IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 2, pp. 268–292. DOI:10.46586/tches.v2025.i2.268-292

CHERI-Crypt: Transparent Memory Encryption on Capability Architectures

Jennifer Jackson[1], Minmin Jiang[2] and David Oswald[3]

[1] University of Birmingham, Birmingham, UK, j.jackson@bham.ac.uk
[2] University of Birmingham, Birmingham, UK, m.jiang@bham.ac.uk
[3] University of Birmingham, Birmingham, UK, d.f.oswald@bham.ac.uk

Abstract. Capability architectures such as CHERI (Capability Hardware Enhanced RISC Instructions) are an emerging technology designed to provide memory safety protection at the hardware level and are equipped to eradicate approximately 70% of the current software vulnerability attack surface. CHERI is an instruction set architecture extension and has been applied to a small number of processors, including various versions of RISC-V. One of the benefits of CHERI is that it inherently provides segregation or compartmentalisation of software, making it suitable for supporting other types of applications such as Trusted Execution Environments, where sensitive data and computation is conducted inside a secure enclave, away from the rest of the untrusted operating system and services. To prevent untrusted software from accessing these compartments or secure regions of memory CHERI uses the mechanism of sealed capabilities. Trusted execution environments however, have been proven vulnerable to not just software-based attacks, but hardware attacks as well. In this paper we present our CHERI-Crypt design, an encryption engine extension to a CHERI-RISC-V 32-bit processor, for transparent memory encryption of sealed CHERI capabilities to additionally protect sensitive data in memory against physical hardware attacks. We show that our CHERI-Crypt design can run an enclave test program within an encrypted CHERI seal and invoke process, requiring 626 additional clock cycles with a batch size of 32 bytes. Adding CHERI-Crypt reduces the maximum frequency of the base CPU by only 6 MHz, and requires approximately 3.5× more flip flops and LUTs.

Keywords: Memory Encryption · CHERI · RISC-V · Capability Architectures · Confidential Computing · Trusted Execution Environments

1 Introduction

1.1 CHERI Capability Architectures

CHERI (Capability Hardware Enhanced RISC Instructions) (CHERI) [WNW+23] extends Instruction-Set Architectures (ISAs) with architectural capabilities to provide memory safety by the hardware and allow fine grained code protection against approximately 70% [JEA20] of common memory vulnerabilities such as buffer overflows. CHERI capabilities extend traditional pointers with additional information stored in the hardware relating to their bounds and permissions to control fine grained access to memory. Along with this additional information is a validity tag bit which maintains their integrity. When an illegal operation is performed, such as writing beyond the bounds of a capability or de-referencing a NULL capability, a hardware exception is raised and the program execution is halted before the problem can be exploited. The CHERI model can be mapped into architectures in different ways. For example capabilities may be stored in memory in compressed or

Received: 2024-10-15 Accepted: 2024-12-15 Published: 2025-03-04

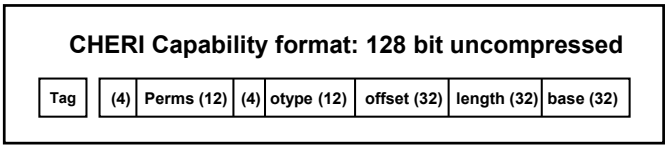


Figure 1: Example capability fields for an uncompressed 128 bit format for a 32-bit architecture.

uncompressed format and the bit width of the information fields may differ between architectures. The 128 bit CHERI concentrate [WJX+19] compressed format model for 64 bit architectures is increasingly being adopted, however 128 bit uncompressed formats for 32 bit architectures as shown in Figure 1 is still useful for low complexity small embedded devices, and where experimental research features can easily be added. In this format the *permissions* (perms) field governs how the memory that the capability is pointing to can be used, for example whether it can perform loads and stores or execute code. The *object type* (otype) field indicates whether a capability is sealed by containing a unique identifier (see Section 1.2), and consequently cannot be de-referenced. The *base* is the lower bound of the capability. The *length* is the length of the memory section, where the sum of base and length is the upper bound of the capability. The *offset* is the offset from the base, where the sum of base and offset gives the address the capability currently points to.

1.2 Compartmentalisation, Sealed Capabilities, and TEEs

One of the benefits of CHERI is that it inherently provides *compartmentalisation* of software or data, making it suitable for supporting applications beyond the fine grained memory safety of preventing vulnerabilities such as buffer overflows. To prevent untrusted software from directly accessing these compartments of memory CHERI uses the mechanism of *sealed capabilities* [WWN+15]. A sealed capability is a capability that has been prevented from directly accessing the area that it is pointing to. In software terms, this means that the capability cannot be de-referenced. A capability can be sealed and unsealed by a *sealing capability* containing an object identifier within its base field, which can be thought of as a key to lock and unlock a memory space.

This mechanism can be useful in various ways, e.g. for isolating software such as external libraries from the rest of the code and reducing the potential attack surface, or it can be used as a representative method of *encryption* where the sealing capability is used as an encryption key to access reads and writes to memory without physically encrypting the data. Additionally, sealed capabilities are an ideal framework for *enclaved execution*, a type of Trusted Execution Environment (TEE), where sensitive data and computation is conducted inside a secure enclave, away from the rest of the untrusted operating system and services. Previous CHERI-enclave work such as CHERI-TrEE [VSNJ+23] has shown that a trusted application sealed in memory can be invoked through a secure CHERI unsealing process. A *pair of sealed capabilities* pointing to the trusted code and data sections can be accessed through a special CHERI *invoke* instruction. The CHERI invoke instruction performs a domain transition off to an isolated piece of code and data, and then returns. In CHERI-TrEE, the transitioned domain resides within a trusted and carefully managed memory space. Part of this management involves ensuring there are no overlapping capabilities pointing to the enclave memory. This is achieved by a memory sweep operation over the whole of the memory space every time an enclave is initialised.

1.3 Transparent Memory Encryption of Sealed Capabilities

Relying on sealed and sealing capabilities alone for symbolic "encryption" (as in [VSNJ+23]) and other controlled memory admittance from the software side can leave the system vul-

nerable to other forms of unwanted physical memory access. For example TEEs have been proven vulnerable to not just software-based attacks, but also hardware attacks [CVM+21], including those targeting the memory bus [LJF+20]. Therefore, the need for *physically encrypting sealed capabilities* in memory can protect the data from physical hardware attacks as well as protect the data from software attacks by sealing. In addition to this, control over fine grained memory access with capabilities allows different memory areas to be encrypted with different keys. Encrypting with different keys (through object types) is useful in preventing accidental or adversarial overlapping capabilities from gaining access to sensitive data, either during current use, or from past use of the same memory space if it has not been cleared effectively. The only alternatives to this overlapping problem that have been proposed are a full memory sweep [VSNJ+23], and linear capabilities [Lip19] but linear capabilities are difficult to implement in practice. We propose that the concept of transparent memory encryption can be achieved by seamlessly integrating the physical encryption process into the sealing and unsealing operations of the CHERI instructions.

1.4 Our Contribution

In this paper we present our CHERI-Crypt design, a Memory Encryption Engine (MEE) extension to a CHERI-RISC-V 32-bit processor for *transparent memory encryption* of sealed CHERI capabilities to additionally protect sensitive data in memory against physical attacks. The encryption engine can seamlessly encrypt data behind capabilities during a seal operation and additionally decrypt and encrypt capabilities on-the-fly when running invoked enclave code. Our main contributions of this paper are:

1. This work presents, for the first time, how transparent memory encryption and decryption meaningfully interacts with CHERI capabilities, in particular sealing. To this end, we create two new instructions `CSealEncrypt` and `CInvokeEncrypt` to demonstrate the encryption operations, with the intention to eventually roll them into the existing CHERI `CSeal` and `CInvoke` instructions.
2. We implement our design as a 128-bit AES-Galois Counter Mode (GCM) encryption engine extension to a CHERI-RISC-V 32-bit processor using the high-level Spinal HDL language.
3. We show that our CHERI-Crypt design can run with a small enclave test program as part of an encrypted CHERI seal and invoke process. To encrypt and run the enclave with an encryption data batch size of 32 bytes requires 626 additional clock cycles. Adding CHERI-Crypt reduces the maximum frequency of the base CHERI-RISC-V CPU by only 6 MHz, and requires approximately 3.5x more flip flops and LUTs.

The source code and additional material for CHERI-Crypt is available at: https://github.com/cap-tee/cheri-crypt

Paper Organisation The rest of the paper is organised as follows: In Section 2 we discuss background and related work, in Section 3 we outline the CHERI-Crypt concept, in Section 4 we detail the design of CHERI-Crypt on a CHERI-RISC-V 32-bit processor, in Section 5 we present benchmarking results, and finally in Section 6 we conclude.

2 Background and Related Work

Trusted Execution Environment A TEE provides a secure and isolated processor area where sensitive computation can occur without interference or observation by untrusted parts of the system. By using encryption, a TEE creates a clear boundary between trusted and untrusted parties, protecting the data processed in the TEE even if the raw content of the memory is tampered with. Intel Software Guard Extensions (SGX) [Int15], a

prominent implementation of TEE, was launched in 2015. SGX is a set of instructions built into certain Intel architectures that enables the creation of hardware-enforced encrypted memory regions known as *enclaves*, where code and data are protected from external access, even from a compromised Operating System (OS) or hypervisor. However, SGX remains vulnerable to Side-Channel Attacks (SCAs) and its performance overhead poses a burden on the system, highly dependent on the frequency of enclave boundary crossings. The original MEE for SGX used a Merkle tree and counters to prevent memory replay attacks, limiting the memory available to an enclave. In subsequent revisions, SGX-Scalable and Trust Domain Extensions (TDX) [Int23b, Int23a], Intel removed this aspect in favour of large enclaves and higher performance.

In 2016, AMD introduced Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) [KPW21] to protect memory from physical access. SME encrypts the entire system memory through a dedicated hardware encryption engine that utilises a single key. This approach offers a general protection for the system memory, making it suitable for on-premise servers or personal devices. SEV is an extension of SME, specially tailored to protect virtualised environments by encrypting each Virtual Machine (VM) memory using a unique key, isolating them from each other, as well as the hypervisor. The keys are protected and distributed by a dedicated secure processor. Although each VM memory is encrypted and isolated from each other, the data structure that stores some critical information such as VM states and configurations, the communication methods with the hypervisor remains unencrypted, which can be exploited to gain control of the VM execution [HB17]. SEV-Encrypted State (SEV-ES) [Kap17], introduced in 2017, mitigates such attacks by splitting the unencrypted data structure into two parts: one accessible by the hypervisor containing necessary control information for VM management, and the other, protected from hypervisor, containing all remaining information with encryption applied when VM exits. However, due to the lack of integrity protection in SEV-ES, arbitrary code can be injected into a SEV-ES secured VM by reuse existing ciphertext to create an encryption oracle and thereby break Xor-Encrypt-Xor (XEX)-based encryption [WWME20]. To address this, AMD introduced SEV-Secure Nested Paging (SEV-SNP) [AMD20], another variant of SEV that offers memory integrity protection through Reverse Map Table (RMT), enabling detection of malicious memory modifications and preventing attacks like replay attacks.

Memory Encryption Engines Hardware memory encryption primarily focuses on protecting data confidentiality/authenticity from hardware attacks. Yan *et al.* use GCM with a split-counter scheme to reduce per-block counter size, improving performance and security [YEP+06]. Memory encryption has subsequently been extended to smart cards, Internet of Things (IoT) devices, and so on [HT14]. For example, Counter (CTR) and XEX modes with AES and PRESENT block ciphers for securing Non-Volatile Memory (NVM) on smart cards were proposed in [EKY11]. "MemEnc" offers a hardware-based, lightweight, and low-latency memory encryption solution in resource-constrained IoT devices without OS intervention [GJC21]. Transparent encryption and authentication pipelines are also applied to modern System-on-Chips (SoCs) implemented on Field Programmable Gate Arrays (FPGAs), using ciphers like PRINCE, AES, and ASCON to protect and validate hardware and software Intellectual Property (IPs) at power-off and boot-up stages, as well as sensitive data during runtime [WUS+17]. To address the limitation of coarse granularity provided by memory encryption, an approach named "MEMES" was introduced that allows fine-grained sub-page memory encryption on existing hardware to mitigate heap memory vulnerabilities [SSL+23]. Recently, "Voodoo" introduced a combined scheme for authenticated encryption, Dynamic Random-Access Memory (DRAM) error correction, and memory tagging, reducing complexity and overhead while ensuring data integrity, confidentiality, and runtime security with minimal performance impact [LUSM24].

CHERI-RISC-V Cores and SoCs CHERI was initially integrated into the 64-bit Microprocessor without Interlocked Pipelined Stages (MIPS) architecture in 2014, with a QEMU CHERI-MIPS implementation developed for ISA-level simulation. Subsequently, the University of Cambridge extended CHERI to 32-bit RISC-V architecture (3-stage in-order pipeline CHERI-Piccolo) and 64-bit RISC-V architecture (5-stage in-order pipeline CHERI-Flute, superscaler out-of-order CHERI-Toooba). In collaboration with ARM, CHERI was integrated into ARM architecture in 2014, including the 32/64-bit Armv8-A core, and extended to a general-purpose ARM architecture (Neoverse N1) under the research program known as Morello [ARM19], where the Morello board was released in early 2022. Last year, Microsoft integrated CHERI into the 32-bit Ibex RISC-V core, designed by LowRISC, creating a (Capability Hardware Extension to RISC-V for Internet of Things) (CHERIoT) to provide security protection for low-cost embedded systems [ACC+23a] [ACC+23b].

CHERI-RISC-V cores can be added into various systems, such as BESSPIN Government Furnished Equipment (BESSPIN-GFE) [Blu20], to form SoCs and then be implemented on Field Programmable Gate Array (FPGA) boards (e.g., Xilinx). BESSPIN-GFE provides an evaluation platform for testing the effectiveness of hardware security architectures. The morello board, featuring the Morello SoC, is the most prominent CHERI-supported SoC to date, designed specially for research and development. These setups allow for developing and evaluating CHERI features within hardware environments. Although the progress of CHERI is primarily academic, some industrial companies, such as Codasip [Cod24], have shown strong interest in using CHERI to provide secure processors for their customers.

AES-GCM Authenticated Encryption Integrating an encryption engine into general-purpose processors always requires additional memory space and design complexity leading to an area/performance overhead. Since transparent memory encryption operates at the hardware level, it does not require any modifications to the applications running within the TEE. Like the XEX-based encryption engines applied to AMD SME, SEV, and SEV's variants, the ciphertext remains the same length as the plaintext, where no additional memory space is required. For Intel SGX, storing the long tag bits generated by the standard Message Authentication Code (MAC) algorithm expands the memory usage, which increases hardware cost.

In this context, the Advanced Encryption Standard - Galois/Counter Mode (AES-GCM) algorithm [MV07] offers an efficient, robust, and proven solution that ensures data confidentiality and authenticity. The AES-GCM algorithm is a widely used cryptographic method that provides both encryption and authentication in a single process. It operates by encrypting data with the AES block cipher in counter mode, while also generating an Authentication Tag (AT) using a Galois field multiplication to ensure data authentication. AES-GCM can also benefit significantly from efficiency, as it performs encryption and authentication in a parallelizable style, reducing performance overhead compared to separate algorithms. Additionally, authentication prevents unauthorized tampering, such as fault injection attacks. The tradeoff is that AES-GCM requires an Initialization Vector (IV) and generates an AT for each batch of data, which must be stored. Furthermore, to guarantee data remains encrypted in external memory, some form of temporary internal storage is required to hold a decrypted batch of data while it is being used.

3 Concept

3.1 Adversary Model

Following the adversary model commonly employed in standard TEE attack scenarios, we assume that the attacker has full software `root` access to the system. Additionally, we consider that the attacker has physical access to the external memory and can monitor

data on the memory bus between the processor and the external memory, as shown in Figure 2. We further assume that the adversary can tamper with the bus data to some extent, but exclude full replay of memory contents by the adversary from our threat model. We note that these assumptions are similar to commercial TEEs like SGX-Scalable, TDX, and SEV-SNP, all of which do encrypt but do not cryptographically verify the freshness of memory contents.

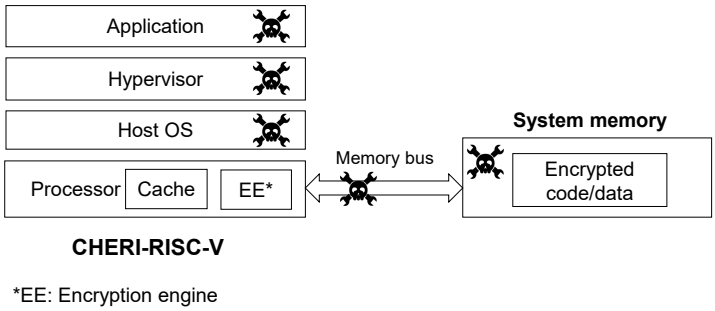


Figure 2: Adversary model: the attacker has `root` privileges on the processor, and in addition physical access to the memory bus.

We do not consider physical attacks like fault injection and side-channel analysis. We also exclude cache attacks and related microarchitectural vulnerabilities in the TEE code, as these are orthogonal problems with different mitigations. In Section 6, we compare the susceptibility of CHERI-Crypt w.r.t. different attack vectors with widely used TEEs.

3.2 CHERI-Crypt: Transparent Memory Encryption Concept

To show that the concept of transparent memory encryption is possible for a CHERI enclaved execution scenario it is necessary to focus on the two main CHERI instructions, `CSeal` and `CInvoke`, which need to be modified to support encryption and decryption of the code and data sections of an enclave. We demonstrate memory encryption through a basic seal and invoke process which form the building blocks of a more complex TEE.

3.2.1 The Basic CHERI Seal and Invoke Process

Before outlining the memory encryption concept, we briefly describe the seal and invoke process as pictured in Figure 3 which can be called in sequence to perform a basic domain transition to an isolated piece of code such as an enclave. (1) First, two capabilities, C0 and C1, are created pointing to a pre-loaded code and data memory section. (2) A second capability, C2, is created called a sealing capability with an otype value loaded into the base field of the capability. (3) A CHERI seal instruction, `CSeal`, is used to seal both capabilities, C0 and C1, with the same otype using the sealing capability C2. During this process the otype is copied from C2 to the otype fields of C0 and C1. (4) A CHERI invoke instruction, `CInvoke`, performs the domain transition. If the otype of the code and data capabilities are the same, the code capability is passed to the program counter capability, PCC, and unsealed, and the data capability is passed to the data capability register, C31, and unsealed. Enclave code is then run starting from the program counter address.

3.2.2 Modification to the Seal and Invoke Instructions

To perform transparent memory encryption we create two new instructions `CSealEncrypt` and `CInvokeEncrypt` to demonstrate the encryption operations, with the intention to eventually roll them into the existing CHERI `CSeal` and `CInvoke` instructions. With

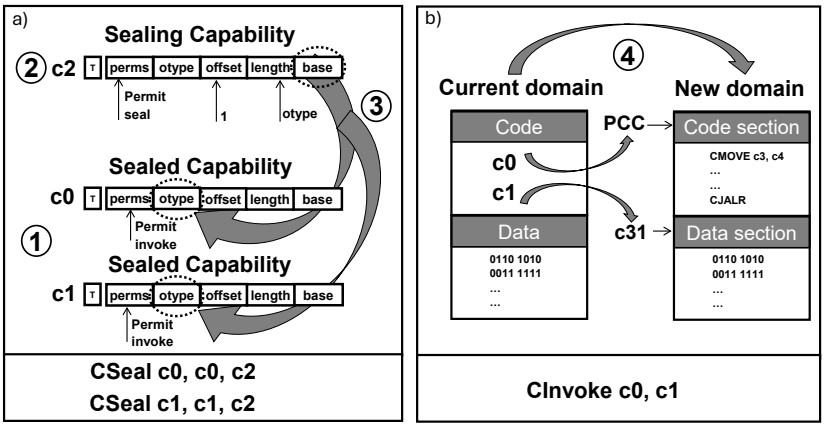


Figure 3: (a) Sealing two capabilities with the same otype, (b) The invoke process.

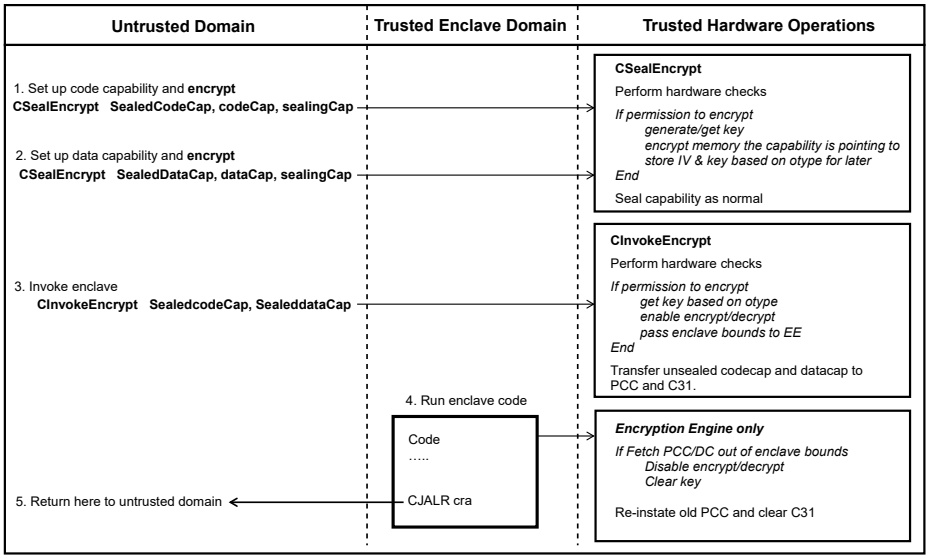


Figure 4: Seal and Invoke process with added encryption hardware operations.

this intention, we use an extra hardware permission bit to indicate whether encryption is selected. The CHERI specification defines two types of permissions: *hardware permissions* and *software permissions*. As shown in Figure 1, there are 12 bits representing the hardware permissions leaving 4 spare bits that can be used for software-defined permissions. We transfer one of these software permission bits into a hardware bit, resulting in 13 hardware permissions and 3 software permissions.

Figure 4 shows the hardware operations performed by the seal and invoke instructions together with the additions (shown in italics) necessary to achieve transparent memory encryption for the basic seal and invoke process. The new `CSealEncrypt` instruction format follows the CHERI `CSeal` instruction except that if encryption takes place, as determined by the permission bit, the output capability is a *resized* sealed capability whose memory content is encrypted. As discussed previously two `CSealEncrypt` instructions are performed to seal and encrypt a code and a data section. If encryption is permitted the pipeline is stalled, a key is generated based on the otype, the contents of the memory is read, encrypted, and then written back into the same location. At this stage we make some assumptions: we assume the AT and IV will be stored in memory immediately following

the data and therefore assume the passed capability is large enough to accommodate these, as well as the actual encrypted data (see Section 4.2). The bounds of the returned sealed capability is then reduced to cover only the encrypted data. We also assume the encryption engine provides two assurances associated with key generation: (1) a new encryption key is generated for each new otype, and (2) once two seal instructions have been performed with the same otype, a new encryption key is generated. This removes the potential issue associated with having a repeated otype to seal a different capability pair.

The new `CInvokeEncrypt` instruction follows the `CInvoke` instruction format. Additionally a hardware check tests that both the code and data capability encryption permission bits are set the same, otherwise a hardware exception will result. If encryption is not required the instruction will unseal and complete the domain switch straight away as normal. Otherwise the pipeline is delayed whilst the encryption key and IV is obtained based on the otype, and the bounds of the enclave are passed to the encryption engine, which is then enabled. After the `CInvokeEncrypt` instruction completes, decryption/encryption of the enclave code and data is performed until the PCC falls outside the bounds of the enclave. At this point the encryption circuits are disabled and registers are cleared.

4 CHERI-Crypt Design and Implementation

We implement the transparent memory features on Proteus [BNP23, Pro23], a RISC-V 32 bit processor developed under the Spinal HDL language that has been extended with the CHERI instruction set. We chose this five stage (*fetch, decode, execute, memory, writeback*) pipelined processor because it can be extended readily using a plug-in feature, which is used to add the new instructions and encryption circuitry. Also the uncompressed 128 bit capability format it uses as given in Figure 1, allows the experimental proof-of-concept features to be added without additional complexity.

4.1 Transparent Memory Encryption Engine: Outline Design

Figure 5 shows the extended modifications to the Proteus core: the basic blocks of CHERI-Crypt and the transparent memory encryption engine. The CHERI-RISC-V-32 five-stage pipeline is shown together with the *CHERI Memory Tagger and Logic*. The Tagger component sits in the path of the data bus and stores and retrieves CHERI tag bits as capabilities are loaded and stored to memory. The *data bus* from the *Memory* stage and the *instruction bus* from the *Fetch* stage interface to the *memory* over the AXI interconnect.

The new `CSealEncrypt` and `CInvokeEncrypt` instructions (see Section 3.2.2) are processed from the memory stage and are used to start the encryption engine and pass control parameters such as capability bounds information. Each of the new instructions take multiple cycles to process and so the pipeline is stalled while these take place.

The *Key Generation and Management* unit (see Section 4.4) generates and stores the keys, as well as stores the next available IV invocation field for the *AES core* (see Section 4.3). During the `CSealEncrypt` instruction, if encryption is required, a key is requested based upon the sealing capability's otype and this, together with an IV and capability information relating to the area of memory to encrypt is passed to the *CSealEncrypt read/write* unit (see Section 4.5). This unit controls access to the AXI databus and hence memory via a control selector, *DbusCntrlSelector* component, and access to the *AES core* via the *AESCntrlSelector* component. Data is read and encrypted in 128 bit blocks, multiple blocks form a batch of data, and each batch is associated with an AT and IV. These are written back out to *memory* (see Section 4.2).

During the `CInvokeEncrypt` instruction, a previously stored key associated with the otype is requested from the *Key Generation and Management* unit along with the next available IV invocation value. These are passed along with the bounds of the code section

of the enclave to an *Instruction Cache*, and the bounds of the data section of the enclave to a *Data Cache* (see Section 4.6). The `CInvokeEncrypt` instruction then completes.

The caches inspect commands on the instruction and data bus, and take control of the data path from the AXI bus through to the encryption core via the *DbusCntrlSelector, IbusCntrlSelector* and the *AESCntrlSelector* components. Each cache fetches a cacheline of data from memory when required, and only if it falls within the bounds of the enclave, decrypts it and temporarily stores it in the cache memory ready for use. Each cacheline holds one batch of data, and each batch is associated with a single AT and IV. Write operations may also occur to the *Data Cache* from the processor and therefore cachelines can also be encrypted and written back into memory when necessary. These operations continue until the *Fetch* stage PCC goes out of bounds, signaling that the processor has left the enclave. Before disabling the encryption engine and releasing the instruction and data bus, the caches are flushed and registers are cleared.

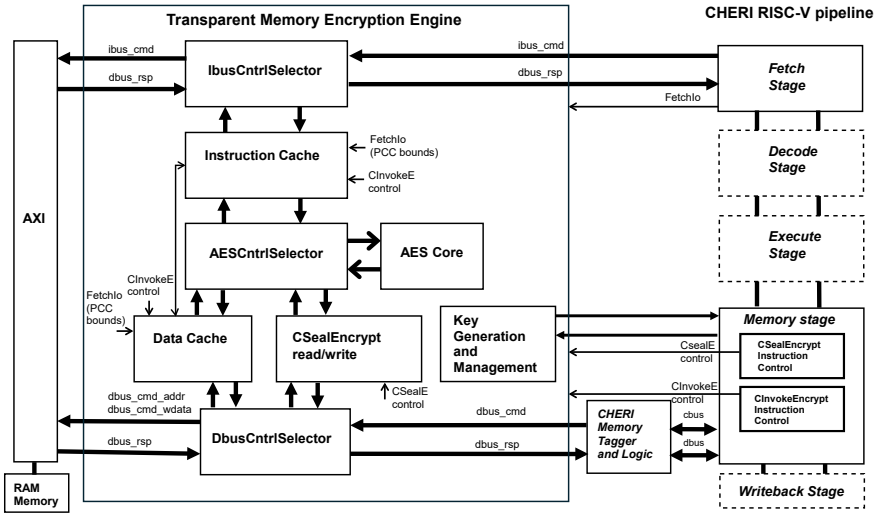


Figure 5: CHERI-Crypt Outline design.

4.2 Encrypted Enclave Memory

We store the AT/IV alongside the encrypted data in memory as shown in Figure 6.

Encrypted data During encryption the data is split into fixed sized blocks of memory called *batches*. The length of a batch (L_b) is assumed to be a multiple of 128-bit *blocks* (16 bytes) processed by the 128-bit *AES core*. L_b is a fixed hardware generic which can be chosen to optimize speed and resources for typical enclave sizes.

Authentication Tag For each batch of data encrypted, a 128-bit authentication tag is generated by the AES core which is stored alongside the data in memory. During decryption this tag is used to validate the decrypted data.

Initialization Vector We use 96 bits for the IV, which is used by the 128-bit AES-GCM algorithm because it is the most efficient [Dwo07]. For each batch of data encrypted with a fixed key, the IV (nonce) is different and since the IV is authenticated it is also stored visibly in memory alongside the data. We use the deterministic method [Dwo07] for constructing the IV value which is comprised of an upper 32-bit fixed field, and a lower 64-bit invocation field determined by a counter.

Storage size of the authentication tag and initialisation vector For simplicity we assume the storage size of the AT and IV should be a multiple of 16 bytes. This gives two

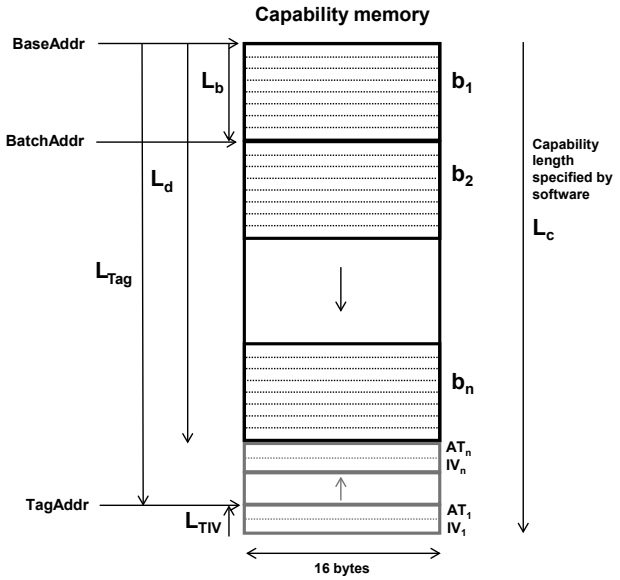


Figure 6: Encrypted Capability Memory.

storage options, either truncate the AT to 64 bits and concatenate this with the 64-bit invocation field of IV (since the upper 32 bits are fixed for the hardware instance), or store the full 128-bit AT and pad the IV to 128 bits. The first method uses less storage but reduces the authentication accuracy. We choose to use the second method to fully test the design.

Capability length to be encrypted We assume the capability length (L_C) specified by the software to be encrypted is passed as a whole number of batches in size plus the memory size required to store an AT and IV for each batch. It is also assumed that the capability is aligned in memory to the batch size which also corresponds to the size of a single decryption cache line.

Encrypting memory storage order As shown in Figure 6, data is stored one batch at a time. Batch 1 (b_1) is encrypted and stored starting at the lower bound (base address). AT_1 and IV_1 are stored at the upper bound. The new resized length of the data (L_{d1}) is calculated. This process is repeated for the second batch b_2, AT_2, IV_2, L_{d2}, etc. The end is determined when the current tag address (*TagAddr*) is no longer greater than the current data address (*DataAddr*) plus the batch length (L_b). If these two are not equal, then an encryption length error will occur, leading to a hardware exception. For each batch n the iterative calculation is as follows:

$DataAddr_n = DataAddr_{n-1} + L_b$, $TagAddr_n = TagAddr_{n-1} - L_{TIV}$ and $L_{dn} = L_{n-1} + L_b$ where $DataAddr_1 = BaseAddr$, $TagAddr_1 = BaseAddr + L_C - L_{TIV}$ and $L_{d1} = L_b$.

Decrypting from memory Because the batches are aligned with the cache lines, a batch address (*BatchAddr*) is first provided for decryption. To work out the corresponding AT and IV addresses we firstly calculate the number of batches in the capability from the resized capability length (L_d), and the batch number to be decrypted. From this we can find the length of the AT (L_{Tag}) from the base address, and then its corresponding absolute address. Since the batch size is specified as a power of 2 we can apply bit shift operations to find the AT length (L_{Tag}) directly. We assume that the bits to shift are fixed values, and that $L_b \geq L_{TIV}$, and are calculated as follows:

$$S_b = log_2(L_b), S_t = log_2(L_{TIV})$$

Then the tag address can be calculated from the Tag length as follows (where >> is a right shift and << is a left shift operation):

Number of batches: $BN = L_d/L_b = L_d >> S_b$

Batch number to decrypt: $b_n = (BatchAddr - BaseAddr)/L_b + 1 = ((BatchAddr - BaseAddr) >> S_b) + 1$

Tag length from base address: $L_{Tag} = L_d + (BN - b_n) \times L_{TIV} = L_d + (BN - b_n) >> S_t = L_d + (L_d >> (S_b - S_t) - ((BatchAddr - BaseAddr) >> (S_b - S_t) + L_{TIV}))$

Tag address: $TagAddr = BaseAddr + L_{Tag}$

4.3 AES core: Encryption and Decryption

We conservatively chose AES-GCM as authenticated encryption algorithm, similar to classic SGX. We note that other algorithms such as ASCON [TMC+24] or other modes such as XEX-based Tweaked-codebook mode with Ciphertext Stealing (XTS) could be used alternatively and might lead to lower runtime/size overheads. However, for the present paper, our focus was on demonstrating a proof-of-concept of the overall design. We acknowledge that further optimisations and different algorithm choices are possible as discussed further in Section 6.

The *AES core* can be configured for encryption and decryption in multiple ways to either minimise resources or maximise parallel processing for speed. We configure the *AES core* with separate encryption and decryption functions to allow parallel processing during simultaneous requirement. The *encryption function* is shared between the *CSealEncrypt Read/write* component and the *Data Cache* via the *AESCntrlSelector*, since neither of these require simultaneous use. Additionally, each cache has access to a dedicated *decryption function* to allow simultaneous decryption operations of cachelines.

We design the encryption and decryption functionality according to the AES-GCM NIST specification [MV07, Dwo07] (see Section 2). As the pipeline must be stalled whilst waiting for the completion of the encryption and decryption functions, minimizing the encryption latency is crucial. To address this, after the hash calculation, we process the Galois field multiplication (GF MULT) in parallel with the encryption (AES ENC) and thereby reducing the overall timing overhead, as illustrated in Figure 7.

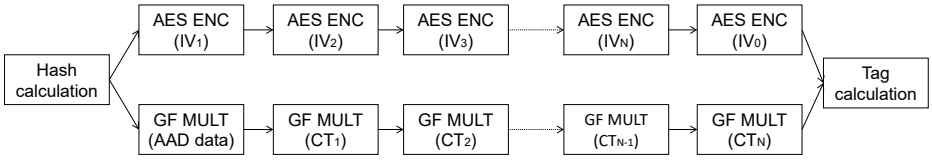


Figure 7: Parallel processing of authenticated encryption.

4.4 Key Generation and Management

Encrypting capability memories with distinct keys is essential for maintaining data integrity and isolation in CHERI. Using different keys prevents accidental or adversarial overlapping capabilities from gaining access to sensitive data, either during current use, or from past use of the same memory space if it has not been cleared effectively. For multi-enclave systems, such as CHERI-TrEE, encrypting each enclave with a unique key removes the need for an exhaustive and time-consuming capability overlap check, which requires a full memory sweep every time an enclave is initialised. The generation and storage of such keys need to be carefully managed, and away from any addressable memory. For this, we chose to implement a managed table stored internally to the CHERI-Crypt core.

The *Key Generation and Management* function is driven mainly by two components: the *Table* block and the *Key Generator* block as shown in Figure 8. The *Table* block contains a controller to store some critical values in a table, including the otype, key, used entry, the next available 64-bit invocation field of the IV (*NextIVCount*), and a key usage counter (*usedCounter*). The key generator is dedicated to producing the encryption keys, and is implemented as a Random Number Generator (RNG) compliant with NIST SP 800-90A [BK15]. These two components work together to ensure secure key generation and management throughout the encryption and decryption process. The mechanism of key generation and management is explained in the following paragraphs.

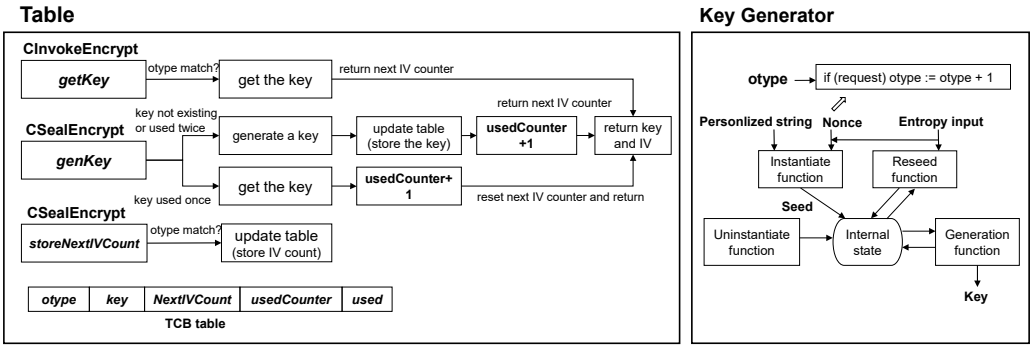


Figure 8: Mechanism of key generation and management.

Key Management Table The main principle of key management is that a single key can be used only once to encrypt or decrypt a pair of capabilities (code capability and data capability). Since the code and data capabilities are encrypted and sealed by two separate instructions this requires the hardware to generate a new key after the second use with the same otype (Section 3.2.2). When a *genKey* command is issued by the `CSealEncrypt` instruction or a *getKey* command is issued by the `CInvokeEncrypt` instruction, the request is transferred to the table unit to obtain a key. For the *getKey* command, the key is directly extracted from the table and returned together with the *NextIVCount*. For the *genKey* command, the first step is to verify whether a key already exists in the table for the specified otype and if it has already been used. If the key exists and has only been used once before, it is returned along with the *NextIVCount*, and the *usedCounter* is incremented by 1. If the key has been used twice or does not exist, a handshake with the *Key Generator* block is initiated to create a new key. Once generated, the new key is stored in the table and returnedth a reset *NextIVCount* value, and the *usedCounter* is incremented by 1. During the `CSealEncrypt` instruction, data is encrypted in batches, requiring a new IV value each time. Once all batches of data have been encrypted, the *storeNextIVCount* command is issued by the the `CSealEncrypt` instruction to store the next available IV count value back into the table as *NextIVCount* for subsequent use by the *Data cache*. Before exiting the enclave, the *Data cache* performs a final writeback operation, encrypting data with the updated IV count.

Following a `CInvokeEncrypt` instruction, batches of data are decrypted by the *AES Core* and controlled by both the *Instruction Cache* and the *Data Cache*. During this process the authentication tags are checked. If any tag checks fail, an error flag is propagated to the *Table* block which flushes all stored keys, and a hardware exception occurs. The system thereby ensures that compromised data cannot continue to be processed.

Key Generator Upon receiving a new key request from the *Table* block along with the otype value, the *Key Generator* starts generating the key. Using the Deterministic Random

Bit Generators (DRBG) mechanism based on block ciphers from NIST SP 800-90A [BK15], a personlised string, nonce, and an entropy input are fed into the initiate function to generate a seed. In our implementation, the nonce is substituted with the otype value supplemented by a counter, while the entropy input consists of a predefined length of data used for proof-of-concept demonstration. This entropy input is flexible and can be replaced with any desired entropy sources in future applications, including high-quality entropy sources or combinations of independent entropy sources tailored to meet specific security strength requirements of various scenarios [Bar20]. Using the seed output from the instantiate function, the generation function outputs a pseudo-random key and passes the key back to the *Table* block. In the key generation mechanism, the otype value is extended with a counter that increments by 1 each time a new key is generated. As a result, even if a capability pair is requested to be encrypted with the same otype, another *genKey* command to the key management unit will result in a different key. This prevents the risk of manipulating a malicious capability to recover the key of a trusted capability using the same otype value.

4.5 CSealEncrypt Read and Write

The functionality of the *CSealEncrypt Read/Write* component is to control the reading and writing of data from memory, and the encryption of capability data during the `CSealEncrypt` instruction. It interfaces with the *DbusCntrlSelector* and the *AESCntrlSelector* to control access to the memory and *AES Core*. As shown in Figure 9 a state machine waits for an encryption command from the `CSealEncrypt` instruction to start encrypting a batch of data, along with a *Batch Address*, a *Tag Address*, *Encryption key*, and a *NextIVCount* value. Firstly the IV is formed from the concatenation of the *NextIVCount* and a fixed part, and is pushed into an input data buffer connected to the *AES Core* encryption function. It then sends AXI commands to read the capability data from memory. Since there is only one address bus it cannot read and write at the same time. Received data is converted from 32 bits to 128 bits and directed to the input buffer. On the last read, a *last read* flag is set and passed along with the data to indicate to the encryption function when to start calculating the AT. When the input buffer is nearly full or all the reads in the batch have completed, the state machine looks at the output buffer of the *AES Core* encryption function for writing the encrypted data back to memory. This output data is converted back from 128 bits to 32 bits. When all the data read and writes are finished it will write the AT at the Tag Address followed by the IV before indicating the batch has complete. This process is repeated until all batches of data for a given capability has been encrypted. The `CSealEncrypt` instruction increments *NextIVCount* for each batch, and issues a command to store the *NextIVCount* back to the *Key Generator and Management* unit on completion of all batches of data.

4.6 Instruction and Data Cache

We chose a cache as the method to temporarily store the decrypted instructions and data since the authenticated encryption algorithm works on batches of data where a whole batch must be decrypted at a time (each batch has an AT and IV). A cache is an efficient choice where a whole batch can be stored and worked on within the confines of internal memory, whilst also being scalable with batch size. The type of cache implemented is a classic direct mapped with write-back functionality where each cache line holds a single batch of data. Both an *Instruction Cache* and a *Data Cache* is required. The *Cache* component consists of the *Cache Memory* and *Logic*, together with a *Cache Controller* as shown in Figure 10. A state machine in the *Cache Controller* resides in the *waitInvoke* state until a `CInvokeEncrypt` instruction command is detected. During this command the encryption key and enclave bounds are passed, along with the *nextIVCount*. Next, the

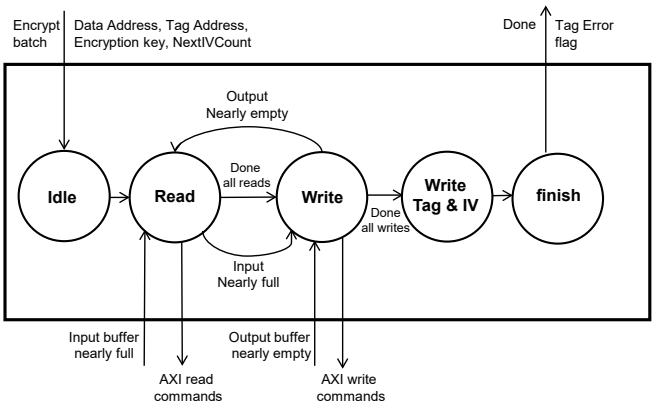


Figure 9: CSealEncrypt Read/Write controller

command part of the instruction or data bus is routed to the *cache* component via the *busCntrlSelector*. Now in the *waitRspStart* state, the controller waits for the responses in progress to finish before taking full control of the bus and then moves to the *startInvoke* state where commands are directed to the *cache memory* component.

Whilst bound checks are carried out by the CHERI instructions, the cache controller also needs to check they are specifically within the enclave region for transparent encryption. A Bounds Checker component is used to check that the current PCC is within the bounds of the enclave and at the same time the address commands on the bus are also checked. For the instruction bus, out of bounds address commands can occur during pre-fetch of instructions from memory which can happen when the PCC is still within the bounds of the enclave but the requested commands are not. When this occurs the commands are re-directed to the main memory. The *cache controller* will then continue and repeat this process until the current PCC also goes outside the enclave bounds. This allows jumping or branching towards the end of the enclave code to be unaffected. For the data bus this scenario can happen if the enclave code reads or writes data outside of its own data section.

When the address commands and PCC are within the bounds of the enclave the decision on what to do next is based on priority. For the *Data cache*, if there is a miss and the dirty bit is set, then the first priority is to perform a write back operation of the cacheline, (*writebackcacheline* state) before performing a read of the new cacheline (*readcacheline*), where both operations are directed via the *AES Core*. For both the *Data Cache* and the *Instruction Cache*, if there is a miss and the dirty bit is not set then the second priority is to perform a read of a new cacheline only, via the *AES Core* for decryption. Once a read of the new cacheline is undertaken, the current read or write command to the cache is repeated (*repeatReadWrite* state) and then the next command is processed. If there are no misses the cache can read or write data on every clock cycle.

For the read cacheline operation a batch of data is read from memory, along with the AT and IV (see Section 4.2) and pushed into an input buffer to the decryption function within the *AES Core*. Unlike the *CSealEncrypt Read/Write* block (see Section 4.5), reading and writing takes place simultaneously and decrypted data is written into the cache memory as soon as it becomes available. A similar process occurs for the writeback operation, where data is read from the cache, encrypted using a new IV value and written to memory.

Exiting from the enclave occurs when the PCC is no longer within the bounds of the enclave. The *Data cache* performs any writebacks of cachelines if they are required and stores the *nextIVCount* back in the *Key Generation and Management* table. For both caches the *Cache Memory* is flushed to empty out any decrypted contents. Before finishing and releasing the bus, the *Instruction Cache* waits for the *Data Cache* to finish which may

be some time after if a final writeback is required. This ensures registers are not cleared and further instructions are not carried out before the data side is ready.

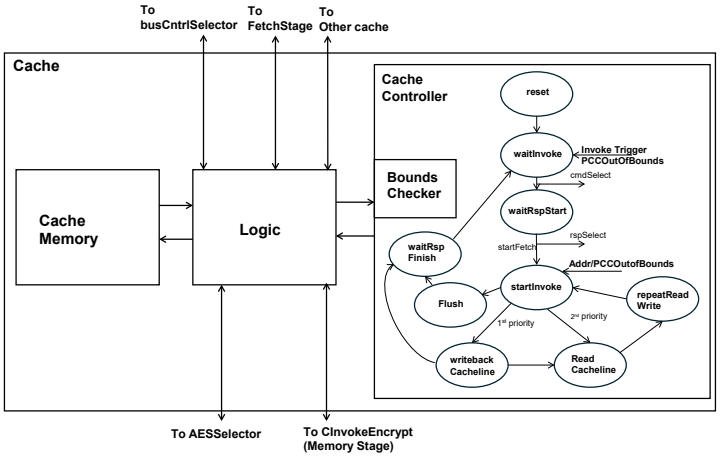


Figure 10: Encryption Cache

5 Microbenchmark

In this section we present microbenchmarks of our design. We firstly present the AES core, and then the full CHERI-Crypt design under the basic seal and invoke process.

5.1 AES Core: Encryption and Decryption Function Block

The AES core is tested against the NIST test vectors [MV07], with a 128-bit tag, 0-bit AAD, 128-bit key, and a 512-bit plaintext. Both encryption and decryption take 16 clock cycles to complete encryption or decryption of a 128-bit block of data. An additional 22 clock cycles are needed to calculate the AT. The total latency, in clock cycles to encrypt or decrypt a single batch of data can be calculated as follows:

$$Lat_{batch} = (N_{blks} + 1) \times clks_{data} + clks_{AT}$$

where Lat_{batch} is the latency of a single batch, N_{blks} is the number of blocks in a batch, 1 represents the additional IV processing, $clks_{data}$ is the number of clock cycles to process one 128 bit data block, and $clks_{AT}$ is the number of clock cycles to process the AT. For a batch size of 32 bytes, the latency is calculated as 70 clock cycles.

The encryption block (*AES_ENC*) and decryption block (*AES_DEC*) were implemented separately on a Zynq UltraScale+ XCZU9EG-2FFVB1156 FPGA. The area usage, maximum operating frequency, and dynamic power estimates are presented in Table 1. The *AES_ENC* and *AES_DEC* can achieve identical performance in terms of operating frequency, while maintaining slightly different area usage and dynamic power estimation.

Table 1: Implementation metrics for *AES_ENC* and *AES_DEC*.

	LUTs	**Flip-flops**	**Maximum freq. (MHz)**	**Dynamic power (mW)**
AES_ENC	4181	2457	100	148
AES_DEC	3701	2345	100	153

5.2 CHERI-Proteus with Full CHERI-Crypt Design

5.2.1 Software and Hardware Setup

We set up a small program to perform a basic seal and invoke process, where the code section and the data section of the enclave, are given in Listing 1 and Listing 2 in the Appendix, respectively. The enclave is aligned in memory, and the capabilities that point to the enclave are derived from labelled addresses to cover the bounds of the data, ATs and IVs. The CHERI-Proteus core is configured with on-chip 128 KiB RAM memory representative of FPGA-based block RAM with low latency expectations. The *Key Generation and Management* Unit is configured with a table size to accommodate a maximum of three enclaves. The `CSealEncrypt` instruction is called twice, once for the code section and once for the data section. The `CInvokeEncrypt` instruction is then called. We run the invoked enclave multiple times. The enclave code itself writes a data value to its own data section which is encrypted and reads and writes data to a location outside of the enclave area which is not encrypted. This is representative of fetching input parameters and writing a final result that is retrievable by the untrusted application. The enclave then returns.

5.2.2 Latency

We measured the latency of the `CSealEncrypt` instruction from simulation as given in Table 2. The latency of the three main parts of the instruction are also given: *AES_ENC*, key generation, and additional latency of reading and writing to memory. (Note that some reading and writing also takes place in parallel with the encryption processing). We used a batch size of 32 bytes which was initially chosen to cover the small enclave test code whilst allowing the data section to be represented by multiple batches (two in this first case).

The time taken (in clock cycles) to run the enclave test code was also measured after the `CInvokeEncrypt` instruction had been called. The results are given in Table 3 with and without encryption, and show both the instruction and data caches together with the latency of the operations performed for two different batch sizes. For the instruction side, a first instruction read requires a read cacheline operation to decrypt the instructions. Similarly when the enclave writes to its own data section this requires a read cacheline operation to decrypt and load the cache with the current memory contents before writing the value. On return of the enclave a writeback operation is performed before both caches are flushed. During processing of commands and other wait operations (Command Proc.) the data cache recognises the reads and writes outside the enclave and sends these out to main memory but waits for these to complete before continuing. Since both the instruction cache and data cache operate in parallel and wait for each other to complete final operations, they both have the same total latency. To encrypt the enclave code and data sections with a batch size of 32 bytes, and run the encrypted enclave required 626 additional clock cycles compared to the 17 clock cycles without encryption.

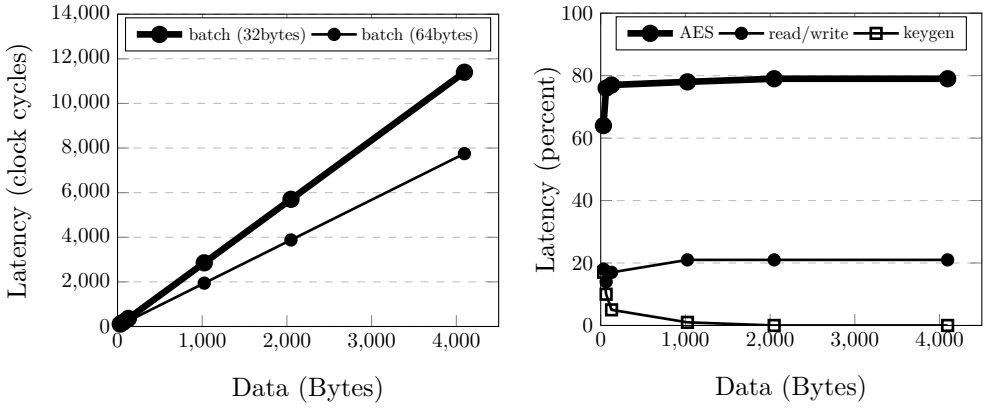
Table 2: CSealEncrypt latency for the enclave code given in Listing 2, with a batch size of 32 bytes

	Code/Data Size (bytes)	**AES Enc Lat. (clks)**	**Key Gen. Lat. (clks)**	**Read/Write Lat. (clks)**	**Total Latency (clks)**
CSealEncrypt (Code)					
Without encryption	32	-	-	-	1
With encryption	32	70	19	20	109
CSealEncrypt (Data)					
Without encryption	64	-	-	-	1
With encryption	64	140	19	25	184

Table 3: Enclave runtime latency for the code given in Listing 2, with batch sizes of 32 and 64 bytes

	Readline Lat. (clks)	Writeback Lat. (clks)	Flush Lat. (clks)	Wait Lat. (clks)	Command Proc. Lat. (clks)	Total Latency (clks)
Instruction cache						
Without encryption	-	-	-	-	-	15
With encryption (32 bytes)	90	0	32	92	136	350
With encryption (64 bytes)	122	0	64	123	167	476
Data Cache						
Without encryption	-	-	-	-	-	15
With encryption (32 bytes)	90	90	32	0	138	350
With encryption (64 bytes)	120	122	64	0	170	476

Figure 11(a) shows the measured latency of the `CSealEncrypt` instruction for the test enclave, whist varying the size of the data section and hence number of batches of data to be encrypted. Results are presented for two different batch sizes (32 and 64 bytes). For the 32 byte batch case Figure 11(b) also shows the main components as a percentage of the total latency. As the data size increases the latency of the `CSealEncrypt` instruction increases, as we would expect. For the same amount of data, latency is greater for a smaller batch size due to the added overheads of generating extra AT and IV values and stopping and starting the reading and writing process. As the amount of data increases, the latency is more dominated by the AES encryption function. As shown in Table 3 changing from a 32 to 64 byte batch size increases the running latency of the enclave code since more memory is decrypted, but the same amount of instructions and data is being run and processed. If memory allows, it makes sense to have the biggest batch size possible (with minimal padding), to reduce initial encryption latency, however when running the enclave code, consideration would also need to be given to what the enclave code is doing. Processing small amounts of data with large batch sizes could cause a large unnecessary writeback delay on return.



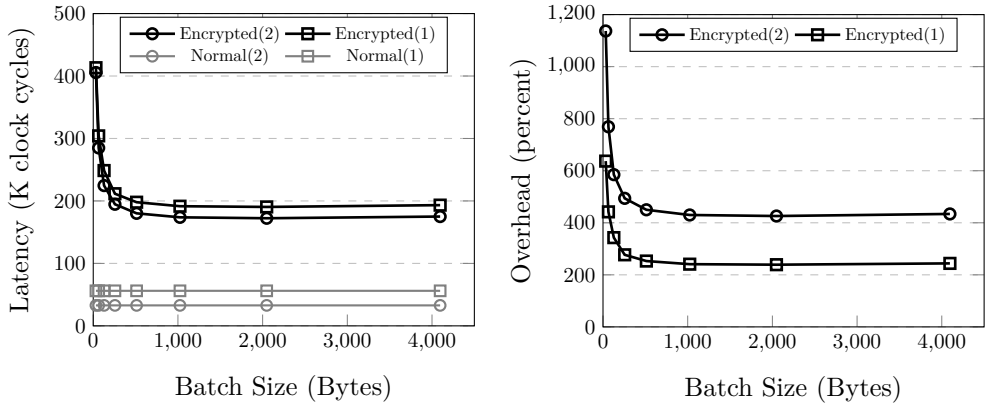
(a) Latency with two different batch sizes.

(b) Latency percentage of individual components

Figure 11: CSealEncrypt latency with increasing amounts of data.

5.2.3 Performance of Larger Applications

To assess performance of more realistic applications, enclaves with a larger code base were considered for two types of computation scenario (1) a program containing repeated instruction sequences to read/write data to a consecutive block of memory within the



(a) Enclave computation latency.

(b) Encryption overhead.

Figure 12: Encrypted enclave computation latency and encryption overhead with increasing batch size for two types of computation: (1)Data access, (2)Instructions only. (Where Normal = not encrypted)

enclave's data section, and (2) a program containing repeated instruction sequences without accessing memory. The programs were written to fill 128KiB of memory. The latter scenario represents a baseline program where only consecutive instructions are decrypted, whereas the former scenario additionally requires data to be decrypted and encrypted, which could be representative of a data processing application. Figure 12 (a) shows the enclave computation latency and Figure 12 (b) shows the encryption overhead (as a percentage of the computation without encryption) for the two program scenarios as the batch size is increased. For both programs, increasing the batch size to around 1KiB minimises the encrypted latency and overhead. However, whilst adding data accesses increases the over-all computation latency, the encrypted case increased by a smaller rate than the non-encrypted, (contributed to by simultaneous decryption of instructions and data), resulting in a smaller overhead for a more complex processing scenario.

5.2.4 FPGA Implementation

We implement our design on a Xilinx Zynq UltraScale+ XCZU9EG-2FFVB1156 FPGA board. Section 5.2.4 shows the utilisation, frequency and dynamic power of CHERI-Proteus, with and without the CHERI-Crypt design. The results include the processor core and AXI interconnect with peripherals and all ports connected. For implementation, the CHERI-Crypt encryption engine is configured with a batch size of 32 bytes, 4 cache lines, and a key generator table with a maximum of three enclaves. CHERI-Crypt utilises four times as many flip flops and more than three times as many LUTs compared to the normal CHERI-Proteus core. CHERI-Crypt utilisation is dominated by the three AES encryption and decryption functions, followed by the data cache. The addition of the CHERI-Crypt encryption engine has some impact on frequency performance but further optimisation of the design could see this impact reduced. Utilisation will vary depending upon the configuration parameters such as: (1) batch size, which will impact the size of the required cache, and (2) the number of enclaves catered for, which will impact the size of the table component within the key generation and management unit.

6 Conclusions

Possible Optimisations As our implementation serves as a basic proof-of-concept, we believe that further optimisations of the design can be achieved to reduce utilisation and

Table 4: Implementation on the Zynq UltraScale+ XCZU9EG-2FFVB1156 FPGA board with 128 KiB memory, 32 byte batch size with 4 cache lines, and a key table for 3 enclaves. Percentages indicate area usage relative to total FPGA size.

Processor (128 KiB memory)	Area occupation			Maximum freq. (MHz)	Dynamic power (mW)
	LUTs	Flip-flops	BRAMs		
CHERI-Proteus	7790 (3%)	3905 (1%)	32 (4%)	63	143
AXI and peripherals	232	232	32		
Pipeline	7558	3673	0		
CHERI-Crypt-Proteus	25 512 (9%)	15 831 (3%)	32 (4%)	57	257
AXI and peripherals	291	295	32		
Pipeline	8083	3671	0		
CHERI-Crypt:	17 138	11 865	0		
Instructions	451	290	0		
Control Selectors	211	0	0		
CSealEncrypt Read/Write	966	941	0		
Data Cache	2017	2260	0		
Instruction Cache	1416	1444	0		
Key Gen: Table	728	214	0		
Key Gen: Generator	1088	429	0		
*AES Core (1*ENC, 2*DEC)*	10 261	6287	0		

increase performance, with the aim to maintain the maximum frequency that can be achieved by the standard CHERI-Proteus. It is possible to reduce the utilisation of the AES Core for example by sharing the decryption function between caches, or amalgamating the encryption and decryption functions into a single design, but this would impact the latency of the read cacheline and write back operations whilst running enclave code, as well requiring additional multiplexing logic. It would also be possible to clock the encryption and decryption functions at a greater frequency than the rest of the design to reduce over-all latency, and these options could be further explored. While we opted for the (conservative) choice of AES-GCM, CHERI-Crypt could equally use other cryptographic constructions commonly employed for memory encryption. This includes modes like XTS, as well as use-case-optimised ciphers like PRINCE [BCG+12], QARMA [Ava16], ASCON [TMC+24], or Voodo [LUSM24]. The batch size is a configurable element of the design so that it can be tailored to the encryption requirement of sealed capabilities and enclave sizes used by the required application. For example to reduce utilization of the cache it would be favourable to have small batch sizes but this may come at a latency cost if there is a large overhead of requiring lots of read cacheline and writeback operations, as well as storing the extra AT and IV values. Similarly, large batch sizes would require more storage space in terms of the cache, but may reduce the number of read cacheline and writeback operations performed. In addition, large batch sizes may consume more of the main memory, if for example the enclave code or data section needed padding up to the size of a whole batch size. We use 32 bytes to store the AT and IV values in memory. This has less of an impact on large batch sizes, but more of an impact on small batch sizes. As discussed previously this storage requirement can be halved by truncating the AT and IV values and by only checking the truncated AT part on decryption.

Limitations There are some limitations in terms of design and results. First, as CHERI-Crypt currently does not provide freshness, e.g., through counters and a Merkle tree as in the SGX MEE [CD16]. Hence, an active attacker could theoretically replay memory contents, though we note that this (i) requires a sophisticated setup in practice (see e.g., [LJF+20]) and (ii) is limited in time until the renewal of encryption keys. We remark that adding freshness is somewhat orthogonal to encryption/authentication, and that CHERI-Crypt shares this limitation in principle with other commercial TEE designs like SEV-SNP [AMD20] and Intel SGX-Scalable and TDX [Int23b, Int23a], where the freshness guarantees were weakened to support large enclave size and maximum performance. We compare the susceptibility of CHERI-Crypt and other common TEEs w.r.t. different

Table 5: Attacks prevented (●), partially prevented (◐), and not prevented (○) by CHERI-Crypt and other commercial TEEs.

	CHERI-Crypt	SGX-Classic	SGX-Scalable/TDX	SEV-SNP	Trustzone
Direct memory read*	●	●	●	●	○
Direct memory write*	●	●	●	◐	○
Direct replay of memory*	○	●	○	○	○
Side channels and fault injection	○	○	○	○	○
Spatial mem. safety of enclaves	●	○	○	○	○
Microarchitectural and cache attacks on enclaves	○	○	○	○	○

* Assuming direct access to the memory bus, not taking MMU/CPU restrictions into account.

relevant attack vectors in Table 5.

The CHERI-Crypt design introduces additional hardware exception rules for checking permissions and other encryption aspects, but does not prevent all types of encryption errors. For example, while encrypting data during the `CSealEncrypt` instruction, the hardware design has some capability length checks in place leading to hardware exceptions, but will not guard against all length errors. If no space is allocated for AT and IV in the capability length passed from the software but it still passes the length checks, the returned result will be partial encryption of data. Another limitation is that the latency measurements of the instructions are based on simulation results with a low latency path to the main memory and does not include any latency that may occur if an alternative off-chip RAM (e.g., DRAM module) is used. While this represents a good picture of the instructions themselves, further testing is required with external memory usage.

Conclusion With our CHERI-Crypt transparent encryption engine design, we presented a solution to safeguarding sealed capabilities stored within external memory. We developed the design based on the assumption that it could help protect CHERI-based TEE from hardware-based attacks, whilst taking advantage of the inherent memory safety provided by the CHERI architecture. CHERI-Crypt also eliminates the need for a complete memory sweep operation or the use of linear capabilities necessary for current CHERI-TrEE designs [VSNJ+23]. An advantage of the design is the configurable nature of the batch size allowing it to be optimised for specific applications. Additionally, we have options to optimise our design further in the future.

Acknowledgements

This research was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1, EP/V000454/1, and EP/S030867/1. Our project is funded by the DSbD (Digital Security by Design) Programme delivered by UKRI to support the DSbD ecosystem.

References

[ACC+23a] Saar Amar, Tony Chen, David Chisnall, Felix Domke, Nathaniel Wesley Filardo, Kunyan Liu, Robert M. Norton, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. CHERIoT: Rethinking security for low-cost embedded systems. Technical report, Microsoft, 2023.

[ACC+23b] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. Cheriot: Complete memory safety for embedded devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, page 641–653. Association for Computing Machinery, October 2023.

[AMD20] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. Technical report, AMD, January 2020.

[ARM19] ARM. ARM Morello Program, 2019. https://www.morello-project.org/.

[Ava16] Roberto Avanzi. The QARMA block cipher family – almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. Cryptology ePrint Archive, Paper 2016/444, 2016.

[Bar20] Elaine Barker. Recommendation for Key Management: Part 1 – General. Technical report, NIST, 2020.

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knežević, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçın. PRINCE - a low-latency block cipher for pervasive computing applications (full version). Cryptology ePrint Archive, Paper 2012/529, 2012.

[BK15] Elaine Barker and John Kelsey. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical report, National Institute of Standards and Technology (NIST), June 2015.

[Blu20] Bluespec. BESSPIN Government Furnished Equipment (GFE)., 2020. https://github.com/GaloisInc/BESSPIN-GFE.

[BNP23] Marton Bognar, Job Noorman, and Frank Piessens. Proteus: An Extensible RISC-V Core for Hardware Extensions. In *RISC-V Summit Europe '23*, June 2023.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Paper 2016/086, 2016.

[Cod24] Codasip. The only commercial implementation of CHERI technology, 2024. https://codasip.com/solutions/riscv-processor-safety-security/commercial-cheri/.

[CVM+21] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. VoltPillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 699–716. USENIX Association, August 2021.

[Dwo07] Morris Dworkin. NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation:Galois/Counter Mode (GCM) and GMAC. Technical report, National Institute of Standards and Technology (NIST), November 2007.

[EKY11] Barış Ege, Elif Bilge Kavun, and Tolga Yalçın. Memory encryption for smart cards. In *Smart Card Research and Advanced Applications*, pages 199–216. Springer Berlin Heidelberg, 2011.

[GJC21] Naina Gupta, Arpan Jati, and Anupam Chattopadhyay. Memenc: A lightweight, low-power, and transparent memory encryption engine for iot. *IEEE Internet of Things Journal*, 8(9):7182–7191, 2021.

[HB17] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, page 129–142. Association for Computing Machinery, 2017.

[HT14] Michael Henson and Stephen Taylor. Memory encryption: A survey of existing techniques. *ACM Comput. Surv.*, 46(4), March 2014.

[Int15] Intel. Software Guard Extensions (Intel® SGX) (Developer Guide). Technical report, Intel, 2015.

[Int23a] Intel. Architecture specification: Intel trust domain extensions (Intel TDX) module. Specification 344425-005US, Intel, February 2023.

[Int23b] Intel. Intel Xeon scalable processors: NEX eagle stream platform, Intel platform security. Technical Report 784473, Intel, August 2023.

[JEA20] Nicolas Joly, Saif ElSherei, and Saar Amar. Security Analysis of CHERI ISA. Technical report, Microsoft Security Response Center (MSRC), October 2020.

[Kap17] David Kaplan. Protecting VM Registers State with SEV-ES. Technical report, AMD, February 2017.

[KPW21] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. Technical report, AMD, October 2021.

[Lip19] Aaron Lippeveldts. Linear capabilities for cheri: an exploration of the design space. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2019, page 47–48, New York, NY, USA, 2019. Association for Computing Machinery.

[LJF+20] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia che Tsai, and Raluca Ada Popa. An Off-Chip attack on hardware enclaves via the memory bus. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 487–504. USENIX Association, August 2020.

[LUSM24] Lukas Lamster, Martin Unterguggenberger, David Schrammel, and Stefan Mangard. Voodoo: Memory tagging, authenticated encryption, and error correction through MAGIC. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7159–7176, Philadelphia, PA, August 2024. USENIX Association.

[MV07] David A. McGrew and John Viega. The Galois/Counter Mode of Operation (GCM). Technical report, NIST, 2007.

[Pro23] Proteus developers. Proteus: a configurable RISC-V core., 2023. https://github.com/proteus-core/proteus.

[SSL+23] David Schrammel, Salmin Sultana, Michael LeMay, David Durham, Martin Unterguggenberger, Pascal Nasahl, and Stefan Mangard. Memes: Memory encryption-based memory safety on commodity hardware. In *Proceedings of the 20th International Conference on Security and Cryptography - SECRYPT*, pages 25–36, 2023.

[TMC+24] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, Jinkeon Kang, and John Kelsey. Ascon-Based Lightweight Cryptography Standards for Constrained Devices. Technical report, NIST, 2024.

[VSNJ+23] Thomas Van Strydonck, Job Noorman, Jennifer Jackson, Leonardo Alves Dias, Robin Vanderstraeten, David Oswald, Frank Piessens, and Dominique Devriese. Cheri-tree: Flexible enclaves on capability machines. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 1143–1159, 2023.

[WJX+19] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Transactions on Computers*, 68(10):1455–1469, 2019.

[WNW+23] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). Technical Report UCAM-CL-TR-987, University of Cambridge, Computer Laboratory, September 2023.

[WUS+17] Mario Werner, Thomas Unterluggauer, Robert Schilling, David Schaffenrath, and Stefan Mangard. Transparent memory encryption and authentication. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, 2017.

[WWME20] Luca Wilke, Jan Wichelmann, Mathias Morbitzer†, and Thomas Eisenbarth. Sevurity: No security without integrity breaking integrity-free memory encryption with minimal assumptions. In *IEEE Symposium on Security and Privacy (SP)*, pages 1483–1496. IEEE Computer Society Technical Committee on Security and Privacy, August 2020.

[WWN+15] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[YEP+06] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving cost, performance, and security of memory encryption and authentication. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, page 179–190. IEEE Computer Society, 2006.

A Test code for Seal and Invoke with Encryption

```
TEST_CASE_START(1)
  // 1. Set up code capability to be sealed in c1
    la t0, capcode_mem_start
    CSetAddr c1, ROOT, t0
    la t1, capcode_mem_end
    sub t0, t1, t0
    CSetBoundsExact c1, c1, t0
  // 2. Set up data capability to be sealed in c2
    la t0, capdata_mem_start
    CSetAddr c2, ROOT, t0
    la t1, capdata_mem_end
    sub t0, t1, t0
    CSetBoundsExact c2, c2, t0
    li t0, ~(1 << PERM_PERMIT_EXECUTE)
    CAndPerm c2, c2, t0
  // 3. Set up third data capability outside enclave in c5
    la t0, outside_mem_start
    CSetAddr c5, ROOT, t0
    la t1, outside_mem_end
    sub t0, t1, t0
    CSetBoundsExact c5, c5, t0
  // 4. Set up o-type seal in c29
    li t1, 0x04
    CSetOffset c29, ROOT, t1
  // 5. Seal and encrypt code and data , c3 and c4
    CSealEncrypt c3, c1, c29
    CSealEncrypt c4, c2, c29
  // 6. Set up return address in c6
    la t0, enclaveret
    CSetOffset c6, ROOT, t0
  // 7. Do invoke 10 times
    li t5, 0xa
    dosub:
      CInvokeEncrypt c3, c4
    enclaveret:
     addi t5, t5, -1
     beqz t5, 1f
     j dosub
    1:
     TEST_PASSFAIL
```

Listing 1: CHERI-RISC-V Seal and invoke with encryption test code.

B Test Code for Enclave

```
// ---code section---
 capcode_mem_start:
  // 1. set the return address for when the enclave completes
  Cmove cra, c6
  // 2. write first word to the enclave data section c31
  li t1, 0x44
  sw.cap t1, (c31)
  // 3. read/write to data outside the enclave
  lw.cap t2, (c5)
  addi t2, t2, +1
  sw.cap t2, (c5)
  // 4. clear tag and return from enclave
  CClearTag c31, c31
  CJALR cra
 capcode_code_end:
  //PADDING for 32 byte batch - none needed
 capcode_pad_end:
  //AT and IV - 32 bytes for 1 batch
```

```
19   .fill 8, 4, 0x00
20  capcode_mem_end:
21 // ---data section---
22  capdata_mem_start:
23   //Two batches of data - 64 bytes
24   .fill 16, 4, 0x22
25  capdata_data_end:
26   //PADDING for 32 byte batch - none needed
27  capdata_pad_end:
28   //AT and IV - 64 bytes for 2 batches
29   .fill 16, 4, 0x00
30  capdata_mem_end:
31 // ---data outside enclave---
32  outside_mem_start:
33   .fill 16, 4, 0x66
34  outside_mem_end:
```

Listing 2: Enclave test code.