) and return the capabilities for its code and data section to untrusted code. As in Sancus, a processor reset is required to deinitialize rogue enclaves. A processor reset is also required to reuse the otypes of decommissioned enclaves, since we do not support memory sweeps to reclaim otypes.

Local attestation. An enclave gets access to another enclave's entry points and public encryption/signing keys by e.g., retrieving them from an (untrusted) enclave registry. Regardless of the origin, enclaves require a way to verify that an otype corresponds to an enclave of interest. The TCB offers a local attestation operation that takes an otype and returns the enclave identity of the corresponding enclave (by storing it to memory through a provided capability). Note that when an enclave calls into another enclave, the callee is not required to attest the caller ahead of time. Indeed, the caller can pass its public keys along with the arguments, so the callee can perform local attestation of the caller while processing the call.

Secure communication. Different primitives combine to secure communication between enclaves. First, any confidential arguments or return values are *encrypted* with the recipient's public encryption key. Additionally, as capabilities carry authority, even non-confidential capability arguments and return values must be encrypted if their authority should not be made available to untrusted code.

Second, when returning from a call, the callee should *sign* (part of) the return value to allow the caller to confirm that its call was indeed processed by the callee. Conversely, the callee might also require the caller's signature to authenticate the caller; the callee could e.g. have a whitelist of enclaves whose requests it services. Lastly, to avoid replay attacks, the caller can make use of a *nonce* as part of its requests.

As the literature shows, it is possible (but difficult) to construct a wide variety of secure communication protocols using the available primitives (asymmetric encryption, signing, and nonces) [18]. Since our aim is to introduce a capability-based design for enclaved execution, not

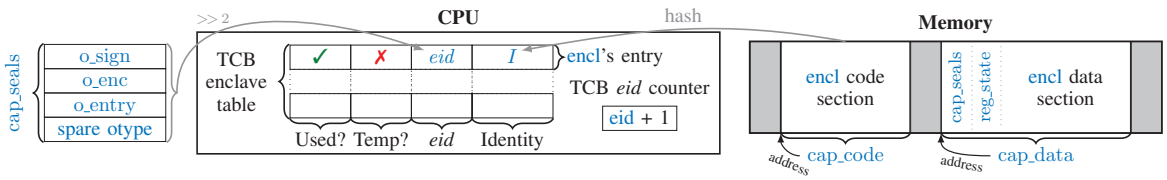

Figure 1. Memory and TCB state after initialization of an enclave encl. The TCB's four enclave table fields indicate whether an entry is in use (Used?), whether it is temporary (Temp?), and record the enclave's identifier (*eid*) and identity (Identity). Importantly, the TCB does not track the enclave's memory layout, but solely connects the enclave's *eid* and identity to each other. All values relating to encl are typeset in blue; its identity (I), identifier (*eid*), code and data sections cap_code and cap_data, four assigned otypes, and their capability cap_seals. The reg_state region stores the enclave's register state during an interrupt. Gray arrows illustrate how I is obtained by hashing the code section and *eid* by right-shifting the otypes. Figure layout inspired by Noorman *et al.* [9].

design secure communication protocols on top, we defer the development of these protocols to future work.

Separate memory sweep. As we will discuss in Section 3.6, by offering the memory sweep functionality as a stand-alone operation as well, we can increase the flexibility of the enclave model. This operation performs a memory sweep for a given capability and outputs a boolean indicating success or failure.

Interrupt handling. For secure interrupts and system calls, we install a fixed, non-bypassable, trusted interrupt handler. When an enclave is interrupted, control passes to the trusted handler. This handler is responsible for storing enclave state to an enclave-designated location (before control is passed to an untrusted interrupt handler) and restoring the state afterwards. It includes measures to prevent reentrancy issues. Much like Keystone, we leave the extension of our scheme to a multi-threaded setting, as well as nested interrupts, attestation of the interrupt handler, and the delegation of synchronous interrupts and errors to an enclave-private interrupt handler to future work.

3.4. Technical details

Otypes and eid. As mentioned above, each enclave uses an *eid* to represent all of its otypes o_sign, o_enc and o_entry. In fact, we reserve an additional fourth *spare* otype (cf. Fig. 1), which the enclave can use to e.g. create additional entry points, because four adjacent otypes share all but their last two bits. The TCB allocates four fresh seals by increasing a hardware counter (TCB *eid* counter in Fig. 1), failing when the otype space has been depleted. Any enclave otype is then efficiently convertible into the *eid* by a 2-bit right shift. Note that the specific role of each otype (except for o_entry) is up to software convention.

For proper operation, we assume the otype space available to enclaves to be sufficiently large. This requirement is currently not met for CHERI Concentrate 64 (CC64), the compressed 64-bit representation of capabilities that CHERI defines for RV32 (32-bit RISC-V); only 4 bits are available for the otype field [19]. To resolve this issue, the CHERI specification contains a proposal to store otypes of sealed capabilities as metadata in memory instead of having the otype be in the capability itself [12]. For RV64 (64-bit RISC-V), 18 bits are available in CHERI Concentrate 128 (CC128), which poses fewer problems.

Invoking sealed capabilities as entry points. One might ask whether o_enc or o_sign could be reused instead of o_entry to seal the entry point. The answer is negative; reusing o_sign would allow adversaries to simply unseal the enclave's sealed code and data sections without calling `CInvoke`, whereas reusing o_enc would allow the adversary to create their own code section, and use that to unseal the enclave's data section by executing `CInvoke`. For similar reasons, it is important that `Permit_CInvoke` is unset on capabilities sealed with enclave encryption keys.

Mapping TCB operations to RISC-V instructions. Unfortunately, the initialization TCB operation is difficult to implement as a single instruction on a RISC ISA like RISC-V, where instructions generally have only one output operand. Initialization produces two outputs, because it has to overwrite both cap_code and cap_data with their now-sealed variants. To solve this, we split the operation into two separate instructions; `EInitCode` and `EInitData`, which initialize the enclave's code and data sections in two consecutive phases.

The `EInitCode cd, cs1` instruction is called first, with cap_code in `cs1`. To ensure that the unsealed cap_code is overwritten by its sealed counterpart, `cd` and `cs1` are required to be equal. For efficiency reasons, `EInitCode` does not yet check uniqueness of cap_code, as `EInitData` will perform a single sweep for cap_code and cap_data simultaneously. Computing the enclave's identity is also deferred until `EInitData`, since adversarial code might still have access to cap_code.

Once the *eid* has been generated, `EInitCode` seals cap_code with o_entry, sets its address to its base (to avoid arbitrary entry offsets) and writes it to `cd`. `EInitCode` will generate the following *temporary* TCB entry (cf. Fig. 1), to be later finalized by `EInitData`, which does contain the enclave's *eid* but not its identity:

✓	✓	*eid*	Don't care

. Note that `EStoreId` ignores TCB entries marked "temporary".

After `EInitCode`, the `EInitData cd, cs1, cs2` instruction finalizes the enclave's initialization. It requires a sealed code section cap_code (the result of calling `EInitCode`) in `cs1`, and an unsealed data section cap_data in `cs2`. As before, `cd` and `cs2` are required to be equal. The `EInitData` instruction now:

1. Looks up a temporary TCB entry with an *eid* corresponding to cap_code's otype.
2. If found, checks unique ownership of cap_code and cap_data in a single memory sweep.

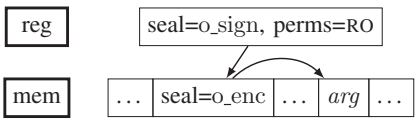

Figure 2. Combination of different security primitives: argument *arg* is encrypted with otype o_enc, then signed with otype o_sign.

3. Verifies that cap_code does not contain any capabilities. Note that cap_data *is* allowed to contain capabilities (including internal references to cap_data/cap_code), and is therefore not zero-initialized, as in e.g. Sancus.
4. Calculates the enclave identity I as the hash of the contents of cap_code, stores I in the temporary entry for *eid*, and marks the entry as permanent (cf. TCB state in Fig. 1).
5. Stores the cap_seals capability in the first address of cap_data. Writes cap_data, sealed with o_entry, to `cd`.

Deinitialization is offered as a `EDeInit rd, cs1` instruction taking cap_seals in `cs1` and indicating success or failure in `rd`. Similarly, local attestation is offered through an `EStoreId rd, rs1, cs2` instruction, which takes an otype in `rs1` (any otype assigned to an enclave), and writes the identity of the corresponding enclave in the TCB (if any) to memory through the capability in `cs2`. The separate memory sweep is performed by the `IsUnique rd, cs1` instruction, which takes its input capability in `cs1` and again indicates success in `rd`.

Trusted exception handler. The trusted exception handler uses the enclave's data capability cap_data to save its register state into the reg_state region (cf. Fig. 1). Then, it seals cap_data using a unique, private otype o_handler, places the result in a predefined register, and jumps to the untrusted interrupt handler. For this scheme to work, cap_data should be present in a fixed register at all times. We use the `idc` (invoked data capability) register because it is atomically set to the unsealed data section when calling into an enclave using `CInvoke`. Once the untrusted code finishes servicing the interrupt, it returns to the trusted handler, which unseals the provided data section, restores the enclave's registers (apart from `pc`) and finally returns. The return should simultaneously reenable interrupts and restore the enclave's code capability. In the concrete case of RISC-V, the interrupt handler is installed in the `mtvec` register. It returns to the interrupted enclave through the `mret` instruction.

The tag of the `pc` capability stored in the reg_state region effectively functions as an "is interrupted"-flag: the trusted handler sets it by storing an enclave's `pc` at interrupt time, and unsets it before returning to said enclave. This flag is used by enclaves to avoid reentrancy issues: if the tag is enabled when an enclave is invoked, the enclave simply returns. Additionally, the trusted handler uses the flag to avoid storing the registers of a previously interrupted enclave and restoring the registers of a non-interrupted enclave.

Signing and encrypting simultaneously. One might wonder how a capability with a single otype field can simultaneously represent encryption and signing. Fig. 2 illustrates that the solution is indirection: a memory argument *arg* is first encrypted by an in-memory capability with otype o_enc, which is in turn signed by a capability with otype o_sign, present in one of the registers. This immediately illustrates one of the pitfalls of secure communication in our setting: messages are represented by capabilities rather than bit-strings. Copying a bit-string corresponds to a deep copy, whereas copying a capability creates an alias. This is the reason the top-most capability in Fig. 2 has read-only permission—if it allowed writes, an adversary could take a copy of the capability, remove the signature on the copy using the public signing key, and overwrite the underlying sealed capability. This would allow the adversary to create arbitrary capabilities, signed with a third party's seal. Similar concerns are at play when encrypting capabilities.

3.5. Security analysis

To summarize the above design, we now provide an analysis for why each stage of an enclave's lifetime respects exclusive access and secure communication. This assumes the enclave is correctly implemented and does not leak any of its private state.

- System boot: the system boots with an omnipotent capability. A small, trusted bootloader ensures that this omnipotent capability does not grant access to seals reserved for enclaves.
- Initialization: after initialization, the memory sweep guarantees us that no aliases for the enclave's entry point exist. Freshness of the generated seals ensures that no code has authority over the enclave's otypes.
- During operation: as long as the enclave does not leak its private state, no external code can unseal capabilities sealed by the enclave. Additionally, the enclave can distribute the public halves of its sealing and signing capabilities without compromising its security, because of the semantics of sealed capabilities.
- During interrupts: the enclave's register state is stored in its data section, which is sealed such that only the trusted interrupt handler can restore the enclave's state. Reentrancy is currently unsupported: an interrupted enclave simply returns upon further invocation.
- Deinitialization: the enclave erases its seals before returning its authority to the context. This ensures forward secrecy and authenticity of messages it encrypted and signed. Other secrets and capabilities in the data section are scrubbed on an application-specific basis.

Formal modeling and proof of these and other properties from Section 3.1 are left for future work.

3.6. Flexibility of our bottom-up design

There are two main reasons for the maximal reuse of existing capability primitives: First, the size of the TCB remains limited. Second, the resulting enclaves do not rely on the hardware to manage their authority but are self-governing, resulting in more flexible software-based design patterns. The cost of this flexibility is establishing unique ownership at runtime (i.e., the memory sweep) and an increased burden on the software developer to not leak security-critical capability authority. On the other hand, spatial memory safety does mitigate some API-level exploits of the types identified by Van Bulck *et al.* [20]. For example, if an enclave does not share any capabilities

pointing into its own memory with the adversary, then it does not require checks to ensure that pointers passed in from untrusted memory fall outside of its bounds, avoiding issues with improper implementation of these checks.

Although some EESs offer enclaves that are flexible in some of the below respects (see Section 7), the combination of different types of flexibility in our design, without requiring additional architectural changes, is novel. We support flexibility in the following ways:

Growing, shrinking, nested and non-contiguous enclaves. All of these are supported by our design because an enclave's footprint is not managed by the TCB, but rather determined by the capabilities it owns.

To grow an enclave, we employ the previously introduced `IsUnique` instruction. An enclave can request a sweep for a capability provided by untrusted code to verify unique ownership and add it to the enclave's memory footprint. In order to shrink, an enclave simply shares capabilities for part of its footprint with untrusted code.

In state-of-the-art EES, nesting enclaves (i.e., creating an enclave inside an existing enclave) is usually impossible because the hardware disallows enclave footprints from overlapping. One motivation for having nested enclaves is that it would allow for easy virtualization (inside an enclave) of code that itself uses enclaves. Nested enclaves can be initialized anywhere inside an enclave's uniquely owned memory. The only restriction is that the memory sweep fails if an external party holds a sealed pair for an entry point that overlaps with the nested enclave.

Enclaves can have non-contiguous footprints, e.g., to take exclusive ownership of an MMIO region mapped to a peripheral, reported to be impossible in Sancus [21].

Sharing memory. Sharing memory between enclaves amounts to simply sharing a capability for a uniquely owned memory region between said enclaves. No encryption is required. Once enclaves are done sharing memory, there needs to be a way to revoke access to this memory. For short-term sharing of e.g., arguments in memory, so-called *local* capabilities [12] (which reside in registers only) could be used. Alternatively, an enclave can check whether its shared memory has been released using the `IsUnique` instruction. Lastly, to repeatedly share non-confidential memory that will never contain any security-critical capabilities, enclaves can simply share a read-only view of said memory.

Early `EDeInit`. Once all parties that wish to communicate with an enclave complete local attestation, there is no need to keep the enclave's entry in the TCB. This is possible as the TCB entry does not provide any hardware protection, but serves the purely informative purpose of linking otypes and enclave identities. Hence, enclaves can prematurely execute `EDeInit` to free TCB space and still operate correctly.

Secure communication without liveness checks. One interesting characteristic of enclave communication in our design is that we do not need to rely on liveness checks (such as the ones present in Sancus) during secure communication. The caller enclave can invoke the callee without verifying whether the callee enclave still exists, and the callee returns to the caller without checking its liveness. We can afford to omit these checks because the alternative, namely both signing and encrypting messages, is very cheap (taking a single instruction, contrary to non-symbolic solutions), and hence can be applied by default. The advantages of omitting liveness checks are that the hardware is simplified and any TOCTOU issues related to checking and then invoking an enclave are avoided.

Two-way sandbox. The spatial memory safety offered by capabilities naturally turns enclaves into *two-way sandboxes* [22], meaning that other code running in the same address space is protected from the enclave: enclaves can only manipulate memory they have appropriate capabilities for. This prevents e.g. Boomerang attacks [23].

Dynamic entry points. Traditionally, enclaves either list their entry points at creation time or have standard, predefined entry points. As we reuse sealed pairs to enforce entry points, an enclave can create more entry points at runtime by creating a sealed pair. Such dynamically created entry points could be selectively shared with attested counterparties to avoid the need to re-attest them on subsequent calls. To avoid mixing parts of different pairs, the enclave either needs to use different otypes for different entry points or have an entry point identifier in each data section.

Relocatable enclaves. `EInitData` does not include the enclave's base into the hash, contrary to hashing in e.g., Sancus [9]. Consequently, a deployer is not required to know the memory address at which an enclave is located beforehand, providing greater ease in deployment. This does place the requirement of writing Position Independent Code (PIC) on the developer, but RISC-V has efficient support for this [24]. This design decision implies that multiple instances of the same enclave have the same identity. Fortunately, this poses no security risk, as these enclaves would have different otypes and are hence distinguishable locally. In the remote case, key derivation can include each enclave's otype to create different keys.

4. Implementation

We now discuss three different implementations of our design from Section 3: one RISC-V specification of the ISA extensions that is written at the instruction level, abstracting away from the hardware, one RISC-V hardware implementation for studying performance, and a prototypical implementation on the ARM Morello platform to show commercial feasibility.

4.1. Sail specification of the extended ISA

To obtain a more formal account of the architectural extensions proposed in Section 3.3, we created a software implementation of our design in Sail [25].

Sail is an ISA specification language used to formalize ISA semantics. On top of that, Sail models serve various purposes: documentation can easily be generated from them, emulators (in C and OCaml) can be derived from the source code, and definitions can be exported to various proof assistants to enable reasoning about properties of

the ISA. CHERI researchers heavily use Sail to formalize different architectural implementations, and hence contains an existing formalization of both 32-bit and 64-bit CHERI-RISC-V [26], on top of which we implemented extensions for CHERI-TrEE.

The Sail implementation of CHERI-TrEE is complete; all previously described architectural implementation details are included. The implementation inherits the characteristics of Sail-CHERI-RISC-V: both RV32 and RV64 implementations are supported in a modular fashion, the model does not support split register banks (instead, a single register bank is shared for both capabilities and integers), and the use of compressed CHERI Concentrate capabilities is mandatory. For our purposes, we were most interested in the emulation functionality. Because generating an assembler from a Sail specification is not yet supported, we also extended CHERI's fork of the LLVM compiler to support the newly added instructions. This allowed using the LLVM toolchain to compile assembly files into well-formed ELF-binaries that Sail's C-emulator can executed. We developed basic unit tests to check functional correctness of each new instruction, as well as a larger scenario test, and ran these on top of the C-emulator.

4.2. Proteus RISC-V CPU framework

To verify and evaluate our design, we implemented the primitives from Section 3.3 in hardware. We based our implementation on *Proteus*, our open-source RISC-V processor designed with configurability and extensibility as its main goals.

Proteus is heavily inspired by VexRiscv [27] and designed in SpinalHDL [28], a Scala-based Hardware Description Language (HDL). SpinalHDL is essentially an HDL code generator: Scala is used to generate an HDL description at runtime (using primitives provided by the SpinalHDL library), which is then converted to either Verilog or VHDL. Designs are simulated using an HDL simulator or synthesized on FPGAs.

Proteus uses a plugin architecture to configure and extend processors. At its core, Proteus provides pipeline stages and the ability to pass values from one stage to the next. The logic contained in stages is not fixed but can be configured through *plugins*. This allows for a lot of flexibility in, for example, the number of pipeline stages and the supported features (e.g., the RISC-V "M" extension is an optional plugin). Concretely, our implementation uses a classic 5-stage RISC pipeline consisting of an IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory Access), and WB (register Write Back) stage. The plugin system also enables the development of custom extensions without having to alter the core implementation files. Currently, Proteus provides plugins to implement in-order pipelines with support for RV32IM and machine mode. A powerful feature of Proteus is the concept of *services*, which allows plugins to provide customization points to other plugins. For example, the plugin that implements load and store instructions offers an interface to intercept the generated addresses. This is used by one of our CHERI plugins to perform the necessary permission checks without having to modify the existing plugin.

We extended Proteus with plugins implementing most of version 8 of the CHERI-RISC-V 32-bit specification [12]. It implements a split register file, capability manipulation, implicit memory access through *DDC* (the *Default Data Capability* register [12]) and *PCC* (the *Program Counter Capability* register [12]), explicit memory access through capabilities, exception handling, and storing capabilities in memory. To track capabilities in memory, an on-chip tag table is maintained that stores one tag-bit per capability-aligned word.

CHERI Proteus is not compliant with the CHERI specification in terms of the memory representation of capabilities: Instead of compressed capabilities (which are mandatory for RISC-V), Proteus uses a full-precision representation, which is easier to implement but causes in-memory capabilities to use 128 bits instead of 64 bits for CHERI-RISC-V 32 bit. This has the advantage that we have ample otype space for enclave seals (cf. Section 3.3). This also means we currently cannot take full advantage of the CHERI compiler toolchain, but are limited to the use of the assembler. However, this is sufficient to run complex software examples (see Section 5.1).

4.3. Implementing CHERI-TrEE on Proteus

To implement our design (Section 3.3), a number of components have to be added to the CPU core. To store the TCB state of enclaves, we add a table (`EidTable`), in which each entry stores the *eid* and identity (I) of an enclave. Entries also keep track of the current state of the enclave, which can be *allocated* (`EInitCode` has been called but `EInitData` not yet), *ready* (enclave fully initialized), or *empty* (the entry does not contain any enclave information). The table provides an interface to allocate a new entry (which increases the *eid* counter, see Section 3.3), retrieve an existing entry (used by `EInitData`), and to finalize an entry by storing I. When searching, `EidTable` iterates through the entries one by one. Note that the table size is parameterized and implemented as a plugin, allowing us to easily create an alternative implementation (e.g., one that stores entries in memory) without changing the rest of the design. To calculate enclave identities, we use a SHA256 implementation from the SpinalCrypto library [29].

As `EInitData` needs access to the memory bus, we decided to implement all new instructions in the MEM stage of the 5-stage pipeline. All instructions are based on state machines of varying complexity. `EInitCode` simply asks `EidTable` to allocate a new entry and, if successful, seals its input capability with o_entry (Section 3.4). `EInitData` is by far the most complex instruction to implement, as it needs to scan all registers and memory for overlapping capabilities. After performing sanity checks on its inputs (e.g., whether the code capability corresponds to an entry in `EidTable`), it starts by requesting exclusive access to the pipeline. This operation makes sure that the requesting instruction is the only one executing in the pipeline by flushing or invalidating stages containing other instructions. This is necessary to correctly perform the register scanning as otherwise, copies of registers might be available in pipeline stages. While having exclusive access, the state machine iterates over all general purpose and special capability registers to verify that there are no overlaps with the code or data capability of the enclave.

To scan memory, we use a service from the CHERI plugins. This service accesses the tag storage to produce addresses of valid capabilities without the need to perform memory loads. It currently only checks a single tag-bit per cycle, leaving room for future optimizations. Once we encounter a capability, we load it and check for overlap. If there are no overlapping capabilities, the hash of the code section is calculated by loading its contents word-by-word and providing it to the SHA256 block. The result is then stored in `EidTable` and the corresponding entry is marked as *ready*. Finally, the sealing capability is created and stored at the first address of the data section.

`EStoreId` checks that the hash fits in its input capability and iterates over the entries of `EidTable` to find the entry corresponding to the given seal. If it finds one and it is marked as *ready*, the hash is written to memory.

4.4. Implementation on ARM Morello

We implemented CHERI-TrEE on the first commercial CHERI-enabled processor, ARM Morello [3], firstly using ARM's FVP simulator, and then on the hardware when it became available. As noted before, the Morello architecture does not necessarily represent the typical embedded processor we envision for CHERI-TrEE; yet, we still use it as an implementation platform to show the feasibility of our design on the only commercial capability machine.

While for Proteus we realized functionality as an ISA extension, here we show that it is also feasible to implement CHERI-TrEE in (low-level) software: On Morello, one can use the Exception Level 3 (EL3) monitor or the EL2 hypervisor to pause (at least in a single-core scenario) an OS running at EL1 to perform the CHERI-TrEE operations, including the memory sweep. We built a small trusted hypervisor at EL2 to implement CHERI-TrEE operations, defining hypervisor calls to trigger an exception to EL2. The inputs are held in the first two registers on entry to EL2, and then passed on as parameters in the handler code. For the register sweep, all EL1 register values are saved on the EL2 stack following a hypervisor call. A capability pointing to this stack is passed through to the exception handler function so that the sweep is performed on the state of EL1 registers. The registers are restored on return, except for the overwritten return values.

Our implementation closely resembles the Proteus implementation, for ease of testing and comparison. Differences in hardware however inevitably lead to deviations. For example, unlike the RISC-V implementations, hypervisor calls on Morello are not limited to two inputs and one output, enabling several improvements. These include having a single initialization instruction `EInit`, and initializing a group of enclaves at the same time. The latter variant is particularly beneficial due to the size of memory on the Morello platform: a substantial amount of time is spent on the memory sweep, checking for the presence of capabilities. Because the relative amount of time spent on checking for overlaps is small, the initialization time can be halved for a two-enclave system using this approach (or proportionally reduced for a larger number of enclaves). We implemented this variant on Morello as an optional feature by defining the number of enclaves to be initialized in a single block, and passing the required number of enclave capabilities to the hypervisor.

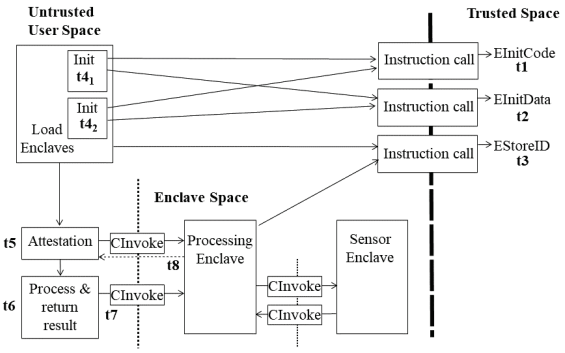

Figure 3. Benchmark setup showing measurements made for both micro and macro benchmarks.

Supporting virtual memory in ARM Morello poses additional challenges: a malicious OS at EL1 could give up ownership of an enclave capability, but then re-map another virtual address (and hence different capability) to again point to that enclave's physical memory. This issue can be overcome by either fully blocking changes to the page tables by EL1 (while the enclave is executing) and/or through appropriate MMU memory permissions and register access restrictions set at EL2. This is included in the prototype by setting bits in the hypervisor control register to block manipulation of EL1 system registers that could cause MMU changes. As the hypervisor is in control of the set up of the page tables for EL1, it is also possible to make the page table area in memory read only. However, both solutions preclude integrating the design with a rich OS using virtual memory. Yet, we note that the CHERI-TrEE design would be a promising candidate to use as the basis for a trusted OS inside ARM Trustzone, similar to the approach taken by Komodo [11], but with the shown benefits of a capability-based system.

5. Evaluation

Next, we evaluate the performance and (hardware) implementation costs of CHERI-TrEE on different platforms: our Proteus (Sections 5.1 and 5.2) and the ARM Morello simulator and the actual prototype hardware (Section 5.3).

5.1. Performance on Proteus

We conducted a number of micro and macro performance benchmarks. The micro benchmarks quantify the cost of individual instructions, while the macro benchmark measures the overhead on a full application consisting of multiple enclaves. Fig. 3 summarizes the setup of the macro benchmark and indicates the time measurements made for both micro and macro benchmarks, which we will reference in the following.

Micro benchmarks. The micro benchmarks respectively measure t1, t2 and t3 from Fig. 3. The runtime of `EInitCode` solely depends on the current occupation of the `EidTable`. Executing it with an empty table takes 4 cycles, and one extra cycle is needed for every non-empty entry at the beginning of the table. For `EInitData`, there is a fixed and a variable cost. The fixed part consists of

getting exclusive access over the pipeline (one cycle) and scanning registers for overlapping capabilities (one cycle per register). The variable cost consists of multiple parts: First, the entry corresponding to the code capability needs to be looked up in `EidTable`. Second, memory needs to be scanned for overlapping capabilities. As mentioned in Section 4.3, tag-bits are scanned at a rate of one per cycle, so to get all addresses containing capabilities, the amount of cycles needed is the memory size divided by the size of a capability (128 bits). Then, for each capability, a capability load needs to be performed, which takes 4 cycles. The last variable part is creating the enclave's identity by hashing its code section. Table 1 shows the measured performance of `EInitData`. The execution time of `EStoreId` depends on the size of the hash and the index of the hash in `EidTable`, as it has to be searched for the correct hash. For the initial enclave, we measured execution time to be 19 cycles. We did not perform micro benchmarks for enclave calls or secure communication, as these use the pre-existing `CInvoke` and `CSeal`/`CUnseal` instructions, and hence incur the same overhead as unaltered CHERI.

TABLE 1. CYCLE COUNT OF `EINITDATA` (t2). RAM STATE IS SIZE (IN KIB) PLUS NUMBER OF VALID CAPABILITIES IN MEMORY.

	Code size (B)		
RAM state	256	512	1024
128+0	8811	9271	10,191
128+100	9211	9671	10,591
256+0	17,003	17,463	18,383
256+100	17,403	17,863	18,783

Macro benchmark. To measure the performance of a more realistic application, we built a scenario where two enclaves communicate with each other (cf. Fig. 3). A "sensor enclave" provides an entry point to read the (encrypted and signed) value of a sensor, e.g., input from a camera or microphone. The "processing enclave" attests the sensor enclave, gets a sensor reading (e.g., an audio sample from a microphone), and performs some operation on the value (e.g., some filtering or similar digital processing).

To correctly initialize enclaves, untrusted code first sets up a bump memory allocator that uniquely owns a region of memory. The untrusted code then relocates the code of both enclaves to a uniquely-owned region and allocates data sections for them. It registers these using `EInitCode` and `EInitData` ($t4_1$ and $t4_2$ in Fig. 3) and invokes their initialization routines as described in Section 3.3. After sanity checks, the stack pointer is initialized and stored in the data section. As the last step, the public parts of the sign and encryption seals are derived from the seal stored by `EInitData` and returned.

Next, the processing enclave attests the sensor enclave (t5 in Fig. 3)): it provides an entry point to receive the entry capabilities and public seals of the sensor from untrusted code. To support multiple entry points, enclaves dispatch based on the value in an agreed-upon register. One specific value is used as a "return" entry point, which is invoked when returning from a call. The actual identity of the sensor enclave is fetched through `EStoreId` and compared to the precomputed identity stored in the code

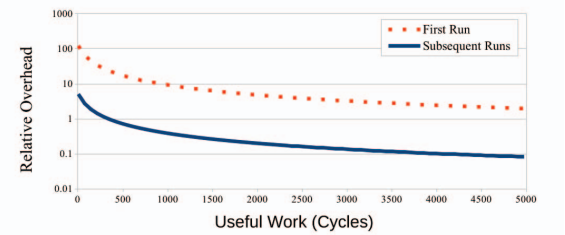

Figure 4. Relative overhead (with and without initialization and attestation) of sensor use compared to an unprotected scenario. Plotted in terms of useful work performed in the sensor enclave.

TABLE 2. TIMINGS FOR OUR MACRO BENCHMARK: "INIT" INCLUDES `EINITCODE`/`EINITDATA`, INVOKING ENCLAVE INITIALIZATION CODE AND PREPARATION (E.G., CLEARING REGISTERS). "SENSOR USE" INCLUDES THE PROCESSING TO SENSOR ENCLAVE ROUND-TRIP .

Step	Runtime (cycles)
Init processing enclave ($t4_1$)	9773
Init sensor enclave ($t4_2$)	9163
Attestation (t5)	384
Sensor use (t6)	471

section. If the identity is correct, the processing enclave stores the entry capabilities and seals of the sensor.

The actual application (t6 in Fig. 3)) starts by calling the "use sensor" entry point of the processing enclave. The processing enclave stores the nonce and the public part of its encryption seal in a buffer, leaving space for the return value, and encrypts the buffer using the sensor's encryption seal. It then calls the sensor enclave with its own entry capabilities as the return pointer. The sensor decrypts the input capability and stores a sensor reading in the buffer. In our prototype, we simulate such a device by storing an arbitrary value as sensor reading. The sensor then encrypts the capability to the buffer with the caller's encryption seal and signs the result using the methodology from Fig. 2. Upon return, the processing enclave verifies the signature and nonce and decrypts the returned capability. Then, it processes (currently simply doubling) the reading, before returning it to the caller.

We ran this scenario bare-metal on a cycle-accurate Verilator-based [30] simulation of CHERI-TrEE with 128KiB of RAM. The code size of the processing enclave is 656 bytes, while that of the sensor enclave is 336 bytes. The deterministic execution times of the different steps of our scenario are shown in Table 2. For comparison, executing an "unprotected" version of our scenario (where no enclaves are used and arguments are passed without sealing or signing) took 79 cycles. The one-time cost of initialization of enclaves makes up the bulk of the execution time. Overhead compared to the unprotected case is large, since little computation is performed in this scenario. Taking inspiration from Sancus, we therefore reran the macro benchmark for different amounts of useful work performed by the sensor enclave. Fig. 4 shows how the overhead relative to the unprotected scenario quickly decreases as the sensor enclave's useful work increases.

TABLE 3. IMPLEMENTATION RESULTS FOR THE PROTEUS PROCESSOR AND ITS VARIANTS ON THE ZYNQ ULTRASCALE+ XCZU9EG-2FFVB1156 FPGA BOARD WITH 128 KIB MEMORY. PERCENTAGES INDICATE AREA USAGE RELATIVE TO TOTAL FPGA SIZE.

Processor (128 KiB memory)	Area occupation					Operating freq. (MHz)	Dynamic power (mW)
	LUTs	Flip-flops	BRAMs	DSPs	CLBs		
Proteus	3054 (1.1%)	1663 (0.3%)	32 (3.5%)	-	694 (2.03%)	180	40
CHERI Proteus	8059 (2.94%)	3915 (0.7%)	32 (3.5%)	-	1298 (3.79%)	70	23
CHERI-TrEE	12,806 (4.7%)	7514 (1.4%)	32 (3.5%)	-	2385 (6.96%)	70	69

5.2. Hardware implementation of Proteus

To quantify the performance and resource requirements of the Proteus processor, we implemented and ran it using the Xilinx Vivado tools [31] on a Zynq UltraScale+ XCZU9EG FPGA [32]. We extended the Verilog file generated by SpinalHDL with support circuitry (e.g., for clocking) and set up constraints and pin connections for our FPGA board. We considered Proteus (without capabilities); CHERI Proteus (with capabilities); and the EES-enabled variant, CHERI-TrEE. We used common metrics to analyze the result: area occupation, operating frequency, and power consumption. Table 3 presents the results. In Section A, we also compare Proteus and variants to the CHERI Piccolo processor developed in Cambridge [33].

First, we evaluated the impact of adding CHERI and CHERI-TrEE functionality. Table 3 confirms that area occupation increases with processor complexity. CHERI Proteus uses more hardware primitives than Proteus, occupying $\approx 1.87\times$ more FPGA area in terms of Configurable Logic Blocks (CLBs). This increase in area occupation is largely caused by CHERI capabilities and tag-bits, which are implemented with distributed RAM and hence realized as LUTs. Similarly, CHERI-TrEE uses $\approx 1.84\times$ more CLBs than CHERI Proteus. The SHA256 block is a big factor in this increase. However, this block's cost can potentially be amortized, for example when it is reused to accelerate cryptographic instructions. All variants use 32 Block Random Access Memories (BRAMs), because they have the same 128 KiB memory size.

Regarding performance, the CHERI Proteus processor and its CHERI-TrEE variant decrease the maximum clock frequency by $\approx 2.57\times$ compared to Proteus. As expected, adding capabilities substantially increased the processor's complexity. This is due to the CHERI trap and exception handling circuits deployed with logical primitives on the critical path. We note that we did not specifically optimize the hardware design for maximum clock frequency; thus, substantial improvements are likely possible, e.g., by adding register stages in the critical path. Finally, CHERI Proteus has a 74% lower dynamic power consumption than Proteus as it runs at a lower clock frequency. Conversely, due to its higher resource usage, CHERI-TrEE consumes 69 mW, i.e., 72.5% more.

In summary, our Proteus core and its capability/EES variants scale well and largely independently of the memory size, requiring $\leq 7\%$ of the total area on our FPGA. We also deployed the CHERI Proteus and CHERI-TrEE variants on the low-end Arty A7-35T XC7A35TICSG324-1L FPGA. CHERI Proteus used 28.23% of the available area, while CHERI-TrEE occupied 51.57%, which shows that our design can also be implemented on small FPGAs.

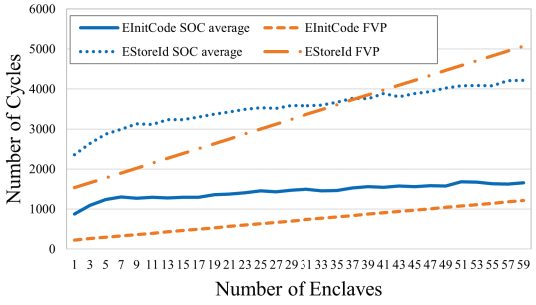

Figure 5. Measured number of clock cycles to run EInitcode and EInitData for the Sensor Enclave on Morello FVP / hardware (SOC).

5.3. Performance on ARM Morello

Micro benchmarks. The number of clock cycles for `EInitCode`, `EInitData`, and `EStoreID` at EL2 (t1, t2 and t3 in Fig. 3) was measured on both the Morello FVP and the hardware (SOC) using Arm's Performance Monitor cycle counter. The full HVC calls take longer because firstly an exception is called to change EL and then the exception handler needs to determine which HVC function to run. Measurements on the hardware were averaged over 100 runs, because we observed some variation in results between runs of the same binary, likely due to factors such as the data and instruction caches. Fig. 5 shows a graph of `EInitCode` and `EInitData` for the sensor enclave over an increasing number of enclaves. The runtime is affected by the number of table entries searched.

For `EInitData` however, the predominant cost is the memory sweep and therefore the increase in the number of clock cycles over a larger number of enclaves is relatively small. On the Morello FVP, an EL1 memory sweep by the EL2 hypervisor took about 2.16M clock cycles per MB of DRAM, corresponding to 1.1 ms/MB on a single core running at 2 GHz. This includes overlap checks of the 233 capabilities that were detected in our example. A "clean" memory sweep (no capabilities) is only marginally faster because if the tag-bit is not set there is no need for an overlap check. Given the sweep is performed on 16 bytes at a time (capabilities are aligned to 16-byte boundaries), this corresponds to approx. 33 cycles to check each capability memory location. Performance on the hardware returned similar results: 2.15M clock cycles per MB (without outputting number of overlap checks).

Macro benchmark. We ran the same macro benchmark as for Proteus. Since the initialization time of the enclaves is dominated by the memory sweep we do not repeat the

TABLE 4. EXECUTION TIMES ON MORELLO MACRO BENCHMARK.

Step	FVP (cycles)	SOC (av. cycles)
Attestation (t5)	2026	1938
Sensor use (t6)	255	206
Enclave function entry (t7)	69	114
Enclave function return (t8)	31	53

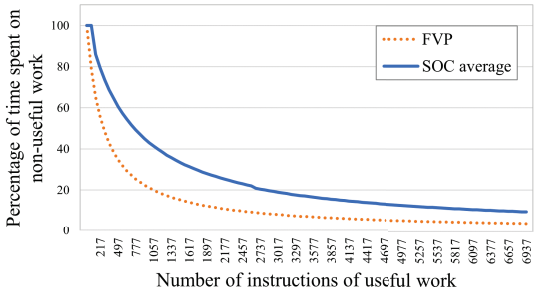

Figure 6. Percentage of time spent on non-useful work to process and return sensor data from the enclaves on Morello FVP / hardware (SOC).

figures here. Instead, Table 4 additionally measures the boundary timings for going into and out of the enclave (t7 and t8 in Fig. 3). t7 is the number of clock cycles to go from the calling function to the entry point of the desired function within the enclave and includes the invoke instruction, setting up registers and stack, and jumping to the function within the enclave. t8 is the number of clock cycles it takes to return and includes some stack and register manipulation as well as the actual return instruction. We re-ran the macro benchmark for different amounts of useful work performed by the sensor enclave, measuring the time to process and return the results (t6). Fig. 6 shows the percentage of time spent on non-useful work for increasing amounts of useful work.

While we demonstrated that CHERI-TrEE can run on a processor like Morello, the potentially larger memory size means a greater time cost of the sweep operation–especially if the large DRAM of several GB is to be fully used. However, this can be mitigated to some extent by the aforementioned grouping of enclaves during the initialisation process (cf. Section 4.4). Section 6 discusses further improvements and alternatives to the memory sweep.

6. Further extensions and future work

Being a research prototype, CHERI-TrEE allows for various extensions and follow-up work.

Going from local to remote. CHERI-TrEE currently only supports local attestation. To support remote attestation and secure remote communication, appropriate cryptographic functionality is required. For example, we could allow each enclave to govern its own keys for remote attestation, possibly in the form of a special token capability representing the key, and add a primitive to perform encryption and decryption given a capability and a key. We expect to be able to reuse a lot of the key distribution infrastructure of e.g. Sancus.

Optimizing or replacing the memory sweep. The current memory sweep is a simple approach to asserting unique ownership over memory regions, that might be acceptable for embedded systems with small memory sizes. That being said, optimizations and alternatives are possible.

The sweep can be optimized by integrating it with CHERI's hierarchical, compressed management of memory tags [34], by checking multiple adjacent tags at once rather than one tag at a time (this should improve efficiency by an order of magnitude) or by initializing multiple enclaves at the same time (as our software-based implementation on top of ARM Morello illustrates). Lastly, Esswood [35] (cf. Section 7) illustrates how a similar memory sweep can be implemented in a non-blocking way on a multi-core system by adding additional architectural registers, reducing the impact on latency.

We could alternatively have used *linear capabilities* [36], [37], [12], capabilities that cannot be duplicated, to provide exclusive access without a sweep. This would require overcoming technical concerns related to concurrency and their implementation in hardware (see [36] for a discussion). Providing the TCB with exclusive ownership over a region of memory provides similar guarantees, although this approach is less flexible, since it does not allow delegating ownership. Both approaches avoid a memory sweep when memory is first used, but would still require sweeping to reclaim said memory, and are hence most effective if new enclaves are infrequently initialized and the average size of the enclaves is small. In higher-end systems with a lot of memory (e.g., our Morello implementation), especially if virtual memory is supported, these alternative approaches become more attractive. The trade-offs involved in these alternative approaches require further study, and were hence left for future work.

Added flexibility motivates verification. A disadvantage of the flexibility highlighted in Section 3.6 is that it becomes more difficult to gain assurance over the security of the system. To clarify, the size of the TCB in our proposal is not fundamentally larger than in other systems like SGX or Sancus. However, the flexible nature of our system simply creates more potential pitfalls for enclave developers (although, as we discussed, capabilities help avoid some API-based vulnerabilities). We believe that verifying the security of such implementations will be an interesting challenge to address using formal methods.

Enclaves are in general rewarding targets for verification, since they rely on a small TCB (no language runtime or operating system is trusted). For relatively small enclaved applications on RISC processors, obtaining full-system security properties for enclaved code should already be an achievable goal, and scaling these results is interesting future work.

Supporting virtual memory. Related to the issues discussed in Section 4.4, integration of CHERI-TrEE with virtual memory in a more principled way is interesting, as it would allow CHERI-TrEE to function with a rich OS using virtual memory. One idea is to have physically-addressed capabilities, which bypass the MMU and carry permission over physical memory directly. If the operating system cannot map virtual memory to ranges covered by physically-addressed capabilities, enclaves can be secured

in the presence of virtual memory. Another idea is to restrict page table manipulation by e.g., using capabilities rather than integer addresses as page table entries.

7. Related Work

Trusted computing and capability systems have a rich history that spans decades. For trusted computing, an excellent survey is given by Parno *et al.* [38], while Maene *et al.* [39] provide a more up-to-date survey of hardware-supported systems. For capability systems, Levy [1] covers early systems, while Watson *et al.*'s paper on compartmentalization in CHERI [2] discusses more recent ones.

Our work is most related to the class of systems we called *enclaved execution systems*. Flicker [4] was the first system to propose the idea of fine-grained attestation and secure execution of small pieces of code, isolated even from malicious system software. Flicker and other early systems [40], [41] were implemented as a small hypervisor that used the late launch feature of Intel/AMD processors to start an isolated VM containing the enclave. Later systems relied on hardware extensions to avoid the use of a hypervisor, further reduce the TCB, and increase security [8], [5]. Intel SGX [16] is a commercial implementation with full support for enclaves. ARM Trustzone [42] initially only supports a single secure world, which then runs a separate operating system to support multiple trusted applications in parallel. The third major commercial system, AMD SEV, isolates complete virtual machines (rather than small enclaves) from the untrusted OS [43]. ARM's CCA [44], [45] and Intel TDX [46] now offer similar protection for entire virtual machines, respectively called *Realms* and *Trust Domains*.

The mechanisms of enclaved execution are complex, and both research prototypes for enclaved execution and commercial systems have undergone revisions: For instance, Intel SGX2 adds support for larger enclave sizes and dynamic enclave memory management. Sancus 2.0 [9] adds support for confidential loading. TrustLite [6] introduces an execution-aware memory protection unit to support more flexible allocation of memory to embedded enclaves. Tytan [7] adds support for real-time guarantees. TIMBER-V [47] enables memory sharing for fine-grained enclaves by using a tagged memory architecture.

We are not the first to observe the complexity of enclaved execution or the usefulness of making it more extensible and configurable. Keystone [10] avoids the fixed set of trade-offs in existing systems and improves customization with a framework for building enclaved execution systems. Elasticlave [48] proposed a significantly more flexible memory model for enclaves on top of Keystone and is designed to make common data sharing patterns efficient. An important difference with our approach is that, rather than designing a new access control model, we *reuse* the capability memory access control model. Similarly, Park *et al.* extend the access control model of Intel SGX with support for nested enclaves [49]. CURE [50] increases flexibility by offering different types of enclaves for various application needs. SERVAS [51] employs authenticated encryption to allow secure enclave memory sharing. Sanctum [8] shares our goal of identifying minimal hardware extensions or modifications and then combining these in software but it builds on page table-based isolation rather than capabilities as the base platform. Komodo [11] also identifies minimal hardware requirements and then implements enclave management instructions in a small trusted software monitor that is formally verified. The prototype is implemented on top of ARM Trustzone. SANCTUARY [52] uses ARM Trustzone together with the address-space controller present in some modern ARM processors to dynamically construct user-space enclaves.

Two closely related capability systems exist. First, the unpublished CAPSTONE system introduces linear capabilities along with a highly general (but potentially costly) architectural capability revocation primitive, in order to securely support various memory access models [53]. As in our work, nested enclaves and memory sharing between enclaves are supported. However, attestation is unsupported, and the implementation is limited to a sketch.

Second, Esswood's PhD thesis discusses concurrent research on the CheriOS capability operating system [35]. The goal of CheriOS is to achieve high performance in the presence of a strong adversary that includes the OS. The CheriOS microkernel is built on a trusted firmware nanokernel, which provides the security properties required to implement an EES. As in our work, remote attestation is out of scope. The nanokernel supports interrupt handling and a limited form of single-address-space virtual memory, where the OS cannot alter page tables.

To obtain unique ownership of memory, the nanokernel offers *reservations*; sealed capabilities that represent a right to uniquely allocate a region of memory. Non-allocated Reservations can be used to create a type of enclaves, called *foundations*, similar to our design. The foundation owns an *authority token*, an object capability analogue to our sealing capabilities, which is tied to the foundation's identity. It can be used to perform both symmetric and asymmetric signing and encryption through the nanokernel, as opposed to our sealed capabilities. In conclusion, both approaches are sufficiently flexible to allow for growing/nested enclaves and memory sharing between enclaves. The work of Esswood *et al.* involves a larger TCB and more software-level abstractions, leading to overhead, but allows for greater flexibility with respect to revocation and a larger sealing space. The exact implications of these trade-offs need to be investigated further.

8. Conclusion

In this paper, we investigated how enclaves can be built on a capability machine like CHERI. Implementing enclaves without duplicating existing functionality was only possible by decomposing the concept of an EES into a set of orthogonal features. This decomposition made it clear that we can reuse CHERI's existing features and only need to add a limited amount of new features to obtain an expressive EES that performs well in many respects (see Section 5). Even better, the resulting design is more flexible in important ways (growing/nested/non-contiguous enclaves, sharing memory, two-way sandboxing, dynamic entry points, etc.). In addition to the design itself, we believe our decomposition of EESs is useful to analyse the design space and inform future designs.

Data Availability

As mentioned in the introduction, all three of our implementations and their benchmarks have been made open source. Snapshots of these implementations along with instructions on how to install them can be found under https://github.com/proteus-core/cheritree.

Acknowledgements

This research was partially funded by the Research Fund KU Leuven, by the Flemish Research Programme Cybersecurity, by the Research Foundation - Flanders (FWO; G030320N), by a European Research Council (ERC) Starting Grant (UniversalContracts; 101040088), funded by the European Union, and the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/V000454/1 and EP/S030867/1. The results feed into DsbDtech. Thomas Van Strydonck held a PhD Fellowship of the Research Foundation - Flanders (FWO) during the work on this project. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council.

References

[1] H. M. Levy, *Capability-Based Computer Systems*. Digital Press, 1984. [Online]. Available: https://homes.cs.washington.edu/~levy/capabook/

[2] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 20–37.

[3] ARM, "ARM Architecture Reference Manual Supplement Morello for A-profile Architecture," Tech. Rep., 2020.

[4] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: an execution infrastructure for TCB minimization," in *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, J. S. Sventek and S. Hand, Eds. ACM, 2008, pp. 315–328.

[5] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base." in *USENIX Security Symposium*, 2013, pp. 479–494.

[6] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *European Conference on Computer Systems*, ser. EuroSys '14. ACM, Apr. 2014, pp. 1–14.

[7] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Design Automation Conference*, ser. DAC '15. Association for Computing Machinery, Jun. 2015, pp. 1–6.

[8] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[9] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A low-cost security architecture for IoT devices," *ACM Trans. Priv. Secur.*, vol. 20, no. 3, pp. 7:1–7:33, Jul. 2017.

[10] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: an open framework for architecting trusted execution environments," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds. ACM, 2020, pp. 38:1–38:16.

[11] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 287–305.

[12] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-951, Oct. 2020. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf

[13] H. Xia, J. Woodruff, H. Barral, L. Esswood, A. Joannou, R. Kovacsics, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Richardson, S. W. Moore, and R. N. M. Watson, "CheriRTOS: A Capability Model for Embedded Devices," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct. 2018, pp. 92–99.

[14] H. Almatary, M. Dodson, J. Clarke, P. Rugg, I. Gomes, M. Podhradsky, P. G. Neumann, S. W. Moore, and R. N. M. Watson, "CompartOS: CHERI compartmentalization for embedded systems," 2022. [Online]. Available: https://arxiv.org/abs/2206.02852

[15] Microsoft Security Response Center, "What's the smallest variety of CHERI?" https://msrc-blog.microsoft.com/2022/09/06/whats-the-smallest-variety-of-cheri/, [Online; accessed 27-10-2022].

[16] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016. [Online]. Available: http://eprint.iacr.org/2016/086

[17] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware Support for Fast Capability-based Addressing," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1994, pp. 319–327. [Online]. Available: http://doi.acm.org/10.1145/195473.195579

[18] A. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996. [Online]. Available: http://cacr.uwaterloo.ca/hac/

[19] J. Woodruff, A. Joannou, H. Xia, A. C. J. Fox, R. M. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. W. Moore, "CHERI concentrate: Practical compressed capabilities," *IEEE Trans. Computers*, vol. 68, no. 10, pp. 1455–1469, 2019. [Online]. Available: https://doi.org/10.1109/TC.2019.2914037

[20] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. Association for Computing Machinery, 2019, p. 1741–1758. [Online]. Available: https://doi.org/10.1145/3319535.3363206

[21] J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens, "Secure resource sharing for embedded protected module architectures," *9th WISTP International Conference on Information Security Theory and Practice (WISTP'15)*, vol. 9311, pp. 71–87, 2015. [Online]. Available: https://doi.org/10.1007/978-3-319-24018-3

[22] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 409–420. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_yanlin

[23] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Krügel, and G. Vigna, "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments," in *NDSS*, 2017.

[24] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume I: Base user-level ISA," Tech. Rep., 2011.

[25] A. Armstrong, T. Bauereiss, B. Campbell, S. Flur, J. French, K. E. Gray, G. Kerneis, N. Krishnaswami, P. Mundkur, R. Norton-Wright, C. Pulte, A. Reid, P. Sewell, I. Stark, and M. Wassell, "The Sail instruction-set architecture (isa) specification language," 2013–2019.

[26] "Sail-CHERI-RISC-V, CHERI-RISC-V model in Sail," https://github.com/CTSRD-CHERI/sail-cheri-riscv.

[27] C. Papon, "VexRiscv, A FPGA friendly 32 bit RISC-V CPU implementation," https://github.com/SpinalHDL/VexRiscv.

[28] ——, "SpinalHDL, A Scala based HDL," https://github.com/SpinalHDL/SpinalHDL.

[29] "SpinalHDL, Cryptography libraries," https://github.com/SpinalHDL/SpinalCrypto.

[30] "Verilator, the fastest Verilog/SystemVerilog simulator," https://www.veripool.org/verilator/.

[31] Xilinx, "Vivado," https://www.xilinx.com/products/design-tools/vivado.html, 2021, [Online; accessed 13-07-2021].

[32] ——, "Zynq UltraScale+ MPSoC Data Sheet," https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf, 2021, [Online; accessed 20-07-2021].

[33] R. N. M. Watson, S. W. Moore, P. Sewell, and P. Neumann, "Capability Hardware Enhanced RISC Instructions (CHERI)," https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/, 2021, [Online; accessed 20-07-2021].

[34] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos, "Efficient Tagged Memory," in *IEEE International Conference on Computer Design (ICCD)*. IEEE, Nov. 2017.

[35] L. Esswood, "CheriOS: Designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor," Ph.D. dissertation, University of Cambridge, 2020.

[36] L. Skorstengaard, D. Devriese, and L. Birkedal, "StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290332

[37] T. Van Strydonck, F. Piessens, and D. Devriese, "Linear capabilities for fully abstract compilation of separation-logic-verified code," *Proc. ACM Program. Lang.*, vol. ICFP, 2019.

[38] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping trust in commodity computers," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 414–429.

[39] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede, "Hardware-Based Trusted Computing Architectures for Isolation and Attestation," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 361–374, Mar. 2018.

[40] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig, "Trustvisor: Efficient TCB reduction and attestation," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 143–158.

[41] R. Strackx and F. Piessens, "Fides: selectively hardening software application components against kernel-level or process-level malware," in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM, 2012, pp. 2–13.

[42] T. Alves and D. Felton, "TrustZone: Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, pp. 18–24, 2004.

[43] AMD, "AMD SEV-SNP: strengthening VM isolation with integrity protection and more," *White paper*, Jan. 2020.

[44] Arm Ltd., "Arm Confidential Compute Architecture," https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, [Online; accessed 27-10-2022].

[45] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell, "Design and verification of the arm confidential compute architecture," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 465–484.

[46] Intel, "Intel Trust Domain Extensions," https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf, [Online; accessed 27-10-2022].

[47] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A. Sadeghi, "TIMBER-V: tag-isolated memory bringing fine-grained enclaves to RISC-V," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[48] J. Z. Yu, S. Shinde, T. E. Carlson, and P. Saxena, "Elasticlave: An efficient memory model for enclaves," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/yu-jason

[49] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh, "Nested enclave: Supporting fine-grained hierarchical isolation with sgx," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 776–789.

[50] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "CURE: A security architecture with CUstomizable and resilient enclaves," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1073–1090. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/bahmani

[51] S. Steinegger, D. Schrammel, S. Weiser, P. Nasahl, and S. Mangard, "Servas! secure enclaves via risc-v authenticryption shield," in *European Symposium on Research in Computer Security*. Springer, 2021, pp. 370–391.

[52] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with User-space Enclaves," in *NDSS*, 2019.

[53] J. Z. Yu, C. Watt, A. Badole, T. E. Carlson, and P. Saxena, "Capstone: A capability-based foundation for trustless secure memory access (extended version)," 2023. [Online]. Available: https://arxiv.org/abs/2302.13863

[54] Bluespec, "Bluespec company," https://bluespec.com/, 2021, [Online; accessed 20-07-2021].

[55] "Tools for BlueSpec HDL," https://github.com/B-Lang-org/bsc.

Appendix A. Comparison to Piccolo

To compare our design to other RISC-V cores, we resort to the Bluespec Piccolo core; to our knowledge, the only other 32-bit processor implementation providing CHERI capabilities. Table 5 details the main modules of each core.

The non-capability Piccolo is developed by Bluespec [54], while the CHERI Piccolo variant is developed by Cambridge University [33], both designed in a high-level hardware description language, Bluespec HDL [55]. Note that neither Piccolo variant implements extensions for trusted execution. Therefore, we can mainly compare the different CHERI implementations. Secondly, Piccolo

TABLE 5. IMPLEMENTATION RESULTS FOR THE PROTEUS AND PICCOLO PROCESSORS AND THEIR VARIANTS ON THE ZYNQ ULTRASCALE+ XCZU9EG-2FFVB1156 FPGA, DEPICTING THE MAIN MODULES WITHIN EACH CORE.

Processor (128 KiB memory)		Area Occupation					Operating freq. (MHz)	Dynamic power (mW)
		LUTs	Flip-flops	BRAMs	DSPs	CLBs		
Proteus	Machine Timers	32	128	-	-	48	180	40
	Pipeline module	2867	1410	-		641		
	Memory	-	-	32		-		
	Total	3054 (1.1%)	1663 (0.3%)	32 (3.5%)		694 (2.03%)		
CHERI Proteus	Machine Timers	80	128	-	-	37	70	23
	Pipeline module	7797	3662	-		1260		
	Capability Register File	161	-	-		21		
	Memory	-	-	32		-		
	Total	8059 (2.94%)	3915 (0.7%)	32 (3.5%)		1298 (3.79%)		
CHERI-TrEE	Machine Timers	80	128	-	-	40	70	69
	Pipeline module	12,510	7261	-		2353		
	Capability Register File	160	-	-		20		
	SHA module	3422	3193	-		839		
	Memory	-	-	32		-		
	Total	12,806 (4.7%)	7514 (1.4%)	32 (3.5%)		2385 (6.96%)		
Bluespec Piccolo	CPU module	4807	2583	3	15	835	140	68
	Debug module	292	405	-	-	183		
	PLIC+CLINT	1810	1194	-	-	504		
	Memory	-	-	33	-	-		
	Total	10,152 (3.7%)	7986 (1.4%)	36 (3.9%)	15 (0.6%)	1942 (5.67%)		
CHERI Piccolo	CPU module	18,347	6464	5	15	3471	120	194
	Debug module	471	402	-	-	176		
	PLIC+CLINT	1840	1146	-	-	467		
	Tag Controller	4349	2124	36	-	966		
	Memory	-	-	33	-	-		
	Total	26,685 (9.7%)	11,533 (2.1%)	74 (8.1%)	15 (0.6%)	4865 (14.2%)		

is a commercial-grade processor with additional functionality, while Proteus is currently a research design. Nonetheless, we implemented both Piccolo and CHERI Piccolo on the same FPGA as Proteus and used 128 KiB memory in all tests. As Proteus is an RV32IM, we configured the Piccolo variants to the same architecture (by default, Piccolo is an RV32ACIMU).

Comparing CHERI Proteus and CHERI Piccolo, our processor uses $\approx 3.31\times$, $\approx 2.95\times$, and $\approx 2.31\times$ fewer LUTs, flip-flops, and BRAMs, respectively, while not requiring dedicated DSP blocks. Different from CHERI Piccolo, CHERI Proteus does not implement CHERI compressed capabilities and uses BRAMs for the instruction and data memory, using considerably fewer resources. The dynamic power consumption is $\approx 8.43\times$ less than CHERI Piccolo. Similarly, for the non-capability versions, Proteus occupies $\approx 3.32\times$, $\approx 4.80\times$, and $\approx 1.12\times$ fewer LUTs, flip-flops, and BRAMs than Bluespec Piccolo. The dynamic power consumption is $1.7\times$ less.

The higher resource usage of both Piccolo variants compared to Proteus can be partially attributed to higher circuit complexity due to additional components, e.g., debug circuitry and interrupt controllers. Besides, CHERI Piccolo uses BRAMs to implement its (relatively large) Tag Controller module. Finally, as mentioned, CHERI Proteus does not currently implement compressed capabilities.

Regarding clock frequency, Proteus reaches 180 MHz, $\approx 1.29\times$ faster than Bluespec Piccolo. However, CHERI Proteus is $\approx 1.71\times$ slower than CHERI Piccolo, which could be due to the fact that Piccolo is already optimized for real-world deployment.

Besides, considering only the actual CPU module of Piccolo cores detailed in Table 5, our Proteus processors use significantly fewer resources than Piccolo: the non-capability Proteus uses $\approx 1.2\times$ fewer CLBs than Bluespec Piccolo, while CHERI Proteus uses $\approx 2.67\times$ fewer CLBs than its CHERI variant. Despite the additional EES features, we note that CHERI-TrEE also uses fewer resources than CHERI Piccolo. In summary, this comparison shows that even though Proteus is currently a research prototype and has not gone through extensive optimization cycles, it offers a smaller size and adequate performance compared to the commercially maintained Piccolo. In addition, our modular design lends itself to the easy addition of functionality at an acceptable cost, as evidenced by the CHERI-TrEE implementation on Proteus.