More Granular, Less Trust: Enforcing Intra-Process Isolation with Arm CCA in an Untrusted Management Environment

Shiqi Liu, Zhouqi Jiang, Jie Wang, Wei Zhou, Kun Sun, Zhaohui Chen, and Yulai Xie

***Abstract*—With the increasing adoption of confidential computing, security-sensitive applications are often deployed in confidential virtual machines (CVMs), which reduce reliance on third-party cloud providers. However, privilege attacks originating from the OS remain a significant threat in these environments. Existing finer-grained isolation schemes, such as SHELTER, provide process-level protection but are still vulnerable to intra-process attacks and potential collusion between the OS and intra-process adversaries. Many current intra-process isolation techniques continue to depend on the OS to manage and enforce isolation domains, leading to a large Trusted Computing Base (TCB). This gap highlights the need for more granular, less trust-dependent confidential computing solutions.**

In this paper, we present CCAegis, a system that extends the Arm Confidential Compute Architecture (CCA) to enforce intra-process isolation of sensitive data and operations, safeguarding them from both intra-process adversaries and the OS. We employ static analysis to track the flow of sensitive data and identify functions that handle such data. Permission-switching instructions are inserted at the function call and return points, adjusting permissions via the Granule Protection Table (GPT) to ensure that only designated functions can access the isolated data. Notably, CCAegis places trust solely in the Secure Monitor, which configures the GPTs and manages domain switching, thereby minimizing the TCB. We implemented CCAegis on both an official emulator and a real development board to assess its performance. Our experimental results show that CCAegis effectively isolates sensitive data and operations, with performance overheads ranging from 1.01× to 1.43× compared to the original version across real-world cryptographic workloads.

***Index Terms*—Trusted Execution Environment, Confidential Compute Architecture, Intra-Process Isolation, Granule Protection Table**

I. INTRODUCTION

PROTECTING sensitive data and operations from memory leakage attacks is crucial, particularly when adversaries gain control over privileged software (e.g., OS and hypervisor), which can directly access user-space address space. Recently, confidential computing has gained significant traction, with major chip manufacturers (e.g., AMD, Intel, and Arm) adopting a virtual machine-based isolation model [1], [2], [3]. This model allows users to deploy security-sensitive applications in confidential virtual machines (CVMs) without needing to trust the hypervisor of the cloud service provider. While this approach simplifies the integration of OS services for security-sensitive applications, it remains vulnerable to privilege escalation attacks [4] against the OS—especially when malicious applications share the same CVM.

Arm has enhanced its TrustZone architecture [5] with the Confidential Compute Architecture (CCA) [3] in Armv9-A, introducing two new execution environments: the realm world and the root world, enabled by the Realm Management Extension (RME) [6] hardware primitive. This advancement allows third-party developers to seamlessly deploy CVMs within the realm world. However, the usability comes at the cost of a large Trusted Computing Base (TCB), which encompasses the guest OS and other privileged software stacks within the realm world and the root world.

A fundamental limitation of current approaches is their overly coarse-grained isolation, given that only a small fraction of an application's code is security-sensitive. For example, RContainer [7] excludes the native OS from its TCB but only extends Arm CCA with container-level isolation. SHELTER [8] implements isolation at a finer granularity (process-level); however, its monolithic isolation model proves insufficient for defending against intra-process attacks (e.g., buffer overflows [9]). Another critical issue is that current intra-process isolation solutions either introduce an excessively large TCB or face significant compatibility challenges. Many solutions predominantly rely on OS-managed mechanisms, making them vulnerable to privilege escalation attacks [4]. For instance, libmpk [10] and ERIM [11] leverage Memory Protection Keys (MPK) for user-space domain switching, but domain configuration still involves OS intervention. On Arm, Shreds [12] and ARMLock [13] build compartments atop legacy memory domains; they likewise depend on the OS for enforcement, and memory domains are not supported in AArch64 and are deprecated on modern Arm platforms, limiting their applicability. Other approaches [14], [15] use Intel SGX [16] to protect against privilege escalation and implement fine-grained isolation by creating multiple domains within a single enclave. However, these approaches require hardware modifications, which lead to compatibility issues

This work was supported in part by the National Key Research and Development Program of China (No.2022YFB4501300), in part by the National Natural Science Foundation of China (No.62202194 and No.62202188). The associate editor coordinating the review of this article and approving it for publication was Prof. Fengwei Zhang. *(Shiqi Liu and Zhouqi Jiang contributed equally to this work. Corresponding author: Jie Wang and Wei Zhou.)*

Shiqi Liu, Zhouqi Jiang, Jie Wang, and Wei Zhou are with Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China, and also with the JinYinHu Laboratory, Wuhan, 430040, China (e-mail: shiqiliu@hust.edu.cn; luojia@hust.edu.cn; wangjie_s@hust.edu.cn; weizhou_sec@hust.edu.cn).

Kun Sun is with the Center for Secure Information Systems, George Mason University, Fairfax, VA, 22030, USA (e-mail: ksun3@gmu.edu).

Zhaohui Chen is with School of Integrated Circuits, Peking University, Beijing, 100871, China, and also with the DAMO Academy, Alibaba Group, Beijing, 100020, China (e-mail: czh@pku.edu.cn).

Yulai Xie is with Key Laboratory of Information Storage Systems, Ministry of Education, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China (e-mail: ylxie@hust.edu.cn).

with Arm architectures.

The key to addressing these issues lies in exploring a new mechanism that ***leverages off-the-shelf hardware features on Arm platforms to achieve intra-process isolation within an untrusted management environment***. Beyond the usual virtual address partitioning via page tables, Arm CCA provides the Granule Protection Table (GPT) to split a process's physical address space into protection domains. Crucially, GPT configuration and switching are performed by the Secure Monitor in the Root world (trusted firmware), which removes the OS from the TCB and keeps it minimal. Because GPT enforces permissions directly over physical memory under the Monitor's control, *per-module* vetting of dynamically loaded kernel code (as required by same-privilege schemes such as Nested Kernel [17]) is not needed to prevent isolation-breaking instructions. Building on these observations, we propose CCAegis, a system that enforces fine-grained, low-trust isolation using the GPT. CCAegis allocates sensitive intra-process data to a dedicated physical memory region and isolates this data by creating separate GPTs for sensitive and non-sensitive code. Specifically, only the GPT associated with sensitive code is configured to grant access to the protected memory region. CCAegis achieves domain switching by switching GPTs at the boundaries of sensitive code.

However, directly employing GPT for intra-process isolation faces two challenges. First, existing isolation models—whether native VM-level isolation or subsequent improvements enabling container/process-granular isolation [7], [8]—are monolithic by design, offering deployment friendliness that requires no restructuring of application logic. In contrast, ***extending CCA to achieve intra-process isolation imposes significant porting costs***, as it requires developers to manually identify sensitive code boundaries and make invasive modifications to application code (e.g., inserting GPT-switching instructions). Second, since the process management environment is provided by the untrusted OS, ***the isolation enforced by the Monitor introduces a semantic gap between the user space and the root world***. Specifically, the process operates within a virtual address space, while the Monitor interacts directly with physical memory addresses, lacking memory mapping information for the pages containing sensitive data.

To maintain the deployment-friendly nature of CCA, CCAegis modifies the LLVM compiler to perform static analysis (including points-to and taint analysis) on security-sensitive application source code, tracking the propagation of sensitive data (e.g., cryptographic keys) and identifying the minimal subset of functions that access such data (§IV). CCAegis then inserts GPT-switching instructions at the identified code boundaries to enforce memory isolation. Additionally, given the untrusted OS, CCAegis automatically replaces system calls related to the memory allocation and usage of sensitive data.

To bridge the semantic gap with user space, CCAegis introduces a kernel driver that initializes and maps virtual memory for sensitive data, as well as creates shadow page tables (parallel to native page tables for direct OS updates) for protected processes (§V). The root world ensures isolation against the OS by marking the physical pages containing sensitive data and the shadow page tables as inaccessible in the OS's GPT, preventing unauthorized access to sensitive data or tampering with memory mappings. Notably, this driver is not considered trusted to maintain a minimal TCB. Therefore, CCAegis's root world component must verify the correctness of memory mappings by inspecting the shadow page tables.

We implement CCAegis on an official Arm emulator and a real Arm hardware SoC. The code size of CCAegis is around 5K lines of code (LoCs), comprising 2K LoCs for static analysis modules in C++/Rust and 3K LoCs for runtime isolation modules in C. We evaluate the performance of CCAegis across five widely used real-world applications (e.g., `Nginx`). The evaluation shows that CCAegis achieves strong isolation by protecting only the sensitive parts of a program containing 10.57% to 35.26% of the total LoCs, with a moderate performance overhead ranging from 1.01× to 1.43× compared to the original version. We release the source code for CCAegis at https://github.com/erhade/CCAegis.

In summary, we make the following contributions:

- We present CCAegis, an intra-process isolation system built on Arm CCA using the GPT primitive, designed to prevent sensitive data leakage in an untrusted management environment.
- We address two key challenges: automating program partitioning and isolation instrumentation through static analysis, ensuring CCAegis remains deployment-friendly, and bridging the semantic gap via an untrusted kernel driver and a trusted Monitor, all while maintaining a minimal TCB.
- We evaluate CCAegis on both an official emulator and a real development board. The results show that CCAegis effectively protects sensitive data with moderate performance overhead.

II. Background

A. Arm Confidential Computing Architecture

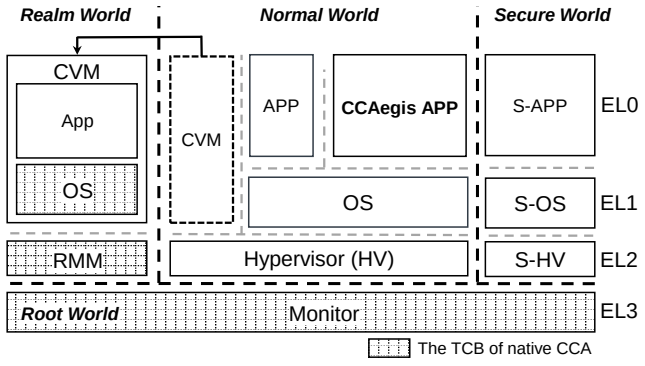

Fig. 1. The architecture of CCA.

TABLE I
PHYSICAL ADDRESS ACCESS PERMISSIONS OF ARM CCA.

Security State	Normal World	Realm World	Secure World	Root World
Normal	✓	✗	✗	✗
Realm	✓	✓	✗	✗
Secure	✓	✗	✓	✗
Root	✓	✓	✓	✓

TABLE II
COMPARISON OF CCAEGIS WITH RELATED TEE SYSTEMS. † CURE SUPPORTS RESILIENT TEE; HERE, WE CONSIDER ITS FINEST GRANULARITY.

System Name	Isolation Primitive	Isolation Granularity	Trusted Computing Base (LoCs)	Intra-Process Isolation	Deployment Friendliness	Arm Support	Unrestricted Data Volume
TDX [2]	TDX	VM	TDX Module (46K) + OS in Trust Domain (≥25M)	✗	✓	✗	✓
SGX [16]	SGX	Part of App	-	✓	✗	✗	✓
Glamdring [18]	SGX	Function	-	✓	✓	✗	✓
SeCage [19]	VMFUNC	Function	KVM Hypervisor (10M)	✓	✓	✗	✓
SEV (-ES, -SNP) [1]	SEV	VM	Firmware (N/A) + OS in Confidential VM (≥25M)	✗	✓	✗	✓
Keystone [20]	PMP	Part of App	Monitor (37K) + Runtime (9K)	✓	✗	✗	✓
CURE [21]	Customized HW	Part of App†	Monitor (3K) + Runtime (16M, Opt.)	✓	✗	✗	✓
Sanctum [22]	Customized HW	Part of App	Monitor (2K) + glibc (0.9M)	✓	✗	✗	✓
PENGLAI [23]	Customized HW	Part of App	Monitor (43K) + musl libc (80K)	✓	✗	✗	✓
TrustZone [5]	TZASC	Part of App	Monitor (0.5M) + OP-TEE (0.4M)	✓	✗	✓	✓
Sanctuary [24]	TZASC	App	Monitor (0.5M) + Runtime (0.5M)	✗	✓	✓	✓
CaSE [25]	TZASC	App	Trusted Boot (500) + Controller (500)	✗	✓	✓	✗
Ginseng [26]	EL3/Virtualization	Function	Monitor (0.5M)	✓	✓	✓	✗
CCA [3]	CCA GPC	VM	Monitor (0.5M) + RMM (33K) + OS in CVM (≥26M)	✗	✓	✓	✓
ACAI [27]	CCA GPC	VM	Monitor (0.5M) + RMM (23K) + OS in CVM (≥26M)	✗	✓	✓	✓
CAGE [28]	CCA GPC	VM	Monitor (0.5M) + RMM (26K) + OS in CVM (≥26M)	✗	✓	✓	✓
RContainer [7]	CCA GPC	Container	Monitor (0.5M) + Mini-OS (5K)	✗	✓	✓	✓
SHELTER [8]	CCA GPC	App	Monitor (0.5M)	✗	✓	✓	✓
CCAegis	**CCA GPC**	**Function**	Monitor (0.5M)	✓	✓	✓	✓

In the latest Armv9.2-A, Arm introduced the Confidential Compute Architecture (CCA) [3] along with a core hardware support feature known as the Realm Management Extension (RME) [6], as depicted in Figure 1. CCA introduces the realm world and the root world in addition to the existing TrustZone architecture. The TCB of CCA encompasses the OS within confidential virtual machines (CVMs) running in the realm world, the Realm Management Monitor (RMM) which is responsible for managing and isolating these CVMs, and the Monitor operating in the root world. RME introduces new data structures, such as the Granule Protection Table (GPT), and new hardware mechanisms, such as the Granule Protection Check (GPC), to enable fine-grained access control over physical memory. The GPT tracks the physical pages assigned to each world. Each entry in the GPT can be configured with one of six attributes, determining its association with a specific world—*normal*, *realm*, *secure*, or *root*—or set to universal accessibility (*all-access*) or complete inaccessibility (*no-access*). Typically, CCA uses a single GPT to manage attributes for all physical memory. In contrast, CCAegis constructs multiple GPTs, allowing different entities to enforce distinct access permissions over the same physical memory, thereby isolating specific memory regions. During each memory access, the CPU retrieves the attributes of the current memory page from the GPT and performs a GPC, as defined in Table I, to verify the access's legitimacy. If the access is deemed unauthorized, a Granule Protection Fault (GPF) is immediately triggered.

B. Trust Firmware-A

Trust Firmware-A (TF-A) [29] is a reference firmware implementation for Arm-based processors, responsible for initializing hardware and ensuring a secure boot process for CCA. Unlike TrustZone, which separates Exception Level 3 (EL3) into the secure world, CCA uses EL3 to establish the root world. TF-A operates in the root world and manages the isolation of the four worlds. With the highest privilege level, it prevents other worlds from modifying the configuration of the GPT and GPC. Given that TF-A (590K LoCs) is significantly smaller than the Linux kernel (26M LoCs), using it as the supervisory layer results in a smaller TCB.

III. MOTIVATION AND OVERVIEW

A. Motivation

Trusted Execution Environments (TEEs) can safeguard sensitive code and data without relying on trust in privileged software, and many approaches have explored isolation designs at varying granularities, as shown in Table II. However, they still face the following critical limitations:

Excessively Large TCB. Both native confidential computing solutions—including AMD's SEV [1], Intel's TDX [2], and Arm's CCA [3]—as well as extended proposals like ACAI [27] and CAGE [28] (which aim to address gaps in support for accelerators like GPUs) adopt VM-level isolation. However, these solutions require trust in the OS within the CVM, introducing millions of LoCs to the TCB when using the Linux kernel. In contrast, Sanctuary [24] uses the Zircon microkernel as the runtime for security-sensitive applications, though it still expands the TCB by approximately 500K LoCs. Notably, CCA further requires trust in the RMM to manage CVMs, adding over 20K LoCs to the TCB.

Overly Coarse Isolation Granularity. Other solutions directly isolate security-sensitive applications without requiring trust in the OS, maintaining a small TCB. For example, CaSE [25] isolates applications within the L2 Cache, while RContainer [7] and SHELTER [8] leverage multiple GPTs to prevent the OS from accessing protected container/application. However, their isolation granularity remains overly coarse, leaving them vulnerable to intra-process vulnerabilities [9].

Lack of Deployment/Porting Friendliness. Various TEEs provide fine-grained isolation for secure parts of applications across different architectures, including those on Intel [16], Arm [5], and RISC-V [23], [20], [21], [22]. However, these TEEs require developers to manually partition programs and adjust isolation interfaces, a process that can be cumbersome and error-prone. To improve deployment friendliness, several solutions [18], [26], [19] have developed automated program partitioning using program analysis technologies. However, Glamdring [18] and SeCage [19] rely on Intel-specific hardware features, limiting their portability on Arm platforms.

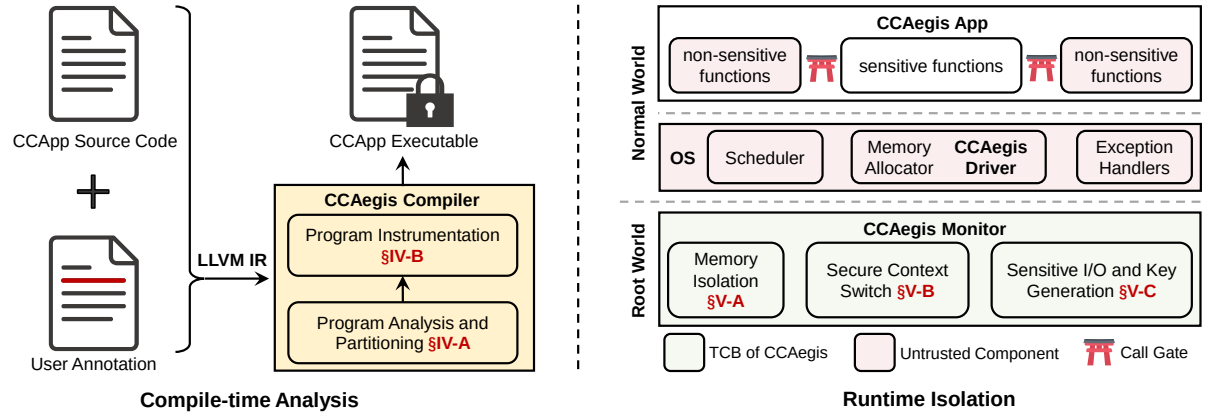

Fig. 2. The workflow of CCAegis' compile-time analysis and its runtime isolation architecture.

Ginseng [26], which uses registers for isolation, is constrained by the limited data volume it can protect.

In conclusion, most TEEs [3], [7], [8] exhibit overly coarse isolation granularity (e.g., VMs, containers, or applications). Even existing fine-grained TEEs face limited practicality due to their heavy reliance on manual developer effort [30], [21] or their use of caches [25] or registers [26] as isolation primitives.

B. CCAegis Architecture

We designed CCAegis to address the limitations of prior approaches, motivated by two key observations. First, manual effort can be minimized by requiring developers to annotate only a small set of initial sensitive data, with taint propagation analysis automatically identifying subsequent sensitive data and enforcing instrumentation-based protection. Second, memory-based isolation is more suitable than cache- or register-centric approaches for protecting general-purpose applications with large data volumes, and the GPT inherently enables memory access control without trusting the OS. Since the Secure Monitor is inevitably in the TCB, a stage-2/Realm-EL2 design would additionally pull the RMM into the TCB, whereas GPT keeps enforcement within the existing firmware boundary. Building on these findings, CCAegis operates through two fundamental stages, as depicted in Figure 2: Compile-time analysis (§IV) is conducted using the *CCAegis-Compiler*, while runtime isolation (§V) is managed by the *CCAegis-Monitor* located in the root world.

Compile-time Analysis. CCAegis's analysis consists of five steps to construct an executable with sensitive function isolation properties. Notably, this analysis is performed *offline*, making it immune to threats from runtime attackers. Initially, developers annotate the program with initial sensitive data and their final forms, establishing sources and sinks for subsequent analysis. Given the source code of a *CCAegis-App* (CCApp), the *CCAegis-Compiler* converts it into LLVM Intermediate Representation (IR) along with its call graph. It then performs points-to and taint analysis to track the propagation of sensitive data, identifying sensitive operations and the functions that contain them (§IV-A). The *CCAegis-Compiler* controls access to sensitive data by inserting call gates at the entry and exit points of identified sensitive functions. These gates invoke

```
1  void encrypt(char *plaintext, char *ciphertext,
2               char *key_path) {
3  ccaegis_grant_permission
4  char *key = malloc(KEY_SIZE); ⇨ ccaegis_malloc
5  /* Taint the initial key. */
6  #pragma taint_source(key)
7  ccaegis_revoke_permission
8  int fd = open(key_path, O_RDONLY);
9  ccaegis_grant_permission
10 read(fd, key, KEY_SIZE); ⇨ ccaegis_read
11 /* Perform encryption operation */
12 ciphertext = /* ... */;
13 /* Erase taints on the ciphertext */
14 #pragma taint_sink(ciphertext)
15 free(key); ⇨ ccaegis_free
16 ccaegis_revoke_permission
17 Return; }
#pragma User Annotation ccaegis_* Automatic Instrumentation
```

Fig. 3. An example of how CCAegis automatically generates instrumentation to protect cryptographic programs by leveraging developer annotations.

the *CCAegis-Monitor* to grant and revoke access permissions (§IV-B). Finally, the *CCAegis-Compiler* completes the compilation and linking processes to produce the executable.

Runtime Isolation. Basic capabilities such as scheduling and exception handling are managed by the untrusted OS. To bridge the semantic gap between CCApps and the *CCAegis-Monitor*, the driver we introduced allocates and maps virtual memory for sensitive functions. However, critical security operations—such as physical memory delegation and recycling (§V-A), state transitions between trusted and untrusted components (including transitions within intra-process and between CCApps and the OS) (§V-B), as well as sensitive I/O operations and key generation (§V-C)—are overseen and authorized by the *CCAegis-Monitor*.

Programming Model. Developers only need minimal annotations to utilize CCAegis's isolation capabilities, as depicted in Figure 3. They begin by marking the initial *taint source*, usually the user-defined cryptographic keys (line 6), and designate the point where data becomes ciphertext as the *taint sink* (line 14), indicating where taint propagation should halt. Following the taint propagation analysis, the *CCAegis-Compiler* identifies sensitive functions and automates the protection process. To isolate these functions, it inserts call gates at critical boundaries—specifically at the entry (line

3) and exit (line 16) of sensitive functions, and before (line 7) and after (line 9) calls to external non-sensitive functions. Note that CCAegis does not require explicit protection for file descriptors, as signature verification is enforced on file contents during read operations. It also replaces unsafe key-related functions with secure functions provided by the *CCAegis-Monitor* for memory allocation (line 4), memory release (line 15), and file loading (line 10).

C. Threat Model and Assumptions

The TCB of CCAegis only comprises the root world Monitor, which is verified through vendor signatures and securely loaded via secure boot. CCAegis trusts the hardware, assuming that Arm CCA and RME conform to their specifications. Its primary goal is to safeguard the confidentiality and integrity of keys within cryptographic programs, shielding them from intra-process adversaries. Additionally, it protects against privileged adversaries including the OS, hypervisor, and components within both secure and realm worlds. It also enforces mutual isolation, preventing malicious CCApps from attacking other CCApps and privileged software.

Attackers' capabilities. For intra-process adversaries, we consider their ability to exploit memory vulnerabilities, such as buffer overflows [9]. Additionally, they may execute control flow hijacking [31] to bypass the permission controls enforced by the call gates. For privileged adversaries, we account for more advanced key theft methods, including directly reading memory, register states, and I/O data from CCApps. These adversaries may also access key-related memory via Direct Memory Access (DMA) and hijack control flow by modifying register states or altering the CCApp's stack. Furthermore, we also focus on memory-oriented Iago vectors that directly impact confidentiality, including forged or aliased mappings, tampered pointers or lengths, and page-lifetime manipulations that could cause secrets to leave protected regions [32], [33].

Out of scope. We do not consider scenarios where sensitive functions intentionally leak secrets or security risks within sensitive function code, which could be mitigated by orthogonal techniques, such as software vulnerability detection [34]. Additionally, we exclude hardware attacks such as bus snooping [35] and DRAM analysis (e.g., Cold Boot [36] and Rowhammer attacks [37]). Arm CCA plans to implement the Memory Protection Engine (MPE) for memory encryption and possibly integrity protection, which should mitigate these hardware threats in the future. Denial-of-Service (DoS) attacks and side-channel attacks (e.g., Meltdown [38] and Spectre [39]) are also out of scope.

IV. CCAegis Compile-time Analysis

A. Program Analysis and Partitioning

Overview. We treat the point where user–defined cryptographic secrets enter a CCApp (for example, a key read from disk) as a *taint source*, and the locations where data becomes ciphertext or a signature as *taint sinks*; once data is in its final protected form, further tracking is unnecessary. Given sources and sinks, the *CCAegis-Compiler* performs points-to and taint analysis [40], [41] to follow secret propagation, identifies all variables and accesses on the secret path, and then classifies any `load` or `store` that touches these addresses as a *crypto operation*. A function is a *crypto function* if it contains at least one crypto operation. Isolation is enforced at function granularity over this discovered set.

Taint IR. Because standard compiler IRs such as LLVM IR do not provide native dataflow tracking, we introduce a *Taint IR* layered over the compiler IR. Taint IR (i) normalizes constructs across IRs so that results can be reapplied to the original IR without binding to a specific compiler, (ii) tags instructions with context–sensitive identifiers to make the analysis context aware, and (iii) incorporates *index–based* modeling that records byte–range offsets for structured and array objects. The index model mitigates over–tainting common in classical Andersen–style analysis by tracking structure elements precisely.

For any global or local variable v within a function, we define its corresponding **points-to analysis instructions**, denoted as v_{pto}, as the set of tuples $\{(l_v, i)\}$. Here, l_v represents the memory location it point to, and i is its containing member variables as byte range index. The points-to analysis IR set includes the following instructions:

1) **alloc** v, len. This instruction creates a memory location l_v pointed to by variable v with memory size len. The value of len is related to the memory representation, which we assume remains unchanged within one module during analysis.
2) **load** p, q. This instruction reads the memory location pointed to by address variable q and writes its contents into p.
3) **store** p, q. This instruction writes the contents of variable q into the memory area pointed to by address variable p.
4) **index** p, q, i. This instruction extracts the memory location pointed to address q with index range parameter i, and writes location into result address p.
5) **merge** $p, \{q_1, ..., q_n\}$. This instruction merges the contents of variables $q_1, ..., q_n$ into result variable p according to specific rules, usually corresponding to unary and binary operations or bit conversions in compiler IR.
6) **call** $f(a_1, ..., a_n) \rightarrow b$. This instruction calls function f with arguments $a_1, ..., a_n$ and writes the return value into variable b.
7) **return** r. This instruction uses variable r as the return value and terminates the current function's execution.

From these instructions we derive index aware points to constraints (see Figure 4). Taint analysis results are derived from points-to analysis results. To incorporate source annotations, the Taint IR adds two **taint analysis instructions**:

1) **taint** p. This instruction marks the memory location pointed to by p as tainted. Note that the points-to set p_{pto} includes memory locations and their corresponding indices. This instruction only taints the memory location within the specified index range, but not beyond it.
2) **erase** p. This instruction erases the taint paths associated with memory locations and indices pointed by p.

According to the above definitions, we produce Taint IR,

$$\frac{\textbf{alloc}\, v, len}{(l_v, [0, len)) \in v_{pto}}\ \text{Alloc} \qquad \frac{\textbf{load}\, p, q\ (l_r, i) \in q_{pto}\ (l_x, j) \in r_{pto}}{(l_x, \text{INDEX}(j, i)) \in p_{pto}}\ \text{Load} \qquad \frac{\textbf{store}\, p, q\ (l_r, i) \in p_{pto}\ (l_x, j) \in q_{pto}}{(l_x, j) \in r_{pto}}\ \text{Store}$$

$$\frac{\textbf{index}\, p, q, i\ (l_x, j) \in q_{pto}}{(l_x, \text{INDEX}(j, i)) \in p_{pto}}\ \text{Index} \qquad \frac{\textbf{merge}\, p, \{q_1, \ldots, q_n\}\ (l_x, i) \in q_{k,pto}}{(l_x, i) \in p_{pto}}\ \text{Merge} \qquad \frac{c : \textbf{call}\, f(a_1, \ldots, a_n) \to b,\ c' : f(p_1, \ldots, p_n) \to r \quad (l_x, i) \in c : a_{k,pto}\ \ (l_y, j) \in c' : r_{pto}}{(l_x, i) \in c' : p_{k,pto}\ \ (l_y, j) \in c : b_{pto}}\ \text{CallRet}$$

Fig. 4. Index assisted points-to analysis constraints in Taint IR.

Algorithm 1: Taint Analysis Algorithm

Data: Points-to sets $\{v_{pto}\}$, taint analysis result $\{v_{taint}\}$, current Taint IR instruction I
Result: Modified taint analysis result $\{v_{taint}\}$

switch *type of I* **do**
  case *load p, q* **do**
    forall $(l_r, i) \in q_{pto}$, $(l_x, j) \in r_{pto}$, $(s, k) \in x_{taint}$ **do**
      $p_{taint} \leftarrow p_{taint} \cup \{(\texttt{concat}(s, I), j \cap k)\}$
    end
  case *store p, q* **do**
    forall $(l_r, i) \in p_{pto}$, $(l_x, j) \in q_{pto}$, $(s, k) \in x_{taint}$ **do**
      $r_{taint} \leftarrow r_{taint} \cup \{(\texttt{concat}(s, I), i)\}$
    end
  case *index p, q, i* **do**
    forall $(s, j) \in q_{taint}$ **do**
      $p_{taint} \leftarrow p_{taint} \cup \{(\texttt{concat}(s, I), i \cap j)\}$
    end
  case *merge* $p, \{q_1, ..., q_n\}$ **do**
    forall $q_k \in \{q_1, ..., q_n\}$, $(s, i) \in q_{k,taint}$ **do**
      $p_{taint} \leftarrow p_{taint} \cup \{(\texttt{concat}(s, I), i)\}$
    end
  case *taint p* **do**
    forall $(l_r, i) \in p_{pto}$ **do**
      $r_{taint} \leftarrow \{(\texttt{makeseq}(I), i)\}$;
  case *erase p* **do**
    forall $(l_r, i) \in p_{pto}$ **do** $r_{taint} \leftarrow \varnothing$;
  case *call* $f(a_1, ..., a_n) \to b$ **do**
    $c' : p_{k,taint} \leftarrow c : a_{k,taint}$
  case *return r* **do**
    $c : b_{taint} \leftarrow c' : r_{taint}$, where b is the return variable in corresponding *call* instruction
end

which are converted from compiler IRs. Note that Taint IR and compiler IRs do not necessarily correspond one-to-one. For instance, the **index** instruction determines the *i* parameter from a list of indices and the module-level memory layout in one compiler IR instruction.

Taint Analysis Algorithm. We detail our taint analysis algorithm in algorithm 1. The algorithm starts at an entry function linearly scan our transferred point-to and Taint IR instructions of program. Note that the different locations call instructions correspond to distinct contexts c, making our analysis algorithm context-sensitive. Finally, we give a definition of taint analysis results for each variable. The **taint analysis result** for a variable v is defined as $v_{taint} = \{(s, k)\}$, comprising pairs of the taint propagation flow s and the corresponding memory location index k. Here, the taint propagation flow $s = (I_1, I_2, ..., I_n)$ represents the sequence of all Taint IR instructions I involved in the propagation chain of v. The taint analysis results will encompass all tainted local and global variables and memory locations. These can then be utilized as sensitive variables and operations in the following instrumentation process.

B. Program Instrumentation

Instrumentation for security-sensitive applications must enforce two critical guarantees. First, only crypto functions are granted access to protected memory regions. Second, secure interactions between crypto functions and non-crypto contexts must be ensured, including: (1) Access to non-sensitive global variables, (2) Invocation of non-crypto functions within CCApps, (3) Execution of untrusted system calls.

Secure Memory Management. Given that the minimum isolation granularity of the GPT is 4KB, CCAegis optimizes memory usage by reallocating cryptographic keys to a contiguous crypto buffer. These cryptographic keys are typically scattered across a process's memory, residing in the stack, heap, and global variables. To manage these keys efficiently, the *CCAegis-Monitor* establishes the crypto stack, crypto heap, and crypto data segment—each contained within the crypto buffer. To secure access to *non-sensitive* global variables from crypto functions, the *CCAegis-Compiler* copies the values of these global variables into the crypto stack. It then verifies and accesses these values directly from the crypto stack.

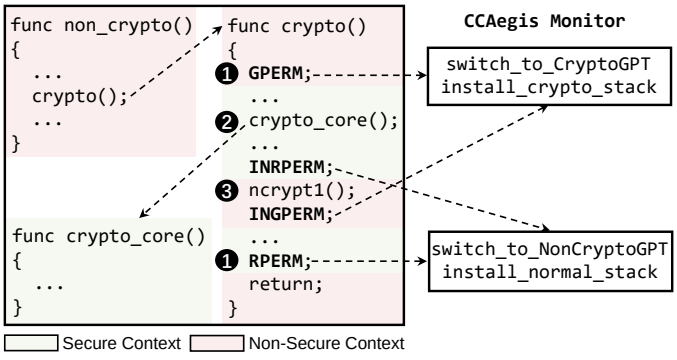

Fig. 5. Execution flow through call gate instrumentation.

Call Gate Generation. The *CCAegis-Compiler* isolates crypto functions by strategically inserting call gates. When entering a crypto function, the *CCAegis-Monitor* switches to the Crypto GPT to grant access to the crypto buffer, and swaps out the normal stack for a crypto stack. Conversely, upon leaving the

crypto function, the *CCAegis-Monitor* switches to the Non-Crypto GPT to revoke access, reverts to the original normal stack, and saves the crypto stack pointer. This process is detailed in three steps, as depicted in Figure 5: ❶ *Function Entry and Exit:* For each crypto function, the *CCAegis-Compiler* inserts a Grant Permission (`GPERM`) call gate at the function entry and a Revoke Permission (`RPERM`) call gate before the function returns to ensure the function operates within a secure context. ❷ *Nested Crypto Functions:* If a crypto function is called by another crypto function, additional call gates at the entry and exit of the callee are unnecessary, as the secure context is already established by the caller. ❸ *Interactions with Non-Crypto Functions:* For non-crypto calls within a crypto function, the compiler inserts an Internal Revoke Permission (`INRPERM`) call gate before the call to isolate the context and an Internal Grant Permission (`INGPERM`) call gate after the call to re-establish the secure context.

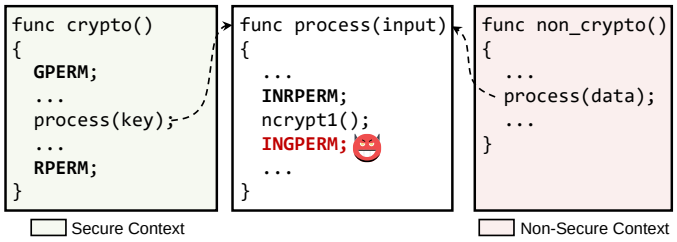

Fig. 6. Call gate exploitation when called by different contexts.

Context-Sensitive Isolation. A function may be identified as either a crypto or non-crypto function depending on the context in which it is called, as illustrated in Figure 6. For example, when called by `crypto()` with a key as an argument, `process()` functions as a crypto function. In this case, inserting call gates at the entry or exit of `process()` would be redundant. However, if `process()` contains non-crypto functions, it still requires `INRPERM` and `INGPERM` to secure the crypto context. Conversely, when `non_crypto()` calls `process()`, the untrusted `non_crypto()` function could exploit the `INGPERM` instruction within `process()` to gain access to the crypto buffer. To mitigate this risk, the *CCAegis-Compiler* uses context-sensitive analysis to generate different versions of `process()`. This ensures that only the version invoked within a crypto context is treated as a crypto function, and it is properly fortified with call gates.

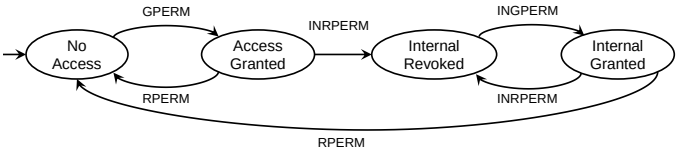

Fig. 7. State machine for call gates.

Call Gate Hardening. Attackers can potentially hijack non-crypto functions to jump to any position within crypto functions, including jumping directly to `INGPERM`. To secure that `INGPERM` is invoked through legitimate control flow, the *CCAegis-Monitor* enforces state transitions of call gates as per a defined state machine, depicted in Figure 7. The execution of `INGPERM` is valid only after passing sequentially through the *Access Granted* and *Internal Revoked* states, ensuring control flow integrity of crypto functions. Moreover, to secure that executing `INGPERM` to re-enter the crypto function corresponds strictly to its paired `INRPERM`, the *CCAegis-Monitor* verifies the address of `INGPERM` before executing it.

Key-Related System Call Replacement. Since the OS is untrusted, system calls within crypto functions must be replaced with secure implementations. For memory management, while the OS remains responsible for allocating and mapping the crypto buffer, this memory is delegated to the *CCAegis-Monitor*, which manages the available memory within the buffer. As a result, the *CCAegis-Compiler* replaces the native `malloc` and `free` with secure versions implemented by the *CCAegis-Monitor*. Similarly, for all I/O operations involving keys and random number generation, the *CCAegis-Monitor* provides secure implementations, as detailed in §V-C.

Iago Attack Defense. The OS can achieve Iago attacks [32], which could deceive the crypto function by manipulating the return values of system calls. Since non-sensitive memory allocation still uses native system calls (e.g., `mmap`), there's a risk that the returned memory area might overlap with the crypto buffer. To mitigate this, CCAegis performs two key actions: it checks page table updates to prevent double mapping of the crypto buffer (§V-B) and inserts runtime checks to ensure the returned memory does not overlap with the crypto buffer. Furthermore, system calls that return length data, such as `recv(socket, buf, len, flags)`, are verified to prevent exceeding the buffer's maximum length.

V. CCAegis Runtime Isolation

A. Memory Isolation

To minimize the TCB, memory allocation is managed by the untrusted *CCAegis-driver*. The *CCAegis-Monitor* then delegates this memory to CCApps by updating the GPT. However, GPT-based memory isolation faces three critical challenges:

Multi-Core Isolation. CCA typically allows multiple cores to share the same GPT, maintaining a unified memory view. However, this approach can lead to permission conflicts: crypto functions on one core require different access permissions for the crypto buffer compared to the OS or non-crypto logic running on other cores. To address this, CCAegis introduces three distinct types of GPTs, as depicted in Figure 8. The Host GPT is designated for untrusted privileged software like the OS. Each CCApp is assigned a Crypto GPT for handling keys within crypto functions, and a Non-Crypto GPT for non-crypto functions. By configuring each core with its own GPT, CCAegis ensures that each core has a unique view of physical memory access permissions. The *CCAegis-Monitor* is responsible for initializing these GPTs and dynamically switching between them based on the execution context.

Multi-GPT Maintenance. The multi-GPT mechanism disrupts CCA's native unified permission management, and the coexistence of these tables introduces additional requirements for consistency maintenance and security constraints. First, accessing these GPTs requires acquiring a spinlock to ensure the GPTs' views are not outdated. We use the Host GPT as

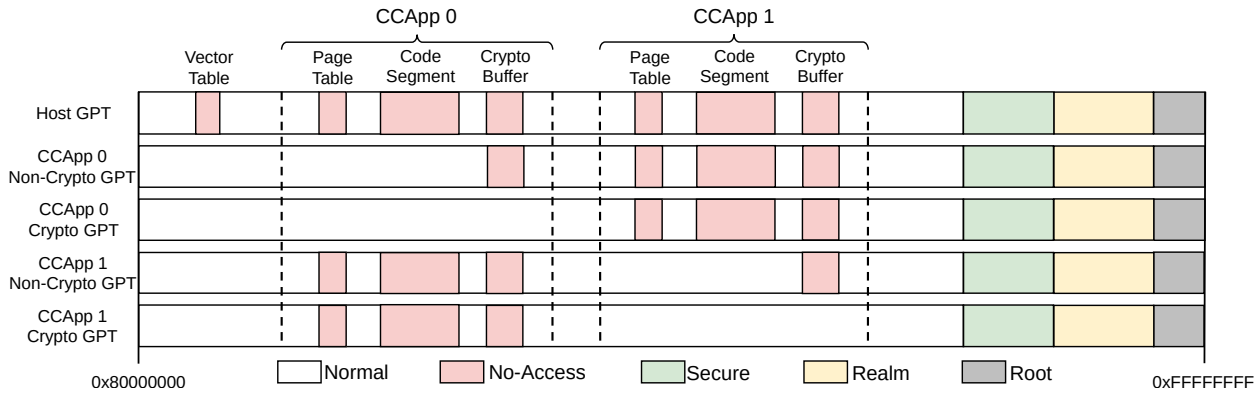

Fig. 8. Physical memory access control views for privileged software, crypto functions, and non-crypto functions.

a template, as it is configured to deny access to the isolated memory of all CCApps, requiring only minimal adjustments to create the GPTs for each CCApp. To ensure that the delegated memory does not overlap—whether delegated to CCApps or to the original TEEs (including the secure world and realm world)—we stipulate that only *normal* memory in the Host GPT can be delegated. Additionally, to enforce isolation between CCApps, the *CCAegis-Monitor*, when delegating memory to a CCApp, configures that memory as *non-access* in the GPTs of all other CCApps. Similarly, to prevent CCApps from accessing memory in the original TEEs, the *CCAegis-Monitor* sets the memory as *realm* or *secure* in the GPTs of all CCApps when memory is delegated to these TEEs.

Mitigating the Semantic Gap. Since GPTs control access to physical memory while CCApps operate with a virtual address view, the OS might tamper with page tables to map sensitive areas like the crypto buffer to unprotected memory regions. To bridge this gap, after the OS creates the page tables, the *CCAegis-Monitor* first sets these tables to *no-access* in the Host GPT and then scrutinizes the mappings in these tables; this order removes any TOCTTOU window [42]. It performs three critical checks: (1) *Single Mapping Verification*: It ensures that the code segment and crypto buffer are correctly and uniquely mapped to the delegated physical memory. This setup prevents code-reuse attacks caused by page-oriented programming [43] and protects against Iago attacks [32]. (2) *Table Protection*: It confirms that the page table and vector table are protected from being maliciously mapped, defending against tampering by the untrusted logic of CCApps. (3) *Executable Space Control*: The code segment is verified to be read-only and executable, while ensuring no other memory areas are executable, preventing code injection attacks. Additionally, if a page fault occurs during CCApp execution, CCAegis employs a shadow page table mechanism to enable the OS to securely update the CCApp page table, as further detailed in §V-B.

B. Secure Context Switch

In this section, we detail how CCAegis handles call gate execution to achieve intra-process isolation and how it intercepts exceptions to effectively isolate CCApps from the OS.

Call Gate Handling. The *CCAegis-Monitor* identifies the call gate executor using the GPT base address. Since the OS and untrusted user-space applications utilize the Host GPT, their attempts to invoke call gates are effectively blocked. We discuss the handling of call gates from three aspects:

(1) *Call Gate Construction:* Call gates are initiated with a Supervisor Call (`svc`) that traps to EL1, followed by a Secure Monitor Call (`smc`) in the vector table that traps to EL3. Typically, the `svc` calling convention uses an immediate number set to 0, with the call number passed through the `x8` register. To prevent tampering with the register to manipulate call gates, our call gates use a different convention where the call number is passed directly through an immediate number (e.g., `svc #4`). Additionally, the *CCAegis-Compiler* ensures that all original `svc` immediate values in the CCApp, before instrumentation, are set to 0 to prevent conflicts.

(2) *Call Gate Routing:* The call gate routing is implemented through the previously delegated vector table. Upon trapping to EL3 via `svc` and `smc`, the *CCAegis-Monitor* switches to the relevant Crypto GPT or Non-Crypto GPT based on the current context of the CCApp. After making this switch, it bypasses the OS and directly returns control to the CCApp.

(3) *Register State Sanitization:* This process includes entry status flag sanitization and exit register leakage prevention. At the entry of crypto functions, the *CCAegis-Monitor* configures the `PSTATE` register by clearing all condition flags and ensuring key status bits are set as *invariants*, such as ensuring the debugging bit is disabled. At the exit of crypto functions, the *CCAegis-Monitor* cleans the general register states to prevent any inadvertent leakage of sensitive data.

Regular Exception Interception. CCAegis also intercepts regular exceptions through the delegated vector table, ensuring that any execution flow leaving the CCApp first enters the *CCAegis-Monitor* for isolation from the OS. We execute three crucial operations to enforce this isolation:

(1) *GPT Switching*: To prevent OS access to isolated memory, the *CCAegis-Monitor* switches to the Host GPT prior to entering the OS and switches back to the relevant GPT upon entering the CCApp. Note that even if the OS directly jumps back to the CCApp, this will trigger a GPF due to lacking permissions to the code and crypto buffer of the CCApp.

(2) *Sensitive Context Cleanup*: To protect against the OS spying on register states via exceptions or modifying the exception return address to hijack control flow, the *CCAegis-*

Monitor preserves sensitive context of the crypto function. This preservation includes general registers, the exception link register, and the saved program status register. Before handing control to the OS, it clears the general registers (except for the parameter registers, frame pointer, and link registers) and later restores the saved sensitive context upon re-entering the CCApp. Furthermore, to ensure that exceptions during CCApp execution are consistently intercepted, the delegated vector table is reactivated each time the CCApp is re-entered.

(3) *CCApp Page Table Updating*: To allow normal updates to the CCApp page tables, the OS prepares a shadow page table before delegating the CCApp page table. When handling a page fault, the *CCAegis-Monitor* temporarily switches to this shadow page table, permitting the OS to update it normally. Before resuming CCApp execution, the *CCAegis-Monitor* reverts to the delegated CCApp page tables and synchronizes changes made to the shadow page table. It also conducts checks on new mappings, similar to those performed during initial CCApp page table delegation. Importantly, the *CCAegis-Monitor* ensures that newly mapped memory does not overlap with the crypto buffer, particularly the crypto stack. This verification ensures that the runtime checks (§IV-B) for system call return values, effectively defend against memory-based Iago attacks [32]. Further defenses against other Iago attacks can incorporate established methods [44].

C. Sensitive I/O and Key Generation

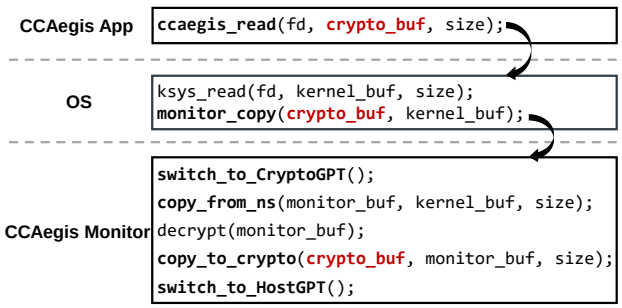

Fig. 9. Encapsulated key loading of CCAegis.

Since the OS is untrusted, the *CCAegis-Monitor* implements protections for key generation, import, and export within CCApps. For key generation, the process follows the call gate routing, which bypasses the OS and is managed directly by the *CCAegis-Monitor*. By utilizing the Random Number Generator (RNG) feature in Armv8.5-A, the *CCAegis-Monitor* generates true random numbers by reading the `RNDR` register and writing its value into the crypto buffer. Key import from the file system, as depicted in Figure 9, begins with the CCApp requesting the OS to read encapsulated keys from a file. Since the OS operates under the Host GPT, it must request the *CCAegis-Monitor* to decrypt the keys and transfer them into the crypto buffer. To safeguard against unauthorized access: (1) Before decrypting the keys, the *CCAegis-Monitor* verifies the file signature to ensure it belongs to the current CCApp, thus preventing the `monitor_copy` interface from being exploited as a decryption oracle. (2) The *CCAegis-Monitor* implements the `copy_from_ns` to verify that the kernel buffer being copied is classified as *normal* memory in the Host GPT, preventing the OS from using this interface to steal secrets from other worlds or the crypto buffer. (3) It also implements the `copy_to_crypto` to prevent key leakage to unprotected memory. The process for exporting keys follows a similar procedure and is therefore not detailed here.

D. Multi-threading Support.

CCAegis supports thread synchronization primitives (such as locks, conditional variables, and semaphores) based on Fast User-space muTEX (Futex). The OS can access the Futex in the CCApp memory normally since this variable is located outside the crypto buffer. Moreover, before executing external calls such as `FUTEX_WAIT`, CCAegis executes `INRPERM` to prevent potential key leakage attacks (§IV-B). For each new CCApp thread, a separate crypto stack is allocated. The *CCAegis-Monitor* checks the mappings of these stacks, similar to checks in the page table delegation (§V-A). During exception handling, the *CCAegis-Monitor* saves the sensitive state for each thread and restores the appropriate sensitive state based on the thread ID before re-entering the CCApp. To support concurrency efficiently, each core maintains its own GPT view so a domain switch is per core: the *CCAegis-Monitor* writes that core's GPT base, performs a local TLB invalidation, and returns to user mode. This hot path is lock free across cores, and we disable TLB sharing (clear CnP) to avoid cross-core shootdowns during switches. A global spinlock is taken only when GPT structures are updated or memory is delegated, which occurs infrequently, and multi-GPT maintenance templates new GPTs from the Host GPT to keep update time short (§V-A).

VI. Implementation

Current hardware lacks Arm CCA support, prompting development of both functional and performance prototypes.

Functional Prototype. We implemented a prototype on the Arm Fixed Virtual Platform (FVP) [45], which supports RME and RNG features, to validate the design of CCAegis and confirm its compatibility with future hardware. FVP has been used in previous works [8], [27], [28] to test the functional correctness of CCA. The static analysis module of CCAegis utilizes LLVM to analyze and transform the source code of CCApps, integrating a taint analyzer and an LLVM Pass, adding 2k LoCs. Our analysis preserve a fail-closed default: when precision is insufficient at a boundary, we conservatively mark it sensitive. If a call target on a tainted path cannot be resolved statically, including calls reached via `dlopen`/`dlsym`, we treat that site as a sensitive boundary, auto-insert domain switches, and place outputs and reachable buffers in the protected region; this conservative choice may enlarge the sensitive slice, which we mitigate with function-boundary hoisting and batching (§IV-B). Additionally, CCAegis modifies the Linux kernel 6.6-rc6 and introduces a Linux driver, adding another 1k LoCs. This driver manages the lifecycle of CCApps, interacting with the *CCAegis-Monitor* via `smc` to handle the creation and destruction of CCApps, and the

creation of new threads. Furthermore, CCAegis develops a loader that utilizes the `ioctl` interface provided by the driver to load instrumented CCApps. The driver is also responsible for CCApp memory allocation and mapping, utilizing the Contiguous Memory Allocator (CMA) to manage three distinct contiguous physical memory areas assigned for the code segment, page table, and crypto buffer. Upon driver initiation, the Linux kernel completes the vector table registration, ensuring that any control flow exiting the CCApp is initially intercepted. In the root world, the *CCAegis-Monitor* exposes a narrow SMC ABI that configures/switches GPT domains and verifies shadow mappings (no scheduler, filesystem, or device stacks). This fail-closed design keeps the trusted code path compact and auditable, and in practice amounts to a small, targeted firmware extension, not a rewrite.

Performance Prototype. As the FVP is not cycle-accurate, we evaluate performance using the Rock Pi 4B development board (Armv8.0-A). The cores used in this development board are consistent with those in previous works [8], [28], [27]. However, since the hardware features (i.e., GPC and RNG) used in this paper are not supported on this development board, we simulate these features as follows: (1) Following previous work [27], we utilize idle registers in EL3—specifically, `afsr0_el3` and `afsr1_el3`—to substitute for the `gpccr_el3` (the GPC control register) and `gptbr_el3` (the GPT base register) for evaluating the overhead of GPC configuration and GPT base address switching. Since the GPT is an in-memory structure, we consistently implement and assess the overhead of creating, maintaining, and destroying the GPTs like the functional prototype. (2) We replace the TLB maintenance instruction (`TLBI PAALLOS`, not available on our board), which typically only invalidates all cached GPT information in the TLB, with an instruction that invalidates the entire TLB cache. (3) We generate a seemingly random 64-bit key using the stack pointer (`sp`), the link register (`x30`), and the counter-timer physical count register (`cntpct_el0`) to simulate the RNG feature.

VII. Evaluation

To evaluate the security and performance of CCAegis, we aim to answer the following five questions:

Q1: How large is the TCB of CCAegis?
Q2: Can CCAegis defeat privileged and intra-process attacks?
Q3: What is the compilation overhead of our static analysis?
Q4: What is the performance cost of our fine-grained isolation measured on micro-benchmarks and real-world applications?
Q5: What is the root cause of the performance overhead?

A. Experimental Setup

We evaluate the security of CCAegis on the functional prototype implemented on FVP, answering **Q1** and **Q2**. We use LLVM 10.0.1 with -O2 optimizations to implement static analysis and evaluate the compile-time overhead to answer **Q3**. It operates on an Arm server powered by a 96-core HiSilicon Kunpeng-920 CPU (2.6 GHz, Armv8.2-A) and is equipped with 256 GB RAM. We assess the performance overhead of CCAegis on the performance prototype implemented on the Rock Pi 4B development board, answering **Q4** and **Q5**. This board is built around a Rockchip RK3399 SoC, featuring two Arm Cortex-A72 cores (running at speeds of up to 1.8GHz) and four Cortex-A53 cores (up to 1.4GHz). To avoid performance measurement discrepancies caused by the big.LITTLE architecture of this SoC, we set the Cortex-A72 cores to run at the highest frequency and disabled all Cortex-A53 cores.

B. Q1: TCB Size of CCAegis

TABLE III
TCB MODIFICATIONS OF CCAEGIS.

Component	Lines of Code (LoCs)
Memory Isolation	990
Exception Handling Interception	483
Secure I/O and Key Generation	154
Other Configuration	176
All	**1803**

We run the `cloc` tool to measure the code size of CCAegis. The TCB of CCAegis is confined to the Monitor located in the root world, which uses Trusted Firmware-A v2.9.0 [29], accounting for 479K LoCs. CCAegis introduces an additional 1,803 LoCs of modifications, as detailed in Table III. Unlike CCA's heavier TCB, CCAegis omits Linux cca-guest-v2 [46] within CVMs, with its 26M LoCs, and TF-RMM v0.4.0 [47], which includes 33K LoCs.

C. Q2: Security of CCAegis

TABLE IV
MAIN ATTACKS AND DEFENSES OF CCAEGIS.

Attack	CCAegis defence
From privileged software:	
At creation-time	
❶ Incorrect binary loading	Monitor checks
❷ Overlapped memory delegation	GPT access synchronization
At runtime	
❸ CPU GPC circumvention	EL3 isolation and TLB invalidation
❹ Unauthorized memory access	Memory delegation and GPC
❺ Malicious I/O and entropy	System call replacement
❻ Exception interception circumvention	Vector table isolation and GPC
❼ Register leakage	Exception context sanitization
❽ Illegal mappings and Iago	Monitor checks and runtime checks
❾ Call gates invocation	Monitor checks
At destruction-time	
❿ Sensitive data remanence	Monitor cleanup
From non-crypto functions:	
❶ Unauthorized memory access	Page table checks and GPC
❷ Control flow hijacking	Monitor checks and GPC
❸ Register leakage	Call gate context sanitization
From malicious CCApp:	
❶ Collusion with OS	Monitor checks and GPC
From peripherals:	
❶ Malicious DMA	SMMU GPC
❷ Peripheral GPC circumvention	EL3 isolation and TLB invalidation

As shown in Table IV, we categorize the security risks and demonstrate how our defense measures address them.

Privileged Software. CCAegis protects CCApp initialization from the untrusted OS. Specifically, ❶ after the untrusted OS loads the CCApp binary, the *CCAegis-Monitor* verifies the loading address and the integrity of the binary. ❷ Before allocating memory to CCApps or the original TEEs, the *CCAegis-Monitor* ensures the memory is allocable by consulting the

TABLE V
THE NUMBER AND TIME OF ANNOTATIONS, TCB SIZE, CALL GATES, SYSCALL REPLACEMENTS, AND THE COMPILE TIME FOR CCAEGIS.

Application	Number of Tag (Source + Sink)	Annotation Time (h)	Total LoC	Crypto LoC	Total Function	Crypto Function	False Positive	Call Gate	Internal Call Gate	Syscall Replace	Original Time (s)	CCAegis Time (s)
`wolfSSL`	8 + 5	5.48	174067	41384 (23.77%)	2700	343 (12.70%)	46 (1.91%)	704	2534	3	34.41	103.06
`ccrypt`	6 + 3	2.25	5670	1156 (20.38%)	39	21 (53.85%)	3 (14.29%)	295	126	12	2.52	15.84
`libhydrogen`	2 + 4	2.85	2995	1056 (35.26%)	75	40 (53.33%)	3 (7.89%)	108	236	2	2.33	4.43
`libxcrypt`	3 + 2	1.72	16649	1759 (10.57%)	121	8 (6.61%)	0 (0.00%)	172	180	3	12.34	21.95

Host GPT. To avoid memory overlaps due to outdated views, it synchronizes access to all GPTs across cores (§V-A).

We further outline the corresponding attacks and countermeasures during the execution of CCApp. ❸ Since the GPT is stored in the root world and the GPC registers reside at the EL3 level, privileged software cannot disable the GPC or modify the GPT. Given that the GPT may be cached in the TLB, the *CCAegis-Monitor* invalidates the TLB during GPT updates or when switching GPT base addresses. As the TLB may be shared across multiple cores, the *CCAegis-Monitor* clears the `CnP` bit in the `TTBR` registers to prevent TLB sharing. ❹ Memory areas like the code, crypto buffer, and page tables of the CCApp are isolated with GPTs such that unauthorized access by privileged software will trigger a GPF. ❺ All random number generation and key-related I/O are secured by implementations in the *CCAegis-Monitor*, and direct reads of encapsulated key files do not result in leakage (§V-C). ❻ During CCApp execution, the *CCAegis-Monitor* intercepts exceptions via a delegated vector table, managing GPT switching to ensure proper isolation (§V-B). Modifications to this vector table will also trigger a GPF. To guarantee that this interception mechanism is not circumvented, the *CCAegis-Monitor* activates the vector table each time the CCApp is re-entered. ❼ The *CCAegis-Monitor* also sanitizes the sensitive register states of crypto functions during interceptions to prevent key leakage. Additionally, it manages exception returns to prevent tampering with return addresses. ❽ Furthermore, the *CCAegis-Monitor* reviews page table updates to block malicious mappings and the *CCAegis-Compiler* integrates runtime checks of system call return values to counter Iago attacks [32]. ❾ To prevent privileged software from calling the interfaces provided by the *CCAegis-Monitor*, such as calling the call gate to grant access to the crypto buffer, the *CCAegis-Monitor* checks `gptbr_el3`.

CCAegis also protects the destruction phase. Specifically, ❿ upon destroying a CCApp, CCAegis clears its crypto buffer, cache, and register states, then releases the delegated physical memory areas (i.e., set as *normal*) from the GPTs of the host and other CCApps, and finally destroys the CCApp's GPTs.

Non-Crypto Functions. ❶ The *CCAegis-compiler* inserts call gates into crypto functions to ensure that any access to the crypto buffer by non-crypto functions triggers a GPF. As depicted in Figure 8, the vector table, page table, and CCApp code are not marked as *no-access* in the Non-Crypto GPT. Instead, the *CCAegis-Monitor* safeguards these components by checking updates to the page table (§V-B). ❷ To prevent attackers from gaining access to the crypto buffer through control flow hijacking, the *CCAegis-compiler* inserts `INRPERM` before all untrusted calls to revoke access permissions. Moreover, the *CCAegis-Monitor* manages the state transitions of call gates to ensure that call gates are invoked through legitimate control flow (§IV-B). ❸ The *CCAegis-Monitor* clears the registers before exiting the crypto function or invoking untrusted calls to prevent key leakage.

Malicious CCApp. ❶ An CCApp can potentially collude with the OS to allocate memory from other CCApps or worlds to itself. The *CCAegis-Monitor* verifies the memory to be delegated, ensuring no overlap occurs.

Peripherals. ❶ Attackers can use DMA to bypass the CPU GPC and directly access isolated memory. The *CCAegis-Monitor* utilizes the SMMU to enable GPC for all peripherals. It configures the Host GPT for SMMU and ensures that peripherals maintain the lowest access permissions. ❷ The SMMU GPC can only be configured by the root world. When modifying the Host GPT, the *CCAegis-Monitor* also invalidates the SMMU TLB, preventing peripheral GPC bypass.

D. Q3: Evaluation on Static Analysis

We considered four widely used cryptographic libraries (i.e., `wolfSSL`, `ccrypt`, `libhydrogen`, `libxcrypt`) to test the static analysis. For each library, we evaluated the number of manual annotations as well as the results of automated program partitioning and instrumentation.

Annotated Tags. We designate the keys in cryptographic algorithms as *taint sources* and the outputs of these algorithms (including encrypted ciphertext, decrypted plaintext, and signatures) as *taint sinks*. The LoCs and time required for annotations in the cryptographic libraries are described in Table V. We invited five graduate students to understand the semantics of API parameters and annotate them. For most libraries, the task took on average less than three hours (experienced developers could likely complete it faster). For larger cryptographic libraries with extensive APIs, such as `wolfSSL`, the annotation process required under six hours. This level of effort is acceptable, which indicates that CCAegis has good practicality and deployment friendliness. For `wolfSSL`, we added annotations in 13 LoCs, which include **eight** *taint sources* on keys used in the `AesGcm` and `AesCfb` APIs, along with their associated `Aes` context structures. Additionally, we annotated **five** *taint sinks* on ciphertext variables associated with these contexts. In `ccrypt`, we applied annotations in **six** LoCs for *taint sources* on command-line key inputs (`cmd.keyword`) and internal cryptographic states, such as `block2` in `xrijndaelEncrypt`, plus **three** *taint sinks* for the stream inputs of `streamhandler` functions that process encrypted data. For `libhydrogen`, the annotations were minimal, involving only **two** *taint sources* on secret keys in the `hydro_sign_create` and `hydro_secretbox_setup`, and **four** *taint sinks* on their output buffers, signatures,

and internal states. In `libxcrypt`, we used **three** *taint sources* on `HMAC` keys and **two** *taint sinks* on the outputs of `hmac_sha1_process_data` functions.

False Positive Analysis. As noted in prior studies [48], [49], points-to analysis in C programs is inherently imprecise due to its undecidability [50]. Consequently, some non-crypto functions may inadvertently be included within the crypto function set, resulting in false positives. While this leads to slight over-protection, it ensures no genuine crypto functions are missed, thereby preventing key leakage. As shown in Table V, for `wolfSSL`, 343 functions (12.70%) were identified as crypto functions, constituting only 23.77% of the total LoCs. Similar trends are observed in other cryptographic libraries, where crypto code ranges between 10.57% and 35.26% of the total LoCs. In the absence of ground truth, we manually verified false-positive instances. Our evaluation revealed zero false positives for `libxcrypt`, and a low false-positive rate of 1.91% for `wolfSSL`. Both `ccrypt` and `libhydrogen` exhibited higher false-positive rates (14.29% and 7.89%, respectively), primarily due to their small total number of crypto functions—indeed, only three functions across these libraries were incorrectly flagged as crypto functions.

Compilation Overhead. We evaluated these workloads by focusing on three aspects: the number of call gates inserted, the number of system call replacements, and the overall compilation time. Regarding call gate insertion, `wolfSSL` exhibited the highest number, as shown in Table V, with a total of 704 call gates and 2,534 internal call gates inserted. The number of domain-switching instructions directly impacts execution efficiency; this relationship is further analyzed in §VII-F. Additionally, key-related system calls were replaced to ensure secure handling; for instance, `ccrypt` required 12 system call replacements (primarily `malloc` and `free`), as it mainly utilizes the crypto heap for key storage. The compilation time overhead introduced is moderate, with the largest overhead observed in `wolfSSL`, which adds approximately one minute.

E. Q4: Evaluation on Benchmarks and Applications

Creation and Destruction Cost. We measured the basic costs incurred during the creation and destruction phases of an CCApp by repeatedly launching and terminating an empty application 10,000 times. On average, when the driver is loaded, CCAegis delegates the vector table, resulting in an overhead of about 1 ms. During the CCApp loading phase, CCAegis creates Crypto and Non-Crypto GPTs, as well as verifies page tables and program code (including dynamic libraries), adding an overhead of 16.99 ms. Correspondingly, during the destruction phase, CCAegis releases the delegated memory and destroys the established GPTs, incurring an additional overhead of 1.90 ms. Nevertheless, these are all one-time costs that do not affect the runtime performance. The primary runtime overhead arises from the following domain-switching operations.

Domain Switching Overhead. We measured the overhead of domain switching in CPU cycles using the Performance Monitoring Unit (PMU) over one million iterations. The average overhead was 1,211 cycles for `GPERM` and 1,269 cycles for `RPERM`. This overhead primarily arises from expensive TLB flushes associated with GPT switches and the context save-and-restore triggered by traps to EL3. Internal call gates within crypto functions exhibited similar overheads. Additionally, we measured the extra overhead introduced by traps when exiting and entering the CCApp for exception handling (e.g., system calls), which were 1,067 and 1,437 cycles, respectively.

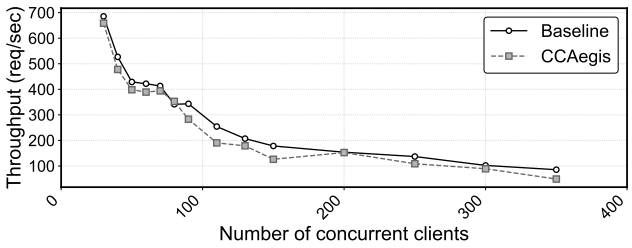

Fig. 10. Throughput cost on `Nginx` incurred by CCAegis.

`Nginx` Throughput. Following previous studies [51], [8], [19], we used the `Nginx v1.21.4` web server to measure the impact of CCAegis's protection on throughput. As our baseline, we employed a `Nginx` server compiled with an uninstrumented `wolfSSL v5.1.0` using the `ECDHE_RSA_WITH_AES_128_GCM_SHA256` cipher suite, without using the CCAegis loader. We conducted the evaluation using two machines connected via LAN: one configured as the `Nginx` server, and the other running `Apache Bench (ab) v2.3` to simulate a client sending 10,000 requests with a file size of 32KB. Since the overhead primarily arises during the connection establishment phase, we enabled the `keepalive` option to more accurately measure overhead during request processing, aligning closely with realistic usage scenarios. To compare the throughput overhead under high and low concurrency, we incrementally increased the number of concurrent clients up to the system's maximum capacity (350 concurrent clients, beyond which significant connection failures occurred). As illustrated in Figure 10, under low concurrency conditions ($\leq$80 clients), the overhead introduced by CCAegis remained below 10%, peaking at 9.33% with 40 concurrent clients. However, as concurrency increased further, the throughput overhead rose sharply, reaching a maximum of 42.72% at 350 concurrent clients.

`wolfSSH` Latency. To evaluate the overhead CCAegis introduces to SSH connections, we used a `wolfSSH v1.4.17` client (compiled with the instrumented `wolfSSL` cryptographic library) to establish 1,000 connections to a local `OpenSSH v8.2p1` server, comparing the results against an uninstrumented baseline. On average, CCAegis 's protection incurred a 25.35% overhead per connection request, corresponding to approximately 27.2 ms. Note that this overhead applies primarily during the initial connection phase; subsequent data transfers after the connection is established would exhibit significantly lower latency.

Cryptographic Libraries Performance. In addition to `wolfSSL`, we further evaluated the performance overhead on other libraries: `ccrypt v1.11`, `libhydrogen` (commit `7ff9582`), and `libxcrypt v4.4.36`. To measure

TABLE VI
COST OF CCAEGIS IN CCRYPT, LIBHYDROGEN, AND LIBXCRYPT.

	ccrypt encrypt	ccrypt decrypt	libhydrogen encrypt	libhydrogen decrypt	libxcrypt SHA
Baseline (s)	0.5552	0.5587	0.4361	0.2168	0.0488
CCAegis (s)	0.5835	0.5830	0.4388	0.2175	0.0492
Overhead	5.10%	4.35%	0.61%	0.31%	0.84%

TABLE VII
STRESS TEST OF WOLFSSL'S AES-GCM WITH A 2048-BYTE INPUT SIZE.

	AES128 encrypt	AES128 decrypt	AES192 encrypt	AES192 decrypt	AES256 encrypt	AES256 decrypt
Baseline (µs)	36.36	36.36	39.76	39.72	43.31	43.27
CCAegis (µs)	63.73	63.91	66.97	66.95	70.45	70.42
Overhead	75.28%	75.80%	68.44%	68.56%	62.68%	62.74%

overhead, We modified the libraries' built-in test suites. Specifically, `libhydrogen` employs the `Gimli` permutation for encryption, `ccrypt` utilizes the `Rijndael` cipher, and `libxcrypt` implements `HMAC-SHA1`. Each test was performed on 10MB of data, repeated 1,000 times, with results summarized in Table VI. Overall, CCAegis introduced low overhead across all three cryptographic libraries, with negligible performance impact (<1%) on `libhydrogen` and `libxcrypt`, and moderate overhead on `ccrypt`—5.10% for encryption and 4.35% for decryption.

Comparison with the State-of-the-Art. We compared CCAegis with state-of-the-art CCA-based systems, specifically SHELTER [8]. Since SHELTER was developed on the Juno R2 development board and its performance prototype is not open-sourced, we modified the functional prototype and ported it to the Rock Pi 4B development board, which we currently use. We evaluated the performance overhead on all the real-world applications previously tested. Compared to SHELTER's process-level isolation, CCAegis provides intra-process isolation. However, achieving this fine-grained isolation with CCAegis introduces additional overhead due to the insertion of call gates for isolation domain switching. Relative to SHELTER, CCAegis imposes less than 0.83% additional overhead for cryptographic libraries like `ccrypt`, 7.43% for `wolfSSH`, and the highest overhead for `Nginx` at 16.92%.

F. Q5: Root Cause Analysis

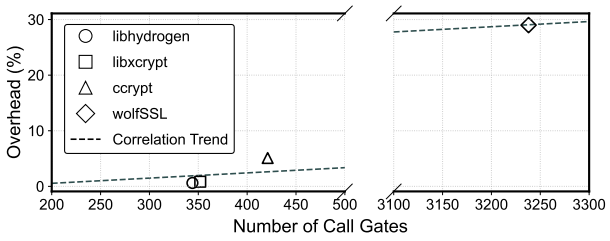

Fig. 11. Correlation between number of call gates and overhead.

Through comparative experiments, we observed that CCAegis's performance overhead is influenced by two key factors. First, as shown in Figure 11, the overhead increases with the number of call gates inserted to create isolated domains. For example, `wolfSSL`—with 3,238 call gates—experiences higher overhead (over 20% greater) compared to other cryptographic libraries with fewer than 500 instrumented call gates. Second, the invocation frequency of call gates also directly affects execution efficiency. Even libraries with similar levels of instrumentation exhibit higher overhead under high-pressure workloads. As illustrated in Figure 10, high-concurrency scenarios trigger more frequent domain switches, leading to a threefold increase in overhead compared to low-concurrency conditions. We further conducted a stress test on `wolfSSL`'s AES-GCM. As depicted in Table VII, this direct benchmarking of crypto functions revealed an even greater overhead (up to 75.80%) compared to the overhead observed in high-concurrency `Nginx` scenarios invoking crypto functions.

VIII. DISCUSSION

Multi-Domain Support. CCAegis currently supports only a single crypto domain, but it can be easily extended to support multiple independent domains. Separate GPTs can be created for function sets handling different types of secrets, ensuring that only the relevant functions have access to the memory.

Performance Optimization. Internal call gates secure untrusted calls within crypto functions, but frequent trapping to EL3 can be costly. To optimize efficiency without compromising security, we can perform security verification [34] for frequently called external functions instead of instrumentation. Another practical optimization is to combine GPT enforcement with lightweight intra-process guards such as Pointer Authentication Code (PAC). GPT remains the authoritative barrier that keeps the OS and other processes from reading protected pages, while PAC provides a user-mode fast path that reduces `svc+smc` transitions inside a sensitive region.

Scalability. CCAegis primarily targets cryptographic applications because they stress-test intra-process isolation: secrets are repeatedly derived, transformed, and propagated across many small functions and library boundaries, which enlarges the taint surface and increases potential domain switches. By contrast, many noncryptographic scenarios (e.g., an ML service reading an API token) load a secret once and use it sparingly, where simple static labeling is often sufficient. Keys and their derivatives traverse multiple call chains and buffers, which is exactly where our taint guided function partitioning and GPT based enforcement are exercised most, so the reported overheads serve as a conservative upper bound for many general-purpose workloads. The design can be extended to other security-sensitive applications by adding monitor support for system calls beyond I/O and key generation (§V-C).

False Negatives. While we report false positives in §VII-D, we did not emphasize false negatives because our analysis prioritizes soundness over completeness. We adopt a fail closed default: when precision is insufficient at a boundary, we conservatively mark it sensitive. For ground truth, we manually validated the false positives, but quantifying false negatives is difficult because real applications lack an oracle that labels all true secret flows. We do not claim to eliminate all false negatives, and complete soundness for complex programs is undecidable. Potential sources include implicit or indirect control transfers that obscure flows, assembly with branches

or jumps, and missing taint or erase annotations. As future work, we will ship default signatures for common cryptographic APIs and emit warnings on control flow redirections in assembly or indirect jumps.

IX. Related Work

Compartmentalization. Partitioning a program and isolating the secure parts is an effective way to enforce the principle of least privilege. To assist developers in program partitioning, SOAAP [52] provides an interactive tool based on data flow analysis. Wedge [53] offers dynamic tracing components to analyze runtime memory access behavior. However, these tools still require developers to manually identify clear partition boundaries. To resolve this, other works have been proposed to support automated partitioning. Privtrans [54] uses static analysis to separate privileged and non-privileged processes. ProgramCutter [55] uses dynamic data dependency analysis to separate intra-process privileged code. PtrSplit [56] supports general pointers by constructing a program dependence graph for each function. CCAegis focuses on safeguarding keys and their propagation, using taint analysis to automate program partitioning. Specifically, after partitioning, CCAegis enables privilege-aware instrumentation (e.g., replacing key-related system calls) in §IV-B.

Intra-Process Isolation. Intra-process isolation [51], [11] provides fine-grained protection for sensitive data. Some works utilize existing isolation primitives. For example, libmpk [10] implements a secure and scalable isolation framework using the Intel Memory Protection Keys (MPK). CALI [57] employs `nsjail` [58] to isolate libraries. Other works construct their own isolation primitives. For example, PANIC [59] utilizes Privileged Access Never (PAN) and Load Store Unit (LSU) to implement an access-control-based isolation mechanism. Capacity [60] employs PAC and Memory Tagging Extension (MTE) to create a capability-based framework. While these features effectively control memory access permissions within user space, they are not designed to protect against attacks from privileged software.

CCA Enhancement. Although the hardware for CCA has not yet been released, several manufacturers are updating their software stacks. Arm has released a lightweight hypervisor for the realm world, the Realm Management Monitor (RMM) [61]. Samsung has implemented Islet [62], a Rust-based RMM that supports confidential machine learning. Recently, some works have focused on verifying the TCB of CCA or applying CCA in various security contexts. The Verification Infrastructure for Armv9 (VIA) [63] conducts formal verification of the RMM's security, while the Trusted Firmware Explorer (TFX) [64] verifies the RMM and RME hardware implementation interface. Acai [27] and CAGE [28] enable CVMs in CCA to securely access accelerators. In addition to utilizing Realms, SHELTER [8] and RContainer [7] use GPTs to protect security-sensitive applications and containers in the normal world, respectively. However, none of these CCA-based systems provide function-level fine-grained isolation.

X. Conclusion

CCAegis is the first system to extend Arm CCA for fine-grained isolation. It decouples analysis and isolation, providing a generalized isolation capability for cryptographic programs. Utilizing static analysis, CCAegis automates the partitioning of program components that require protection and employs three types of GPTs to safeguard cryptographic keys. Experiments demonstrate that CCAegis has a more compact TCB, which effectively prevents key leakage from privileged and intra-process threats. Real-world applications such as `Nginx`, SSH connections, and popular cryptography libraries show only a modest overhead of 1.01× to 1.43× on our system.

References

[1] AMD, "Amd secure encrypted virtualization (SEV)," 2023, https://www.amd.com/en/developer/sev.html.

[2] Intel, "Intel trust domain extensions (intel TDX)," 2023, https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[3] Arm, "Confidential compute architecture," 2023, https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[4] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1963–1976.

[5] T. Alves, "Trustzone: Integrated hardware and software security," *Information Quarterly*, vol. 3, pp. 18–24, 2004.

[6] Arm, "Learn the architecture - Realm management extension," 2021, https://developer.arm.com/documentation/den0126/latest/.

[7] Q. Zhou, W. Cao, X. Jia, P. Liu, S. Zhang, J. Chen, S. Xu, and Z. Song, "Rcontainer: A secure container architecture through extending arm cca hardware primitives," in *NDSS*, 2025.

[8] Y. Zhang, Y. Hu, Z. Ning, F. Zhang, X. Luo, H. Huang, S. Yan, and Z. He, "Shelter: Extending arm cca with isolation in user space," in *32nd USENIX Security Symposium (USENIX Security'23)*, 2023.

[9] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 conference on internet measurement conference*, 2014, pp. 475–488.

[10] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 241–254.

[11] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1221–1238.

[12] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 56–71.

[13] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for arm," in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014, pp. 558–569.

[14] 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)*. IEEE, 2020, pp. 776–789.

[15] J. Gu, B. Zhu, M. Li, W. Li, Y. Xia, and H. Chen, "A hardware-software co-design for efficient intra-enclave isolation," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3129–3145.

[16] Intel, "Intel® software guard extensions (intel® SGX)," 2023, https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html.

[17] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 191–206.

[18] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza *et al.*, "Glamdring: Automatic application partitioning for intel SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 285–298.

[19] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1607–1619.

[20] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[21] 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)*, 2021, pp. 1073–1090.

[22] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 857–874.

[23] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable memory protection in the PENGLAI enclave," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 275–294.

[24] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves." in *NDSS*, 2019.

[25] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "Case: Cache-assisted secure execution on arm processors," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 72–90.

[26] M. H. Yun and L. Zhong, "Ginseng: Keeping secrets in registers when you distrust the operating system." in *NDSS*, 2019.

[27] S. Sridhara, A. Bertschi, B. Schlüter, M. Kuhne, F. Aliberti, and S. Shinde, "Acai: Protecting accelerator execution with arm confidential computing architecture," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 3423–3440.

[28] C. Wang, F. Zhang, Y. Deng, K. Leach, J. Cao, Z. Ning, S. Yan, and Z. He, "Cage: Complementing arm cca with gpu extensions," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium*, 2024.

[29] Arm, "Trusted firmware-a documentation," 2024, https://trustedfirmware-a.readthedocs.io/en/latest/.

[30] Keystone, "Eyrie enclave runtime kernel," 2023, https://github.com/keystone-enclave/keystone-runtime.

[31] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 552–561.

[32] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 253–264, 2013.

[33] R. Cui, L. Zhao, and D. Lie, "Emilia: Catching iago in legacy code." in *NDSS*, 2021.

[34] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A guided fuzzer to find buffer boundary violations," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 49–64.

[35] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, "An Off-Chip attack on hardware enclaves via the memory bus," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[36] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin, "Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 313–324.

[37] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[38] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[39] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[40] N. Grech and Y. Smaragdakis, "P/taint: Unified points-to and taint analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[41] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR.CHECKER: A soundy analysis for linux kernel drivers," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1007–1024.

[42] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith, "Toctou, traps, and trusted computing," in *International Conference on Trusted Computing*. Springer, 2008, pp. 14–32.

[43] S. Han, S.-J. Kim, W. Shin, B. J. Kim, and J.-C. Ryou, "Page-oriented programming: Subverting control-flow integrity of commodity operating system kernels with non-writable code pages," in *33nd USENIX Security Symposium (USENIX Security'24)*, 2024.

[44] S. Shinde, S. Wang, P. Yuan, A. Hobor, A. Roychoudhury, and P. Saxena, "BesFS: A POSIX filesystem for enclaves with a mechanized safety proof," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 523–540.

[45] Arm, "Fixed virtual platforms," 2023, https://developer.arm.com/downloads/-/arm-ecosystem-models.

[46] ——, "Guest arm cca linux branches," 2024, https://gitlab.arm.com/linux-arm/linux-cca/-/tree/cca-guest/v2.

[47] ——, "Reference implementation of arm-cca rmm specification," 2023, https://github.com/TF-RMM/tf-rmm/releases/tag/tf-rmm-v0.4.0.

[48] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 2011, pp. 289–298.

[49] P. M. Gharat, U. P. Khedker, and A. Mycroft, "Flow-and context-sensitive points-to analysis using generalized points-to graphs," in *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*. Springer, 2016, pp. 212–236.

[50] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[51] X. Jin, X. Xiao, S. Jia, W. Gao, D. Gu, H. Zhang, S. Ma, Z. Qian, and J. Li, "Annotating, tracking, and protecting cryptographic secrets with cryptompk," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 650–665.

[52] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson, "Clean application compartmentalization with soaap," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1016–1031.

[53] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: splitting applications into reduced-privilege compartments," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008, pp. 309–322.

[54] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, vol. 57, no. 72, 2004.

[55] Y. Wu, J. Sun, Y. Liu, and J. S. Dong, "Automatically partition software into least privilege components using dynamic data dependency analysis," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 323–333.

[56] S. Liu, G. Tan, and T. Jaeger, "Ptrsplit: Supporting general pointers in automatic program partitioning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2359–2371.

[57] M. Bauer and C. Rossow, "Cali: Compiler-assisted library isolation," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 550–564.

[58] Google, "nsjail: a light-weight process isolation tool, making use of linux namespaces and seccomp-bpf syscall filters," 2023, https://nsjail.dev/.

[59] J. Xu, M. Xie, C. Wu, Y. Zhang, Q. Li, X. Huang, Y. Lai, Y. Kang, W. Wang, Q. Wei *et al.*, "Panic: Pan-assisted intra-process memory isolation on ARM," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 919–933.

[60] K. Dinh Duy, K. Cho, T. Noh, and H. Lee, "Capacity: Cryptographically-enforced in-process capabilities for modern arm architectures," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 874–888.

[61] T. Firmware, "TF-RMM: An implementation of arm-cca rmm," 2023, https://github.com/TF-RMM/tf-rmm.

[62] Samsung, "Islet: An on-device confidential computing framework," 2023, https://github.com/islet-project/islet.

[63] 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.

[64] A. C. Fox, G. Stockwell, S. Xiong, H. Becker, D. P. Mulligan, G. Petri, and N. Chong, "A verification methodology for the arm® confidential computing architecture: From a secure specification to safe implementations," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 376–405, 2023.

Shiqi Liu received the B.S. degree in Cyber Science and Engineering from the University of International Relations, Beijing, China, in 2022, and the M.S. degree in Cyber Science and Engineering from Huazhong University of Science and Technology, Wuhan, China, in 2025. He is currently a Ph.D. student with the Center for Secure Information Systems at George Mason University, Fairfax, VA, USA. His research interests include system security and software security.

Zhouqi Jiang received the B.S. degree in Cyber Science and Engineering from Huazhong University of Science and Technology, Wuhan, China, in 2022, and the M.S. degree in Cyber Science and Engineering from the same university in 2025. He is currently pursuing the Ph.D. degree with the Institute of Software, Chinese Academy of Sciences (ISCAS), and the University of Chinese Academy of Sciences, Beijing, China. His research interests include secure firmware and system security.

Jie Wang received his Ph.D. degree from the University of Chinese Academy of Sciences (UCAS) in 2021. He was a visiting scholar at George Mason University and is currently an Associate Professor at the School of Cyber Science and Engineering, Huazhong University of Science and Technology. His research interests include system security, trusted computing, and trusted AI accelerators.

Wei Zhou received his Ph.D. degree from the University of Chinese Academy of Sciences (UCAS) in 2021. He was a visiting scholar at Pennsylvania State University and is currently an Associate Professor at the School of Cyber Science and Engineering, Huazhong University of Science and Technology. His research interests include system security, trusted computing, and IoT system security.

Kun Sun (Member, IEEE) received the Ph.D. degree in computer science from North Carolina State University. He has more than 15 years working experience in both industry and academia, and serves as the Director of the Sun Security Laboratory (SunLab) and the Associate Director of the Center for Secure Information Systems (CSIS). He has published more than 100 peer-reviewed conference and journal articles. His research focuses on systems and network security.

Zhaohui Chen received the Ph.D. degree in computer science and technology from the University of Chinese Academy of Sciences (UCAS), Beijing, China, in 2022. He is currently working with DAMO Academy, Alibaba Group, Hangzhou, China. His research interests include applied cryptography, hardware security, and homomorphic encryption.

Yulai Xie (Member, IEEE) received the B.E. and Ph.D. degrees in computer science from the Huazhong University of Science and Technology (HUST), China, in 2007 and 2013, respectively. He is currently a Professor with the School of Cyber Science and Engineering in HUST, China. He was a Visiting Scholar with the University of California, Santa Cruz, CA, USA, in 2010, and a Visiting Scholar with the Chinese University of Hong Kong, Hong Kong, in 2015. His research interests mainly include cloud storage and virtualization, digital provenance, intrusion detection, machine learning, and computer architecture.