

A Code-Refactoring-Based Migration Scheme for Trusted Applications across Heterogeneous TEEs

Di Lu¹, Qingwen Zhang¹, Yujia Liu¹, Xiangkun Yang², Li Yang¹, and Yulong Shen¹

¹School of Computer Science and Technology, Xidian University, Xi'an 710126, China

²Huawei Technologies Co., Ltd., Shenzhen 518129, China

Trusted execution environments (TEEs) have become a key building block for privacy-preserving and confidential computing because they protect sensitive code and data through hardware-assisted isolation. However, the current TEE ecosystem remains highly fragmented. Different platforms expose different execution abstractions, programming interfaces, build procedures, and deployment models, making trusted applications difficult to migrate and costly to adapt. This paper presents CRTAMS, a code-refactoring-based migration scheme for trusted applications across heterogeneous TEEs. CRTAMS introduces an integrated programming model that uses Clang Attributes to express TEE boundary annotations, named CATBA, so that developers can describe trusted and untrusted code boundaries in a platform-neutral form. Based on the annotated source code, CRTAMS converts the program into LLVM intermediate representation, extracts annotation metadata, refactors the code according to the selected target platform, and automatically generates a target-platform-compliant trusted-application project structure and auxiliary build files. We implement a prototype and evaluate it on three categories of heterogeneous TEE hardware, including ARM TrustZone, Intel SGX/TDX, and AMD SEV. Experimental results show that CRTAMS enables trusted-application migration across heterogeneous TEEs with low refactoring cost and moderate execution overhead in the evaluated workloads, while reducing the amount of platform-specific code that developers must write manually.

Index Terms—Trusted execution environment, LLVM IR, control-flow graph, application migration, code refactoring.

I. INTRODUCTION

TRUSTED execution environments (TEEs) [1] provide protected execution regions through coordinated hardware and software mechanisms. Code and data running inside a TEE are isolated from the rich execution environment (REE), which makes it difficult for an external adversary to read or tamper with trusted-application state at runtime. This isolation property has made TEEs an important technical foundation for confidential-computing, cloud-computing, and edge-computing scenarios that process sensitive data. Nevertheless, the practical deployment of TEEs is still constrained by the lack of a unified development and migration model.

The first obstacle is cross-platform migration among heterogeneous TEEs. Mainstream processor vendors have introduced their own TEE mechanisms, such as ARM TrustZone [2], Intel SGX [3] and TDX [4], and AMD SEV [5]. According to their execution granularity, these platforms can be broadly divided into process-level TEEs, represented by TrustZone and SGX, and virtual-machine-level TEEs, represented by TDX and SEV. Because these platforms differ substantially in architecture, runtime behavior, and application development workflows, trusted applications are often tightly coupled to a specific TEE backend and cannot be migrated directly to another platform.

The second obstacle appears even within the same broad TEE family. For example, besides standard ARM TrustZone-based systems, vendors provide their own secure operating systems and security frameworks, such as Huawei iTrustee, Qualcomm QSEE, and Samsung KNOX. Although these systems are all built around TrustZone-based isolation, differences in

hardware configuration, secure OS design, available services, and build processes still create compatibility barriers. As a result, a trusted application developed for one device is often difficult to reuse on another TrustZone-based device without non-trivial engineering adaptation.

The third obstacle is the complexity of trusted-application development. Process-level TEEs usually require a dual-component TA/CA programming model that separates a trusted application (TA) from a client application (CA). In TrustZone, for example, the CA runs in the REE and invokes the TA through platform-specific communication APIs, while the TA executes in the secure world. Fig. 1 gives a typical TrustZone code example. Developers must explicitly initialize communication contexts, open and close sessions, configure shared memory, invoke platform-specific call interfaces, and use TEE-specific data structures. These low-level details are necessary for a native TEE application but impose a high learning and maintenance cost when the same application is expected to run across heterogeneous TEEs.

These issues restrict the broader adoption of TEE technology. On the one hand, architectural differences among chip vendors hinder trusted-application migration across heterogeneous platforms. On the other hand, the platform-specific programming model increases development time and makes TEE adoption difficult for developers who are not familiar with each platform's native APIs. Therefore, a practical question is how to support source-level trusted-application migration across heterogeneous TEEs while keeping each target platform's native build and deployment path.

To address this problem, this paper proposes CRTAMS (Code-Refactoring-based Trusted Application Migration Solution). CRTAMS starts from the compiler layer rather than from a new in-TEE runtime. Developers write integrated

<pre> 1 TEEC_Context ctx; 2 TEEC_Session sess; 3 TEEC_Operation op; 4 TEEC_UUID uuid = TA_HELLO_WORLD_UUID; 5 uint32_t err_origin; 6 7 TEEC_InitializeContext(NULL, &ctx); 8 TEEC_OpenSession(&ctx, &sess,...); 9 10 op.paramTypes = TEEC_PARAM_TYPES(...); 11 12 TEEC_InvokeCommand(&sess,...); 13 14 TEEC_CloseSession(&sess); 15 TEEC_FinalizeContext(&ctx); </pre> (a) CA code	<pre> 1 TEE_Result TA_CreateEntryPoint() ^a ` 2 void TA_DestroyEntryPoint() ^a ` 3 TEE_Result TA_OpenSessionEntryPoint() ^a ` 4 void TA_CloseSessionEntryPoint() ^a ` 5 6 TEE_Result inc_value() ^a ` 7 8 TEE_Result TA_InvokeCommandEntryPoint() { 9 switch (cmd_id) { 10 case TA_HELLO_WORLD_CMD_INC_VALUE: 11 return inc_value(param_types, params); 12 } </pre> (b) TA code
--	--

Fig. 1. TrustZone code example.

application code with lightweight annotations. CRTAMS then automatically refactors the code and generates target-platform project files and build configurations so that the application can be compiled by the native toolchain of the target TEE. This design retains the native deployment path of each platform while reducing the amount of platform-specific code that developers must handle manually. The main contributions are as follows.

- We design an integrated programming model for process-level TEEs. The model uses Clang Attributes-based TEE Boundary Annotations (CATBA) to mark TEE-side and REE-side code boundaries, automatically separates TA and CA code, and hides much of the conventional dual-component TA/CA programming model from developers.
- We design and implement an LLVM-IR-based code-refactoring mechanism. The mechanism converts integrated application code into an intermediate representation, extracts CATBA metadata, reconstructs trusted-application code for the selected TEE platform, and generates the auxiliary build and project files needed for deployment.
- We implement CRTAMS on heterogeneous TEE hardware, including ARM TrustZone on Raspberry Pi 3B and STM32MP157C-DK2, Intel SGX/TDX on a Supermicro X13SEI-F server, and AMD SEV on a Supermicro H11DSi server. Functional and performance evaluations show the feasibility of the proposed approach and identify the main sources of performance overhead.

II. RELATED WORK

Existing work on trusted-application migration and development simplification can be grouped into three technical lines: building an in-TEE runtime, constructing confidential containers, and performing compiler-level code refactoring.

A. In-TEE Runtime Systems

One approach is to embed a general runtime layer inside the TEE so that existing application binaries or minimally modified applications can run in a protected environment. This approach reduces the amount of application-level changes but usually introduces an additional runtime into the trusted computing base (TCB).

For SGX, Graphene-SGX [6] is an open-source library OS [7] that runs unmodified Linux binaries inside SGX enclaves. It supports application execution through a limited set of ECALL interfaces and provides services for system calls inside the enclave. Based on Graphene-SGX, Occlum [8] further improves memory safety and multi-process support and supports modern programming languages such as Rust. These systems simplify SGX development, but they are primarily tied to Intel SGX and do not provide cross-architecture portability. Unlike these SGX-oriented runtime systems, CRTAMS does not aim to execute unmodified binaries inside a library OS or runtime layer. Instead, it refactors annotated source code and generates native trusted-application projects for different TEE backends.

For TrustZone, HiddenApp [9] and TrustShadow [10] build lightweight runtime environments and system-call proxy layers so that native Linux applications can execute with TrustZone-based protection. Tarnhelm [11] allows developers to designate confidential code regions by source-code annotations and constructs a lightweight TEE-side runtime for confidential execution. UIEE [12] further proposes a user-space isolated execution environment for embedded TEE systems, enabling Linux applications to enter a TEE with limited TCB growth.

WebAssembly (WASM) has also been used as a portable execution format for TEEs. Twine [13] builds a trusted runtime for WASM inside SGX based on the WebAssembly Micro Runtime (WAMR). Applications written in C, C++, or Rust can be compiled into WASI-compatible WASM binaries and executed inside the enclave. Follow-up work extends Twine to support file-system and network functions [14]. WATZ [15] brings a WASM runtime to TrustZone and supports remote attestation, allowing WASM code to run in the TEE without following the native TrustZone dual-component TA/CA programming model.

Although these systems reduce the effort of moving applications into a specific TEE, most of them are optimized for a single TEE platform or a narrow class of platforms. Moreover, placing a general runtime inside the TEE expands the TCB and may enlarge the attack surface. Recent confidential-computing architectures, such as Arm CCA [16], introduce new execution abstractions such as Realms, but they focus on low-level confidential execution rather than on unified trusted-application migration across multiple heterogeneous TEE platforms.

UniTEE [17] and SecGear [18], [19] move closer to unified TEE development. UniTEE proposes a unified enclave abstraction and supports Intel SGX, AMD SEV, and ARM TrustZone. SecGear provides a confidential-computing development kit for heterogeneous platforms, including SGX, iTrustee, and RISC-V Penglai [20]. These systems are important steps toward unified development, but their portability is still mediated by framework-level interfaces or platform-specific adapters. In contrast, CRTAMS focuses on compiler-level source refactoring and native project generation, so that the generated applications can still follow the target platform's original build and deployment path.

B. Confidential Containers

Confidential containers provide a cloud-native deployment abstraction for applications protected by TEE hardware. The Confidential Containers project [21] integrates confidential computing with container orchestration and uses TEE-based memory protection to defend container workloads against unauthorized access and tampering. However, this line of work is mainly designed for cloud-native environments and is difficult to apply directly to resource-constrained embedded devices. Alibaba Cloud has also provided an SGX-based cloud-native secure container solution [22], which combines container management flexibility with VM-like isolation. Its hardware foundation, however, is still platform-specific and therefore does not directly solve the migration problem across diverse TEE platforms.

C. Compiler-Level Code Refactoring

Compiler-level refactoring transforms normal application code into TEE-compatible code automatically, thereby reducing manual adaptation. RT-Trust [23] is an early representative in this direction. It first targets TrustZone and is later extended to different TEEs including SGX [24]. It combines program analysis, shared-memory configuration, and function redirection to transform C programs into trusted applications. Related studies extend this idea to more languages and more complex program forms. Cadote [25] assists Rust enclave development through compiler support. Montsalvat [26] partitions Java/GraalVM Native Image applications into trusted and untrusted components by annotations. Privagic [27] performs automatic code partitioning for multi-threaded C/C++ applications using explicit secure typing. Compared with RT-Trust, CRTAMS extends automated refactoring from trusted-execution adaptation to an end-to-end migration workflow that includes boundary annotation extraction, LLVM-IR-based code reconstruction, and native project generation for both process-level and VM-level TEEs.

Compared with in-TEE runtimes and confidential containers, compiler-level refactoring has three advantages. First, it can be applied to resource-constrained embedded devices because it does not require a large general runtime inside the TEE. Second, it is friendly to native TEE platforms because it uses the target platform's existing toolchain and avoids expanding the TCB with a new runtime layer. Third, it has relatively low deployment complexity because the output is target-platform-compliant code and project files. However, existing refactoring approaches still tend to focus on a specific platform or a limited set of platform combinations. CRTAMS follows the compiler-level route but aims to support trusted-application migration across process-level and VM-level heterogeneous TEEs under a unified programming and generation workflow. Overall, CRTAMS differs from prior compiler-level refactoring work by treating migration as a complete source-to-native-project process rather than only as code partitioning or TEE-call adaptation.

III. BACKGROUND

This section introduces the background needed to understand the design of CRTAMS, including TEEs, the LLVM

compilation framework, and graph-based program representation.

A. Trusted Execution Environments

A TEE is a hardware-assisted secure execution architecture that provides confidentiality, integrity, and runtime protection for sensitive code and data. According to the execution granularity, TEEs can be divided into process-level and VM-level TEEs. Process-level TEEs, such as ARM TrustZone and Intel SGX, create protected execution contexts for trusted applications and destroy these contexts after the applications finish. VM-level TEEs, such as Intel TDX and AMD SEV, create protected virtual machines in which trusted applications run. The VM lifecycle is independent of a single application inside the VM.

TrustZone. ARM TrustZone is a hardware-partitioning-based isolation architecture widely used in mobile and embedded systems. It divides the processor into a normal world and a secure world. A Non-Secure flag in the system bus identifies the current security state and prevents normal-world components from interfering with secure-world operations. A secure operating system is required inside the secure world; OP-TEE [28] is one of the most mature open-source implementations. The OP-TEE architecture includes user-space client interfaces, a kernel driver for hardware interaction, and the TEE core that provides secure OS services.

SGX. Intel SGX is a user-space hardware extension that creates isolated enclaves for sensitive code and data. Enclave memory is protected by hardware encryption and access control, so even a privileged operating system cannot directly inspect or modify enclave contents. When an enclave is loaded, the processor measures its code and data, and the measurement can be used to verify enclave integrity.

TDX. Intel TDX is a confidential-virtualization technology. It relies on TDX-capable processors and a TDX module to manage the full lifecycle of trusted domains (TDs). TDX extends VMX with Secure-Arbitration Mode (SEAM), which allows multiple TDs to run concurrently while isolating them from one another and from the hypervisor.

SEV. AMD SEV is a confidential-virtualization mechanism built on Secure Memory Encryption (SME) [29]. It integrates a hardware encryption engine into the memory controller and encrypts data before writing it to memory. The AMD Secure Processor manages per-VM encryption keys and isolates the memory of different VMs, thereby protecting the confidentiality of VM memory.

B. LLVM Compilation Framework

LLVM [30] is a modular compiler infrastructure consisting of front ends, optimizers, and back ends. A front end converts high-level source code into LLVM intermediate representation (IR). The optimizer performs language-independent IR optimization and target-independent hardware optimization. A back end then lowers optimized IR to target machine code. The unified IR design allows developers to insert customized passes at different stages of the compilation pipeline without rewriting a complete compiler.

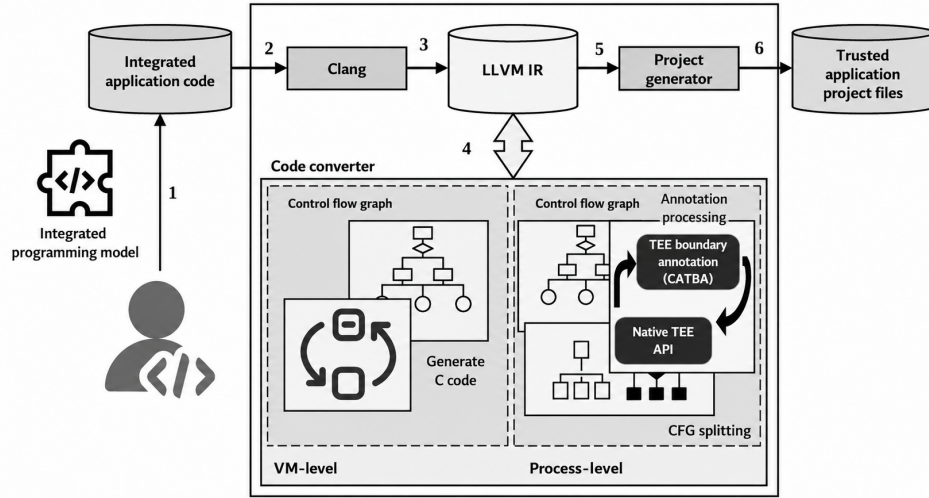


Fig. 2. Overall architecture of CRTAMS.

LLVM IR [31] is an assembly-like, architecture-independent intermediate language that retains enough high-level structure for program analysis and transformation. Clang is the C/C++ front end of LLVM. It also provides Clang Attributes [32], a language-level annotation mechanism that allows developers to attach metadata to source code. In CRTAMS, this mechanism is used to represent TEE boundary annotations. These annotations are preserved in the generated LLVM IR and then consumed by the code converter.

C. Graph-Based Program Representation

A control-flow graph (CFG) [33] is a graph-theoretic representation of program execution flow. A node represents a basic block, and an edge represents a possible control transfer between basic blocks. A basic block is a consecutive sequence of instructions with a single entry and a single exit.

A program-dependence graph (PDG) [34] combines data-dependence and control-dependence information in a unified representation. Data dependence describes how values are transferred among operations, while control dependence describes how the execution of an operation depends on a branch condition. Because PDGs are language-independent and abstract, they are widely used in compiler optimization, static analysis, and program slicing.

IV. SYSTEM DESIGN

This section presents the architecture and execution workflow of CRTAMS, describes the integrated programming model, introduces the core components, and discusses semantic consistency under the modeled transformation.

A. Overall Architecture

Fig. 2 shows the overall architecture of CRTAMS. The system consists of an integrated programming model and three main components: Clang, the code converter (CODC), and the project generator (PROG). Developers write integrated

application code using the programming model. The code is processed by Clang, CODC, and PROG in sequence, and the output is a target-platform-compliant trusted-application project. The generated project can then be compiled by the native toolchain of the selected TEE platform.

The integrated programming model is designed mainly for process-level TEEs, such as TrustZone and SGX. Its purpose is to hide the dual-component TA/CA programming model that requires developers to write TA and CA code separately. By using CATBA annotations, developers explicitly mark code boundaries while leaving platform-specific communication generation to CRTAMS. For VM-level TEEs such as TDX and SEV, the same integrated source code can be transformed into ordinary C code after annotation removal and then placed into the VM-level project structure.

The three core components work as follows.

- **Clang** translates the integrated application source code into platform-independent LLVM IR. During this stage, Clang performs syntax and semantic checks and emits IR that retains CATBA annotation information.
- **CODC** takes LLVM IR as input and performs platform-specific code refactoring. For process-level TEEs, it separates TA and CA modules according to CATBA, maps LLVM IR data types and operations to portable C constructs, and generates the platform-specific communication interfaces. For VM-level TEEs, it removes CATBA metadata and converts LLVM IR into C code that can be compiled inside a protected VM.
- **PROG** generates auxiliary project files using templates. For process-level TEEs, it creates build files and platform-specific project structures for TA and CA compilation. For VM-level TEEs, it creates scripts that encapsulate code compilation, image construction, and VM instantiation.

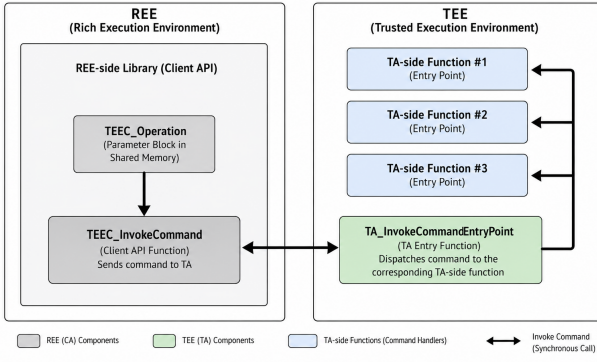


Fig. 3. TA-CA communication on TrustZone.

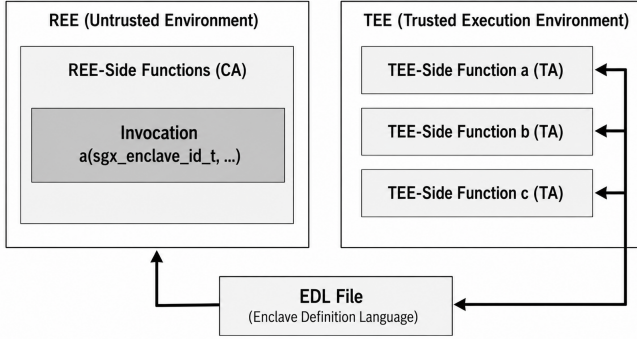


Fig. 4. TA-CA communication on SGX.

B. Integrated Programming Model

Native process-level TEE applications require platform-specific APIs for communication between the TEE and the REE. CRTAMS introduces an integrated programming model that uses lightweight annotations and simplified interfaces to abstract these differences.

In TrustZone, communication depends on the TA-side `TA_InvokeCommandEntryPoint` function, the CA-side `TEEC_InvokeCommand` function, and the `TEEC_Operation` structure. As shown in Fig. 3, the CA specifies the target TA function by a command identifier and passes parameters through shared memory. `TEEC_Operation` declares shared-memory buffers, while `TEEC_InvokeCommand` sends a command identifier and operation object to the TA. The TA entry function dispatches the request to the target function and returns results through shared memory.

In SGX, bidirectional calls between the TEE and the REE are declared in an EDL file. Developers must explicitly specify each function and the memory size needed for parameter passing. As shown in Fig. 4, when the REE invokes a trusted function, the generated bridge code routes the call into the enclave using an enclave identifier. Unlike TrustZone, SGX also supports calls from trusted code back to untrusted functions.

To abstract these common communication patterns, CRTAMS defines CATBA annotations based on `__attribute__((__annotate__(...)))`. The annotation information is preserved in LLVM IR but does

TABLE I
CATBA GRAMMAR

Line	Attribute
1	<code>__attribute__((attribute-list))</code>
2	<code>attribute-list = __annotate__("CATBA"), ...</code>

TABLE II
CATBA TYPES

Line	CATBA item
1	<code>tee(type)</code>
2	<code>type = TZ SGX</code>
3	<code>tafunc</code>
4	no parameter
5	<code>memsize(size1, size2, ...)</code>
6	<code>size = n</code> , where <code>n</code> is the byte length
7	<code>cafunc</code>
8	no parameter

not affect normal program execution. Table I gives the basic grammar, and Table II lists the supported annotation types.

The `tee(type)` annotation marks a function that should be generated for execution inside the TEE. Unmarked functions are treated as CA-side code by default. The `tafunc` annotation marks TEE-side functions that can be invoked by the CA. A function marked only with `tee` is treated as an internal TEE-side function. The `memsize` annotation configures the shared-memory sizes needed for communication parameters and is used together with `tafunc`. The `cafunc` annotation supports SGX-style callbacks from trusted code to untrusted code and is also used with `memsize`. Context creation, session establishment, and session teardown are inserted automatically by static analysis, so developers do not need additional annotations for these repetitive operations.

Besides communication APIs, TEE-side functional APIs are also platform-specific. CRTAMS therefore provides simplified TA APIs through two strategies. The first strategy keeps all parameters when the native TEE API contains only general parameters. For example, a random-number generation API with only a buffer and a length can be exposed directly. The second strategy hides platform-specific parameters or aggregates several native calls into one simplified interface. For example, an encryption initialization function may require a platform-specific operation handle, which is omitted in the simplified API. A key-setting operation may require several native calls such as attribute initialization and transient object population; CRTAMS combines them into a single `SetKey` interface. Fig. 5 illustrates these mapping relationships.

C. Code Converter Design

CODC transforms LLVM IR into target-platform-standard code. It follows different processing flows for process-level and VM-level TEEs.

For process-level TEEs, the workflow is shown in Fig. 6. CODC first initializes LLVM target support and registers itself as a valid code-generation backend. It then configures the passes needed for optimization and code emission. After

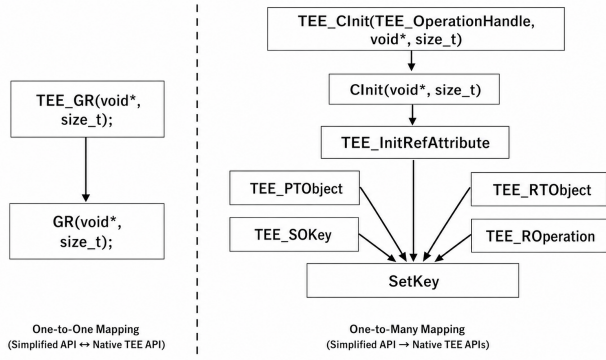


Fig. 5. Mapping between simplified TA APIs and native TEE APIs.

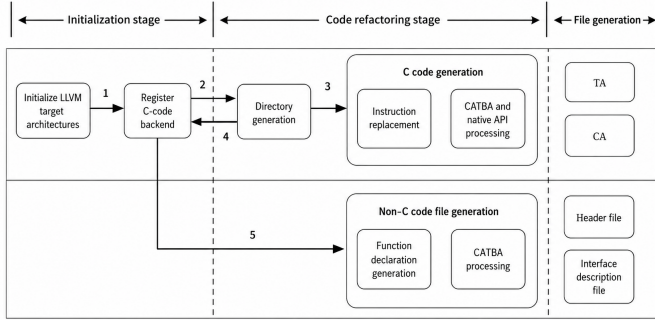


Fig. 6. CODC processing flow for process-level TEEs.

parsing command-line parameters and determining the target TEE platform, it enters the refactoring stage.

The generation process has two sub-stages. In the C-code generation stage, CODC constructs the target directory structure, converts LLVM IR instructions into semantically equivalent C code, extracts CATBA annotations, and replaces annotated calls with platform-specific communication interfaces and TEE APIs. The output is compilable TA and CA C source code. In the auxiliary-file generation stage, CODC generates function declarations and auxiliary files. Function declarations must be regenerated because function signatures may change and inlined helper functions may be inserted during IR-to-C conversion. For SGX, CODC additionally generates the corresponding EDL file according to CATBA metadata.

For VM-level TEEs, the processing flow is shown in Fig. 7. The initialization stage is the same as that of process-level TEEs. The main difference is that CODC does not split code into TA and CA modules or replace communication calls. Instead, it removes all CATBA annotation information and converts LLVM IR into C code that can be compiled and executed inside a protected VM.

If instruction conversion fails or CATBA parsing is abnormal, CODC terminates the current file-generation process and reports the error. This prevents incomplete or inconsistent trusted-application code from being emitted silently.

D. Inter-Function Calling Rules

When developers write process-level TEE code, functions are divided into TEE-side and REE-side components. Trust-

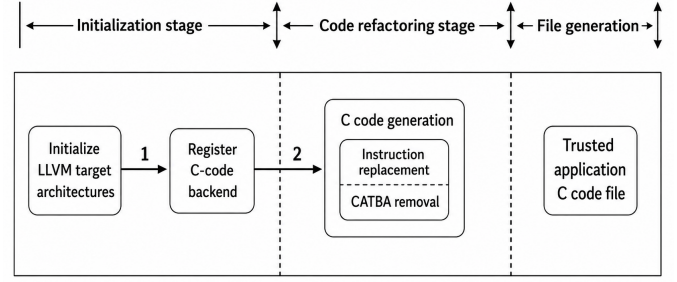


Fig. 7. CODC processing flow for VM-level TEEs.

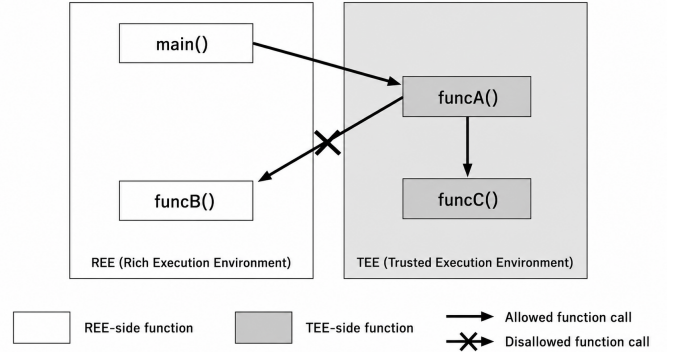


Fig. 8. Calling rules for process-level TEE applications.

Zone imposes a strict calling direction: a TEE-side function cannot directly call an REE-side function. This restriction prevents secure-world code from depending on non-secure-world code and helps maintain the security boundary.

Fig. 8 illustrates this rule. `funcA` and `funcC` are marked by tee and belong to the TEE side, whereas `main` and `funcB` belong to the REE side. If `funcA` calls both `funcB` and `funcC`, the call to `funcB` violates the TrustZone calling rule. To detect this problem, CODC performs static checking on LLVM IR. Each TEE-side function is treated as a root node. CODC recursively traverses all functions called from this root and checks whether they also belong to the TEE side. If an REE-side function is found in the call chain, CODC reports a warning and stops the traversal so that the developer can correct the function relationship.

E. Project Generator Design

PROG generates auxiliary files needed for final deployment. TEE project files are often highly reusable: for the same platform, only a small number of variables change between applications. PROG therefore uses template-based generation.

As shown in Fig. 9, the template generator classifies existing project files according to predefined keywords and creates template files. The templates are organized by TEE platform and maintained by PROG. During generation, developers provide keywords and target-platform information. PROG selects the corresponding templates through a platform adapter and fills the templates with application-specific values.

Fig. 10 shows the template-based generation process. With a unified set of keywords, PROG automatically selects tem-

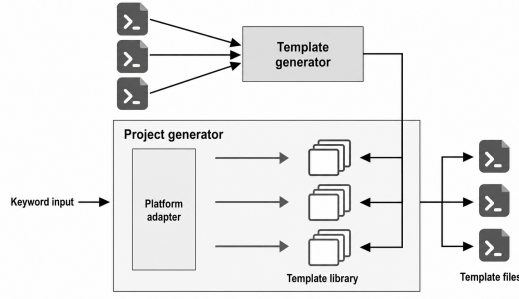


Fig. 9. Architecture of the project generator.

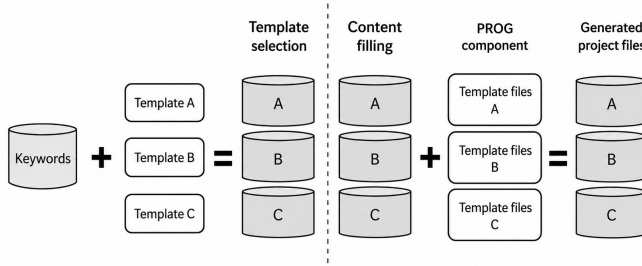


Fig. 10. Template use in project generation.

plates, fills required fields, and produces project files that conform to the target platform. The files generated by PROG are then combined with the trusted-application code generated by CODC, resulting in a complete project that can be compiled by the native toolchain of the target TEE.

F. Semantic Consistency

CRTAMS transforms integrated application code into target-platform-compliant trusted-application projects. During this transformation, code splitting and call replacement should preserve the behavior relevant to the original program. We discuss this property through a PDG-based semantic-consistency argument. The argument assumes deterministic C code without undefined behavior, concurrent interactions, or unsupported platform side effects, and assumes that the generated communication wrappers preserve the input-output behavior of the replaced calls.

Definition 1 (Strong equivalence): Two programs P and Q are strongly equivalent if and only if, for every initial state σ , either both programs diverge from σ , or both terminate and produce the same final values for all variables. Divergence includes non-termination and abnormal termination, such as division by zero or out-of-bounds array access.

Definition 2 (Graph isomorphism): Following the standard notion of graph isomorphism [35], two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if and only if there exists a bijection $f : V_1 \rightarrow V_2$ such that, for any $u, v \in V_1$, $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$.

Theorem 1: If two programs have isomorphic PDGs under a semantics-preserving node mapping, then the two programs are strongly equivalent.

Proof: Let P denote the program before splitting. CODC separates P into a slice M containing the main function and a slice N containing TEE-side subfunctions. After rewriting, they become M' and N' , and the combined refactored program is denoted by Q . The PDG of a program X is written as $G_X = (V_X, E_X)$, where V_X contains assignment and control statements and E_X contains control and data dependences.

For the TEE-side slice, N contains the subfunctions that must execute inside the TEE. CODC adds declarations and dispatching statements needed by the platform, producing N' . The internal execution logic of each TEE-side function is not changed. Therefore, for the same input variables, the corresponding functions in N and N' produce the same outputs, and N and N' are strongly equivalent.

For the main slice, CODC replaces calls to `tafunc`-marked functions with target-platform invocation statements, producing M' . This replacement changes the call mechanism but not the called function's input-output semantics. Since N' provides the same functional behavior as N , a call node in M that invokes a function in N can be mapped to the corresponding platform-call node in M' that invokes the matching function in N' . The remaining nodes of M are unchanged. Thus, there exists a bijection between V_{G_M} and $V_{G_{M'}}$, and the data and control dependence edges are preserved. Hence

$$G_M \cong G_{M'}. \quad (1)$$

Because N and N' are strongly equivalent and the call replacement preserves input-output behavior, the PDG of the original program and that of the refactored program are isomorphic:

$$G_P \cong G_Q. \quad (2)$$

By Theorem 1, P and Q are strongly equivalent under the modeled node mapping. Therefore, within the stated assumptions, the refactored code preserves the execution behavior relevant to the original program. ■

V. IMPLEMENTATION

This section gives implementation details of CRTAMS, including annotation extraction, TA/CA splitting, and native API generation.

A. Extracting CATBA Annotations

CATBA is implemented using `__annotate__`. The mapping between annotations and functions is stored in the LLVM IR global variable `llvm.global.annotations`. As shown in Fig. 11, this variable is a `ConstantArray`. Each element is a `ConstantStruct` containing four fields: a pointer to the annotated function, a pointer to the annotation string, the source-file name, and the source line number. CODC extracts CATBA information from this structure.

The extraction process initializes four data structures. `TafuncVec` stores functions marked by `tafunc`; `CafuncVec` stores functions marked by `cafunc`; `TEEFunSet` records functions marked by `tee`; and `MemMap` maps function names to the integer arrays declared by `memsize`. CODC obtains `llvm.global.annotations` through `getNamedGlobal` and traverses each

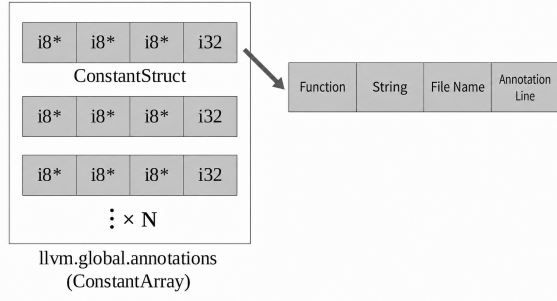
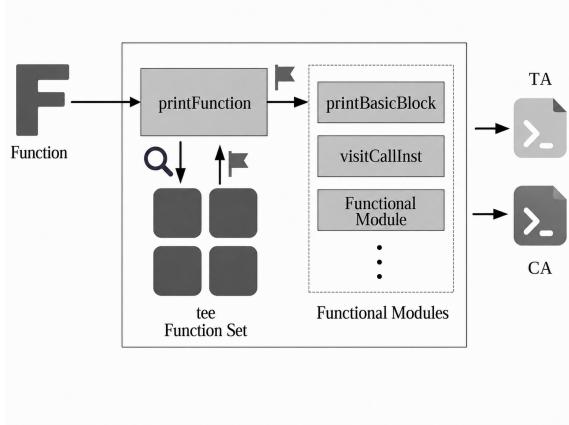
Fig. 11. Structure of `llvm.global.annotations`.

Fig. 12. Splitting TA and CA code.

`ConstantStruct`. For each element, it extracts the function pointer and annotation string. If the annotation is `tafunc`, `cafunc`, or `tee`, the function is inserted into the corresponding vector or set. If the annotation is `memsize`, CODC parses its parameters using regular expressions and updates `MemMap`. The result is a structured mapping between annotations and functions, which is then used by the code-generation stage.

B. Splitting TA and CA Code

CODC integrates several modules for converting LLVM IR into C code. For example, `printBasicBlock` converts basic blocks, and `visitCallInst` converts call instructions. CODC traverses LLVM IR by functions and invokes these modules through `printFunction`.

As shown in Fig. 12, CODC maintains two output streams: one for TA code and one for CA code. After extracting CATBA information and initializing `TEEFunSet`, CODC checks whether the current function belongs to `TEEFunSet` before converting it. If the function is a TEE-side function, the generated C code is written to the TA stream; otherwise, it is written to the CA stream. A global flag is used as a stream selector so that all code fragments generated by submodules are emitted to the correct target stream.

C. Generating Native APIs

Native communication APIs differ significantly across TEE platforms. CODC therefore generates platform-specific communication code from CATBA metadata.

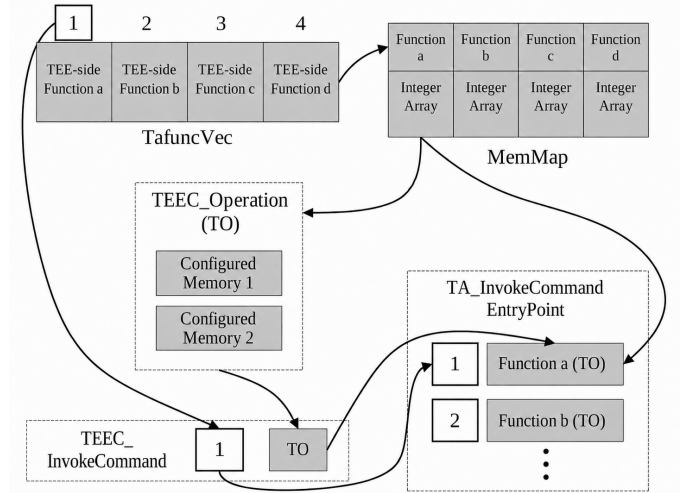


Fig. 13. TrustZone communication API generation.

For TrustZone, the generation process is shown in Fig. 13. On the TA side, CODC traverses `TafuncVec`, obtains all TEE-side functions callable by the CA, and extracts the parameter-size arrays from `MemMap`. Each function is assigned an index in `TafuncVec`, and this index is used as a command identifier. CODC then generates `TA_InvokeCommandEntryPoint`, which dispatches a received command to the corresponding target function.

On the CA side, CODC identifies call instructions that invoke TEE-side functions. It then obtains the function index and memory-size configuration from `TafuncVec` and `MemMap`, generates the required `TEEC_Operation` configuration, and emits a `TEEC_InvokeCommand` call. Because the parameters of `TA_InvokeCommandEntryPoint` correspond directly to the shared-memory buffers configured in `TEEC_Operation`, generating the correct number and size of memory arguments is sufficient to transmit parameters between the CA and TA.

For SGX, code generated from `tafunc` and `memsize` is mainly written into `Enclave.edl`. The trusted section declares TEE-side functions callable by the CA and uses the `size` attribute to specify communication-memory sizes. The untrusted section declares CA functions that can be called from the TEE. CODC first emits the fixed EDL header, then traverses `TafuncVec` to generate trusted declarations, and finally traverses `CafuncVec` to generate untrusted declarations. In the CA code, when CODC finds a call to a function in `TafuncVec`, it inserts the enclave identifier `global_eid` as the first argument so that the generated SGX bridge code can route the call correctly.

For TEE-side functional APIs, CODC maintains a predefined vector `EfuncVec` that contains the names of simplified APIs. During TA-side code generation, it checks whether each call target appears in `EfuncVec`. If it does, CODC enters the corresponding `switch-case` branch, completes platform-specific parameters, replaces the simplified function name with the native API name, and emits the required native calls. For one-to-many mappings, the branch expands a single simplified call into a sequence of native TEE API calls.

VI. EVALUATION

A. Experimental Environment

The evaluation uses two classes of hardware: embedded boards and general-purpose servers. The embedded platforms are Raspberry Pi 3B and STM32MP157C-DK2, both of which support TrustZone and run Linux in the non-secure world with OP-TEE in the secure world. They are used to verify trusted-application migration across heterogeneous TrustZone devices. The server platforms include a Supermicro X13SEI-F server equipped with an Intel Xeon Platinum 8558 processor, which supports SGX and TDX, and a Supermicro H11DSi server equipped with an AMD EPYC 7401 processor, which supports SEV. The CRTAMS prototype is implemented based on Clang, llvm-cbe, and Gengen, with extensions for the four evaluated TEE platforms.

The generated projects are built through the native build and deployment flow of each target platform. On TrustZone, the generated projects follow the OP-TEE client and TA build flow. On SGX, the generated enclave and client projects follow the Intel SGX SDK toolchain. On TDX and SEV, the generated C projects are compiled inside the protected VM environment using the platform's standard Linux compiler configuration. The software components used in our experiments include Clang 17.0.6, llvm-cbe 1.0, Gengen 1.4.2, OpenSTLinux Distribution (Yocto) 5.15, raspbian-tee 1.0, gcc-arm-linux-gnueabi 8.2, linux-sgx 2.25, QEMU 8.2.2, GCC 13.3.0, and virt-customize 1.52.0.

B. Code Size Comparison

This experiment compares the amount of code manually written by developers with the amount of standard API and project code automatically generated by CRTAMS. The benchmark applications are selected from `llvm-test-suite` and `optee_examples`. The former contains recursive programs and cryptographic algorithms, while the latter contains OP-TEE examples that demonstrate TEE-side functional APIs. The experiment is divided into TrustZone and SGX cases. VM-level TEEs are not included in this code-size comparison because they do not require native process-level communication API generation.

For TrustZone, five programs are selected from `llvm-test-suite`, and five are selected from `optee_examples`. Each program is rewritten using the integrated programming model and then refactored by CRTAMS. Table III reports the lines of integrated trusted-application code, generated standard API code, and generated project files. The generated project-file code is 162 lines for both Raspberry Pi 3B and STM32MP157C-DK2.

For the first five `llvm-test-suite` programs, most generated standard API code comes from CATBA-based communication conversion, so the generated API size is relatively stable. For `crc32`, `pc1`, and `rc4`, the total generated API and project code is already larger than the manually written integrated code, with an average ratio of about 1.4. For `des` and `md5`, the generated code accounts for 37% and 81% of the manual code, respectively. For the five `optee_examples`

TABLE III
TRUSTZONE CODE-LINE STATISTICS

Application	Integrated code	Generated API	Project files
<code>crc32</code>	148	63	162
<code>des</code>	605	67	162
<code>md5</code>	279	64	162
<code>pc1</code>	165	61	162
<code>rc4</code>	156	65	162
<code>acipher</code>	146	133	162
<code>hello-world</code>	20	61	162
<code>random</code>	35	67	162
<code>hotp</code>	195	131	162
<code>secure</code>	240	352	162

TABLE IV
SGX CODE-LINE STATISTICS

Application	Integrated code	Generated API	Project files
<code>almbench</code>	366	35	300
<code>crc32</code>	137	33	300
<code>des</code>	588	34	300
<code>md5</code>	279	34	300
<code>pc1</code>	170	34	300
<code>rc4</code>	140	35	300
<code>matrix</code>	69	34	300
<code>n-body</code>	140	34	300
<code>recursive</code>	54	34	300
<code>nsieve</code>	36	34	300
<code>spectral</code>	53	34	300

programs, the generated code includes both CATBA conversion and simplified TA API expansion. In these programs, the generated code exceeds the integrated code in all cases and reaches an average of about 2.4 times the manually written code. The `hello-world` case is especially clear: the generated code is about eleven times the integrated code.

For SGX, eleven `llvm-test-suite` programs are used. As shown in Table IV, the generated project-file code is fixed at 300 lines, and the generated standard API code averages 34 lines. The manually written application code averages 184 lines. Overall, CRTAMS generates about twice as many lines as the developer-written code. This shows that CRTAMS can offload a large amount of repetitive platform-specific work, especially for simple applications whose functional code is short but whose native TEE project scaffolding is still required.

C. Component Performance

This experiment measures the execution time of the three main components during application refactoring. Table V reports the TrustZone results. Because Raspberry Pi 3B and STM32MP157C-DK2 use different PROG templates, their PROG times are recorded separately. Across the ten applications, the average Clang time is 37 ms, the average CODC time is 28 ms, the average PROG time is 3.7 ms for Raspberry Pi 3B, and 3.8 ms for STM32MP157C-DK2. The average total time for one application is 68.9 ms. Even the most complex application, `des`, takes only about 134 ms in total.

Table VI reports SGX and VM-level TEE results. TDX and SEV share the same Clang and CODC processing path, so their component times are reported together. For SGX, the average

TABLE V
TRUSTZONE COMPONENT EXECUTION TIME (MS)

Component	crc32	des	md5	pc1	rc4	acipher	hello-world	random	hotp	secure
Clang	52	54	43	36	34	33	29	30	29	30
CODC	27	76	42.2	23	22.7	17.8	16	17	22	18.5
PROG-raspberry	4	3.9	4.3	3.8	3.3	3.4	3.2	3.8	3.5	4
PROG-stm32	4.1	4	3.9	3.91	3.4	3.6	3.3	4	3.6	4.2

TABLE VI
SGX AND VM-LEVEL TEE COMPONENT EXECUTION TIME (MS)

Component	a-bench	crc32	des	md5	pc1	rc4	matrix	n-body	recursive	nsieve	spectral
SGX-Clang	45	52	53.5	43	36	35	33	43	29	34	41
SGX-CODC	29	26	78	42	24	22	22	23	21	18	20
SGX-PROG	4	3	3.3	3.2	3.3	3.6	3.2	3.7	3.5	3.2	3.7
VM-Clang	36	59	45	48	34	39	38.9	49	27	39	35
VM-CODC	28	26	52	34.5	23	20.9	21	23	20	20.8	22
TDX-PROG	3	3	3.5	3.1	3.3	3.2	3.2	3.8	3.7	3.2	3
SEV-PROG	3	4	3.1	3.4	3.4	3.6	3.3	3.1	3.2	3	3.4

Clang, CODC, and PROG times are 40.4 ms, 29.5 ms, and 3.4 ms, respectively; the average total time is 73.4 ms. For VM-level TEEs, the average Clang, CODC, and PROG times are 40.9 ms, 26.5 ms, and 3.2 ms, respectively; the average total refactoring time is 70.6 ms, and the maximum time for *des* is 100.5 ms.

Comparing the four platforms, process-level TEEs require about 2.8 ms more refactoring time than VM-level TEEs on average, mainly because process-level refactoring must perform code splitting and communication-interface generation. In all four platforms, the CODC time of *des* exceeds the Clang time, because *des* has the largest trusted-code size and therefore requires more IR-to-C conversion work. Overall, component time is positively correlated with trusted-application code size. Even in the slowest case, however, the automatic refactoring time remains at the millisecond level and is negligible compared with manual TEE adaptation.

D. Trusted-Application Execution Performance

This experiment evaluates the execution overhead introduced by code refactoring. The refactored applications are compared with native applications on the corresponding TEE platforms. In this section, a refactored application is called a refactored TA, and the native version is called a native TA.

Fig. 14 and Fig. 15 show the TrustZone results on Raspberry Pi 3B and STM32MP157C-DK2. The average execution time of the refactored TAs is about 1.1 times that of the native TAs. The largest performance difference appears in the *des* application on Raspberry Pi 3B, where the refactored execution time reaches 1.8 times the native version. The main reason is that LLVM IR uses static single assignment (SSA) form, and IR-to-C conversion may introduce redundant assignments that add runtime overhead.

Except for *des*, the slowdown of the other nine applications is within 1.3 times. The *acipher* application even runs faster after refactoring. This is because *acipher* comes from *optee_examples*, and its core computation is mainly performed through TEE-side functional APIs. Refactoring

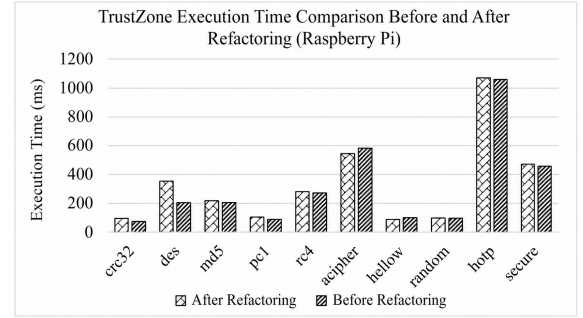


Fig. 14. Execution time before and after TrustZone refactoring on Raspberry Pi 3B.

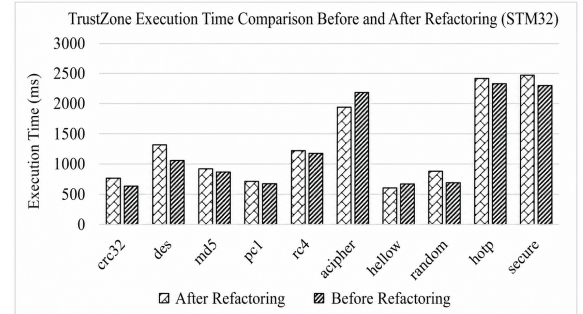


Fig. 15. Execution time before and after TrustZone refactoring on STM32MP157C-DK2.

does not add substantial extra computation, and the generated *goto*-based structure can slightly improve execution performance. Comparing the two boards, absolute execution times differ, but the before/after ratios for the same application are consistent. This indicates that CRTAMS does not introduce additional device-specific behavior in these cases and can support migration across the evaluated TrustZone devices.

Fig. 16 and Fig. 17 show the SGX results. Because the eleven applications differ greatly in execution time, they are divided into two groups. On average, the refactored TAs take about 2.81 times the native execution time. For *nsieve*, *spectral*, *matrix*, and two other applications, the refactored execution time is more than three times the native

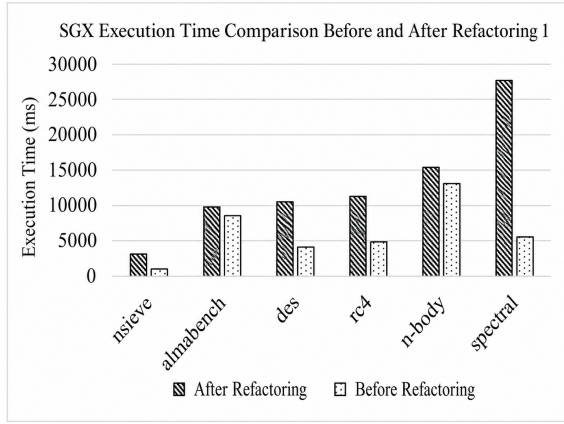


Fig. 16. SGX execution-time comparison before and after refactoring, group 1.

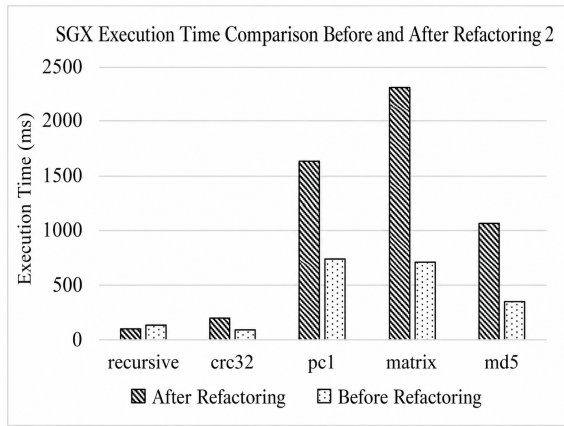


Fig. 17. SGX execution-time comparison before and after refactoring, group 2.

execution time. The reason is that LLVM IR conversion may generate redundant code and code expansion, which reduces the efficiency of the generated C code.

The *recursive* application performs better after refactoring. Its original version contains many recursive calls, which cause stack overhead and repeated computation. The refactored version converts the recursive logic into an iterative structure based on `goto` statements, reducing stack overhead and improving performance. Therefore, although CRTAMS can introduce extra overhead in some compute-intensive programs, structural changes during refactoring may also improve performance in specific cases. The SGX overhead is mainly caused by redundant assignments and control-transfer structures introduced during SSA-based LLVM IR to C reconstruction, together with wrapper code needed to preserve the native enclave-call path. This overhead is the cost of avoiding an additional in-enclave runtime while still producing native SGX projects automatically. It is acceptable for migration, testing, and non-real-time workloads where portability and reduced manual adaptation are more important than peak execution speed. For CPU-bound workloads with strict latency constraints, the generated code should be further optimized or manually tuned before deployment.

We also compare CRTAMS with Occlum, which represents the in-TEE runtime approach for SGX. Ten applications are

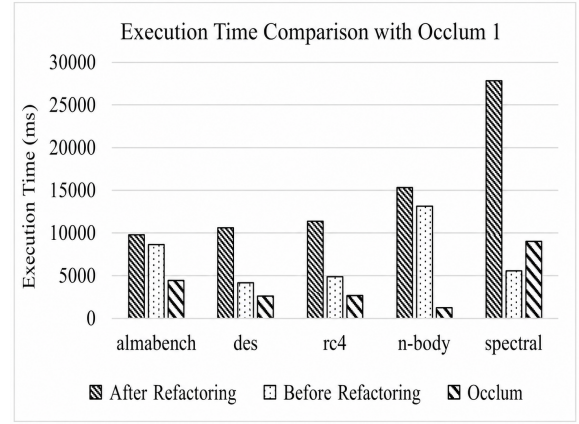


Fig. 18. Execution-time comparison with Occlum, group 1.

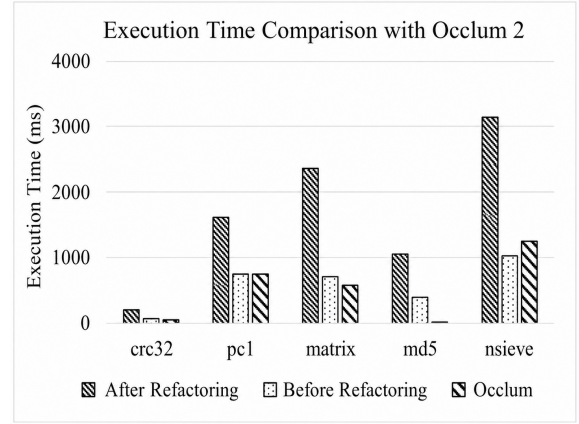


Fig. 19. Execution-time comparison with Occlum, group 2.

executed in the Occlum environment. As shown in Fig. 18 and Fig. 19, Occlum reduces average execution time by about 29% compared with native applications. Seven applications show clear performance improvement, *pc1* is close to the native version, and two applications are slightly slower. Occlum places the whole application inside the TEE, avoiding frequent TEE/REE context switches and communication overhead; therefore, applications such as *des* and *almbench* benefit from the runtime. For *spectral* and *nsieve*, which are computation-intensive and depend heavily on CPU instructions, native SGX remains faster.

Compared with refactored CRTAMS applications, Occlum has a clear performance advantage: its average execution time is only 26.4% of that of the refactored applications. This is expected because Occlum places the whole application in a library-OS environment inside SGX and can avoid part of the TA/CA-style communication path. CRTAMS makes a different design choice: it keeps the native project-generation and deployment model of each target platform and does not introduce a general runtime into the TEE. Therefore, the additional execution cost is a necessary trade-off for cross-platform source migration, smaller runtime assumptions, and native build compatibility.

Finally, Fig. 20 and Fig. 21 show the VM-level TEE results on TDX and SEV. On TDX, the average execution time of refactored TAs is 2.87 times that of native TAs. The slowdown of *des*, *md5*, *nsieve*, and *spectral* exceeds the average.

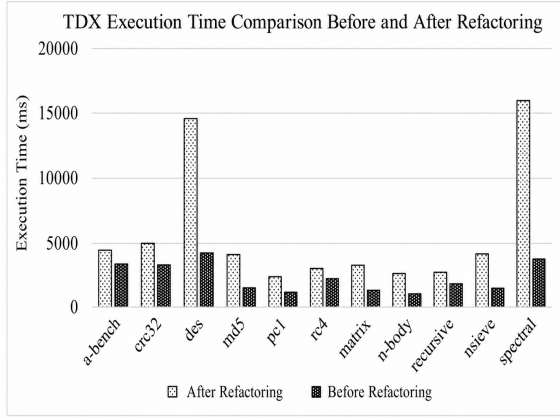


Fig. 20. TDX execution-time comparison before and after refactoring.

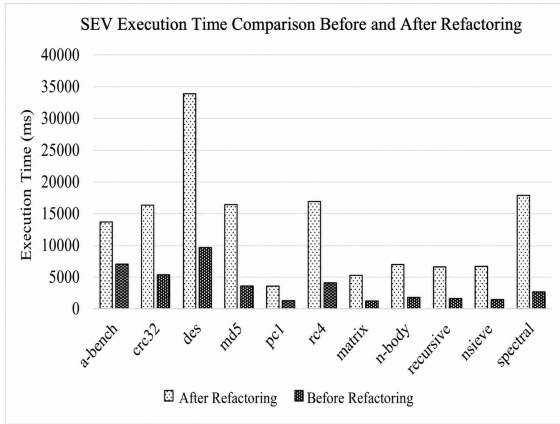


Fig. 21. SEV execution-time comparison before and after refactoring.

On SEV, the average execution time of refactored TAs is 4.16 times the native version. The slowdown of `md5`, `rc4`, `recursive`, `nsieve`, and `spectral` exceeds the average. The most affected application is `spectral`, whose refactored execution time is 4.35 times the native version on TDX and 7.03 times on SEV. This overhead is mainly caused by extra operations introduced during SSA-based IR conversion.

The `rc4` application shows much higher refactored execution time on SEV than on TDX. This difference is related to the architectural characteristics of the two platforms. SEV relies on memory encryption, and frequent memory accesses incur higher runtime cost, whereas TDX focuses on VM isolation and has relatively better memory-access efficiency in this experiment. For VM-level TEEs, the additional cost also includes ordinary C code reconstructed from LLVM IR and the auxiliary I/O and project-control logic used to run the generated program inside a protected VM. These costs do not affect whether the migrated applications can be generated, built, and executed successfully, but they do affect throughput-oriented workloads. Overall, CRTAMS is most suitable when the primary goal is reducing migration effort and preserving native deployment paths. The results should therefore be interpreted as demonstrating practical usability for migration-oriented and non-real-time workloads, rather than as showing that the current backend is already optimal for performance-critical VM-level applications.

VII. DISCUSSION AND LIMITATIONS

The current prototype is intended for C-based trusted applications whose TEE boundary can be expressed at the function level. It focuses on source-level refactoring and native project generation rather than on binary compatibility or transparent execution of arbitrary unmodified programs. Therefore, programs with complex pointer aliasing, dynamic loading, intensive system-call behavior, callbacks across the TEE boundary, or multi-threaded interactions may require additional analysis or manual restructuring before they can be processed safely. Programs that rely on unsupported platform side effects, inline assembly, device-specific system calls, or implicit sharing through global state also fall outside the fully automated path of the current prototype.

The threat model of CRTAMS follows the threat model of the underlying TEE. We assume that hardware-enforced isolation, secure boot, memory protection, and remote-attestation mechanisms are provided by the selected TEE platform and its native runtime. CRTAMS does not attempt to replace these platform mechanisms. Its goal is to preserve the intended trusted/untrusted code boundary during source-level migration and to avoid introducing additional application-level trusted code beyond the generated wrappers and project files. Consequently, confidentiality and integrity of TEE-resident code and data remain the responsibility of the underlying platform, while developers remain responsible for selecting correct boundary annotations and validating application-specific security policies.

For process-level TEEs, CRTAMS currently follows the native TA/CA communication model of the target platform; it does not attempt to replace platform-level access-control or attestation mechanisms. These boundaries are consistent with the goal of reducing migration and project-generation effort while retaining each platform's native toolchain and runtime model.

VIII. CONCLUSION

This paper has presented CRTAMS, a code-refactoring-based migration scheme for trusted applications across heterogeneous TEEs. CRTAMS uses an integrated programming model to unify the development style of heterogeneous TEE platforms and reduce the burden of trusted-application development. Based on LLVM IR, it refactors annotated source code into target-platform-compliant trusted-application code and automatically generates auxiliary project files. The prototype supports ARM TrustZone, Intel SGX/TDX, and AMD SEV. Experimental results show that CRTAMS can support the migration of the evaluated trusted applications across heterogeneous TEE platforms with millisecond-level refactoring cost. The runtime overhead is acceptable for migration-oriented and non-real-time workloads, although compute-intensive SGX, TDX, and SEV workloads may still require further backend optimization.

Future work will proceed in three directions. First, we will optimize IR-to-C conversion to remove redundant statements in generated code and improve runtime efficiency. Second, we will extend the LLVM-IR-based workflow to support more

programming languages. Third, we will add support for open-source TEE platforms such as Keystone to further broaden the applicability of CRTAMS.

ACKNOWLEDGMENT

The authors would like to thank the editor-in-chief, associate editor, and reviewers for their valuable comments and suggestions. This research was supported by the National Natural Science Foundation of China (62232013, U24A20243, 62572377, 62302363), the Innovation Capability Support Program of Shaanxi (No. 2023-CX-TD-02), the Xidian University Specially Funded Project for Interdisciplinary Exploration (No. TZJHF202502) and the Fundamental Research Funds for the Central Universities (No. ZDRC2202).

REFERENCES

- [1] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *Proc. IEEE Trust-com/BigDataSE/ISPA*, 2015, pp. 57–64.
- [2] S. Pinto and N. Santos, "Demystifying ARM TrustZone: A comprehensive survey," *ACM Computing Surveys*, vol. 51, no. 6, pp. 1–36, 2019.
- [3] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, report 2016/086, 2016.
- [4] P. C. Cheng, W. Ozga, E. Valdez, *et al.*, "Intel TDX demystified: A top-down approach," *ACM Computing Surveys*, vol. 56, no. 9, pp. 1–33, 2024.
- [5] AMD, "AMD SEV-SNP: Strengthening VM isolation with integrity protection and more," White Paper, Jan. 2020.
- [6] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *Proc. USENIX ATC*, 2017, pp. 645–658.
- [7] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An operating system architecture for application-level resource management," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 251–266, 1995.
- [8] Y. Shen, H. Tian, Y. Chen, *et al.*, "Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX," in *Proc. ASPLOS*, 2020, pp. 955–970.
- [9] V. Velciu, F. Stancu, and M. Chiroiu, "HiddenApp—securing Linux applications using ARM TrustZone," in *Proc. International Conference on Security for Information Technology and Communications*, 2018, pp. 41–52.
- [10] L. Guan, P. Liu, X. Xing, *et al.*, "TrustShadow: Secure execution of unmodified applications with ARM TrustZone," in *Proc. MobiSys*, 2017, pp. 488–501.
- [11] D. Quarta, M. Ianni, A. Machiry, *et al.*, "Tarnhelm: Isolated, transparent and confidential execution of arbitrary code in ARM's TrustZone," in *Proc. CheckMATE*, 2021, pp. 43–57.
- [12] H. Yan, Z. Ling, X. Chen, *et al.*, "UIEE: Secure and efficient user-space isolated execution environment for embedded TEE systems," in *Proc. NDSS*, 2026.
- [13] J. Menetrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An embedded trusted runtime for WebAssembly," in *Proc. IEEE ICDE*, 2021, pp. 205–216.
- [14] J. Menetrey, M. Pasin, P. Felber, *et al.*, "A comprehensive trusted runtime for WebAssembly with Intel SGX," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 3562–3579, 2023.
- [15] J. Menetrey, M. Pasin, P. Felber, *et al.*, "WATZ: A trusted WebAssembly runtime environment with remote attestation for TrustZone," in *Proc. IEEE ICDSCS*, 2022, pp. 1177–1189.
- [16] X. Li, X. Li, C. Dall, *et al.*, "Design and verification of the Arm Confidential Compute Architecture," in *Proc. USENIX OSDI*, 2022, pp. 465–484.
- [17] J. Y. Gu, H. Li, Y. B. Xia, *et al.*, "Unified enclave abstraction and secure enclave migration on heterogeneous security architectures," *Journal of Computer Science and Technology*, vol. 37, no. 2, pp. 468–486, 2022.
- [18] openEuler, "secGear," 2025. [Online]. Available: <https://www.openeuler.org/en/other/projects/secgear>
- [19] openEuler, "openEuler," 2025. [Online]. Available: <https://www.openeuler.org/en>
- [20] E. Feng, X. Lu, D. Du, *et al.*, "Scalable memory protection in the Penglai enclave," in *Proc. USENIX OSDI*, 2021, pp. 275–294.
- [21] Confidential Containers, "Confidential Containers," 2025. [Online]. Available: <https://confidentialcontainers.org>
- [22] Alibaba Cloud, "Cloud-native secure container solution," 2025. [Online]. Available: <https://www.aliyun.com/solution/cloudnative/securecontainer>
- [23] Y. Liu, K. An, and E. Tilevich, "RT-Trust: Automated refactoring for trusted execution under real-time constraints," in *Proc. GPCE*, 2018, pp. 175–187.
- [24] Y. Liu, K. An, and E. Tilevich, "RT-Trust: Automated refactoring for different trusted execution environments under real-time constraints," *Journal of Computer Languages*, vol. 56, p. 100939, 2020.
- [25] F. Dreissig, J. Roeckl, and T. Mueller, "Compiler-aided development of trusted enclaves with Rust," in *Proc. ARES*, 2022, pp. 1–10, doi: 10.1145/3538969.3538972.
- [26] P. Yuhala, J. Menetrey, P. Felber, *et al.*, "Montsalvat: Intel SGX shielding for GraalVM native images," in *Proc. Middleware*, 2021, pp. 352–364, doi: 10.1145/3464298.3493406.
- [27] S. Tanigassalame, Y. Pipereau, A. Chader, J. Toljaga, and G. Thomas, "Privagic: Automatic code partitioning with explicit secure typing," in *Proc. Middleware*, 2024, pp. 199–210, doi: 10.1145/3652892.3700759.
- [28] OP-TEE, "Open Portable Trusted Execution Environment," 2025. [Online]. Available: <https://www.op-tee.org>
- [29] S. Mofrad, F. Zhang, S. Lu, *et al.*, "A comparison study of Intel SGX and AMD memory encryption technology," in *Proc. HASP*, 2018, pp. 1–8.
- [30] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proc. CGO*, 2004, pp. 75–86.
- [31] LLVM, "LLVM Language Reference Manual," 2025. [Online]. Available: <https://llvm.org/docs/LangRef.html>
- [32] Clang, "Attribute Reference," 2025. [Online]. Available: <https://clang.llvm.org/docs/AttributeReference.html>
- [33] F. E. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, 1970.
- [34] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.
- [35] R. C. Read and D. G. Corneil, "The graph isomorphism disease," *Journal of Graph Theory*, vol. 1, no. 4, pp. 339–363, 1977.



Di Lu (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and technology from Xidian University, China, in 2006, 2009, and 2014. He is currently a full professor in the School of Computer Science and Technology at Xidian University. His research interests include trusted computing, confidential computing, system and network security.



Qingwen Zhang received the BS degree from Xidian University, China, in 2024. She is currently pursuing an MS degree in the School of Computer Science and Technology at Xidian University, China. Her research interests include trusted computing and embedded system security.



Yujia Liu received the BS degree from Xi'an Shiyao University, China, in 2025. She is currently pursuing an MS degree in the School of Computer Science and Technology at Xidian University, China. Her research interests include trusted computing and embedded system security.



Xiangkun Yang received the MS degree from Xidian University, China, in 2025. He now works at Huawei Technologies Co., Ltd. His research interests include trusted execution environments.



Li Yang (Member, IEEE) received the B.S. degree in Instructional Technology from Shaanxi Normal University in 1999, and M.S. degree in computer science from Xidian University in 2005, and Ph.D. degree in cryptography from Xidian University, Xi'an, China in 2010. He was a visiting postdoctoral researcher in the Complex Networks & Security Research (CNSR) Lab at Virginia Tech from 2013 to 2014. He is currently a professor in School of Computer Science and Technology, Xidian University. His research interests include applied cryptography, wireless network security, cloud computing security, and trusted computing. He is the member of IEEE.



Yulong Shen (Member, IEEE) received the BS and MS degrees in computer science and a PhD degree in cryptography from Xidian University, Xi'an, China, in 2002, 2005, and 2008, respectively. He is currently a professor with the School of Computer Science and Technology, Xidian University. His research interests include wireless network security and cloud computing security.