RT-Trust: Automated Refactoring for Different Trusted Execution Environments under Real-Time Constraints

Yin Liu[a,1], Kijin An[a,1], Eli Tilevich[a,1,*]

[a] *2202 Kraft Drive, Blacksburg, VA 24060*

Abstract

Real-time systems must meet strict timeliness requirements. These systems also often need to protect their critical program information (CPI) from adversarial interference and intellectual property theft. Trusted execution environments (TEE) execute CPI tasks on a special-purpose processor, thus providing hardware protection. However, adapting a system written to execute in environments without TEE requires partitioning the code into untrusted and trusted parts. This process involves complex manual program transformations that are not only laborious and intellectually tiresome, but also hard to validate and verify adherence to real-time constraints. To address these problems, this paper presents novel program analyses and transformation techniques, accessible to the developer via a declarative meta-programming model. The developer declaratively specifies the CPI portion of the system. A custom static analysis checks CPI specifications for validity, while probe-based profiling helps identify whether the transformed system would continue to meet the original real-time constraints, with a feedback loop suggesting how to modify the code, so its CPI can be isolated. Finally, an automated refactoring isolates the CPI portion for TEE-based execution, communicated with through generated calls to the TEE API. The reference implementation of our approach profiles and transforms real-time systems to isolate their CPI functions to execute on two different TEE

[*] Corresponding author

Email addresses: `yinliu@cs.vt.edu` (Yin Liu), `ankijin@cs.vt.edu` (Kijin An), `tilevich@cs.vt.edu` (Eli Tilevich)

[1] Software Innovations Lab, Virginia Tech

platforms: OP-TEE and SGX. Although these platforms substantially differ in terms of their respective APIs and performance characteristics, our refactoring completely hides these differences from the developer by automatically synthesizing the correct CPI functionality required for these dissimilar TEE implementations. We have evaluated our approach by successfully enabling the trusted execution of the CPI portions of several microbenchmarks and a drone autopilot. Our approach shows the promise of declarative meta-programming in reducing the programmer effort required to adapt systems for trusted execution under real-time constraints.

Keywords: trusted execution, real-time systems, declarative meta-programming, software refactoring, program analyses

---

1. Introduction

The execution of mission-critical real-time systems must comply with real-time constraints. Many such systems also contain vulnerable critical program information (CPI) (i.e., sensitive algorithms and data) that must be protected. Failing to satisfy either of these requirements can lead to catastrophic consequences. Consider using an autonomous delivery drone to transport packages, containing food, water, medicine, or vaccines, to remote and hard-to-reach locations. Emergency personnel and professional nature explorers often depend on drone delivery when dealing with various crises. The drone's navigation component has real-time constraints; if it fails to compute the instructions for the autopilot to adjust the flight's directions or airspeed in a timely fashion, the drone may become unable to adjust its trajectory properly and deviate from the programmed delivery route. Since the cargo often must be delivered under strict time requirements, deviating from the shortest route can cause the entire delivery mission to fail. In addition, the software controlling module (e.g., navigation) constitutes critical program information (CPI). If an ill-intentioned entity takes control over the module's execution, the entire drone can be misrouted, causing the delivery to fail. Irrespective of the causes, the consequences of a failed delivery can be potentially life-threatening.

The vulnerabilities above can be mitigated by isolating CPI functions in a secure execution environment that would also control their interactions with the outside world. As a way to realize this idea, hardware manufacturers have started providing trusted execution environments (TEEs), special-purpose processors that can be used to execute CPI-dependent functionality. TEE can reliably isolate trusted code (i.e., in the secure world) from regular code (i.e., in the normal world); the secure world comes with its own trusted hardware, storage, and operating system. A special communication API is the only avenue for interacting with TEE-based code. With the TEEs being hard to compromise, isolating CPI in the secure world effectively counteracts adversarial attacks and prevents intellectual property theft. However, to benefit from trusted execution, systems must be designed and implemented to use different implementations of the TEE (e.g., OP-TEE [1], SGX [2]). Adapting existing real-time systems to use the TEE requires non-trivial, error-prone program transformations, while the transformed system's execution must continue to adhere to the original real-time constraints.

In particular, a developer transforming a system to take advantage of the newly introduced TEE module requires undertaking the following tasks: 1) isolate CPI-dependent code; 2) redirect invocations of CPI functions to TEE communication API calls; 3) verify that the transformed system continues to meet the original real-time constraints. Notice that all of these tasks are hard to perform correctly by hand.

To complete task 1), a developer not only needs to correctly extract the CPI-dependent code from the system, but also correctly identify all the dependencies; due to the potential complexity of these dependencies, some CPI-dependent code cannot be isolated in TEEs. Most importantly, different TEEs (e.g., OP-TEE and SGX) expose dissimilar APIs and conventions for isolating CPI functions. A CPI-dependent function can be isolated in both TEE implementations, only one of them, or neither of them. To determine how a CPI function can be isolated, developers must be intimately familiar with both the original source code and the requirements of each TEE implementation. As is often the case, developers performing adaptive maintenance are often not the ones who wrote the original system. To facilitate this difficult and error-prone process, prior work has proposed automatic program partitioning, even in the presence of pointer-based function parameters [3]. However, this prior work leaves out the issues of verifying whether a given partitioning strategy is valid or whether the partitioned system would comply with the real-time constraints.

To complete task 2), the developer must write by hand the communication logic required for the normal and secure worlds to talk to each other, correctly applying suitable TEE APIs that establish customized communication channels. However, to accomplish this task correctly, developers must invest a great deal of time and effort to learn and master both the OP-TEE or SGX implementations: the OP-TEE provides more than 130 APIs and about 40 data types [4, 5, 6], while SGX provides an Enclave Definition Language (EDL) with more than ten syntactic categories [7].

To complete task 3), the developer must be willing to develop additional test cases that can verify whether the transformed system satisfies the original real-time constraints. Existing approaches take advantage of profiling tools, including Pin tool [8] and gperftools [9], which require that profiling probes be added by hand.

To facilitate the process of adapting real-time systems to protect their CPI-dependent code using a TEE, this article presents RT-Trust, a program analysis and transformation toolset that supports developers in partitioning C-language systems in the presence of real-time constraints. The developer can either specify the TEE implementation (i.e., OP-TEE or SGX) as a compiler option, or rely on RT-Trust to automatically determine the available implementation by inspecting the system. Through a meta-programming model, the developer annotates individual C functions to be isolated into the secure world. Based on the annotations, the RT-Trust static and dynamic analyses determine whether the suggested partitioning strategy is feasible, and whether the partitioned system would comply with the original real-time constraints for both the OP-TEE or SGX. A continuous feedback loop guides the developer in restructuring the system, so it can be successfully partitioned. Finally, RT-Trust transforms the system into the regular and trusted parts, with custom generated TEE-specific communication channel between them. If the transformed code fails to meet real-time constraints, it raises custom-handled exceptions. RT-Trust reduces the programmer effort required to partition real-time systems to take advantage of the emerging TEEs.

The contribution of this paper is four-fold:

1. **A Fully Declarative Meta-Programming Model** for partitioning real-time systems written in C to take advantage of the TEEs; the model is realized as domain-specific annotations that capture the requirements of different partitioning scenarios.
2. **Static and Dynamic Checking Mechanisms** that identify whether a system can be partitioned as specified for a given TEE implementation, and how likely the partitioned version is to meet the original real-time constraints. The analyses integrate a feedback mechanism that informs developers how they can restructure their systems, so they can be successfully partitioned.
3. **RT-Trust Refactoring**, a compiler-based program transformation for C programs that operates at the IR level, while also generating customized communication channels and real-time deadline violation handling.
4. **A Platform-Independent Metric** for assessing by how much a CPI function is expected to degrade its performance once moved to the TEE, and comparing such degradations between different TEEs; we evaluate the applicability of this metric on five classic security algorithms and two critical functions in a popular drone controller system.

To concretely realize our approach, we have created RT-Trust as custom LLVM passes and runtime support. Our evaluation shows that RT-Trust saves considerable programmer effort by providing accurate program analyses and automated refactoring. RT-Trust's profiling facilities also accurately predict whether refactored subjects would continue meeting real-time constraints.

This article extends our earlier paper, presented at the 17th International Conference on Generative Programming: Concepts Experience (GPCE 2018) [10]. In comparison to that prior publication, this article reports on the additional research we have performed to enable RT-Trust to support SGX, in addition to the original version that was limited only to the OP-TEE. Our experiences of designing, engineering, and evaluating our approach to support both of these popular TEE implementations should be of value and relevance to the audience of this journal.

The remainder of this paper is structured as follows. Section 2 provides the technical background for this research. Section 3 gives an overview of the RT-Trust toolchain. Section 4 details the RT-Trust meta-programming model. Section 5 and Section 6 further describe the RT-Trust mechanisms for profiling and code refactoring, respectively. Section 7 describes our platform-independent metric. Section 8 describes our evaluation. Section9 discusses the limitations of TEE implementations and RT-Trust. Section 10 discusses related work. Section 11 presents conclusions and future work directions.

2. Background

In this section, we introduce the technical background required to understand our contributions. We briefly discuss CPI, TEE, and real-time constraints. Afterward, we discuss known security risks that motivate this work.

2.1. Critical Program Information (CPI)

Although the concept of critical program information was originally introduced by the US DoD as representing parts of a system that can raise the technological superiority for war-fighters [11], the term has been embraced by all security-sensitive domains. The CPI can include algorithms, data, and hardware of a security-sensitive system. In our design, we designate C functions as constituting CPI, if they happen to contain critical algorithms and manipulate sensitive data. Hence, RT-Trust operates at the function level, including static analysis, profiling, and code transformation. Our declarative programming model provides special-purpose annotations for developers to mark the CPI functions (we detail our programming model in Section 4).

2.2. Trusted Execution Environment (TEE)

TEE [12] offers a standardized hardware solution that protects CPI from being compromised. First, TEE isolates a secure area of the CPU (i.e., the secure world for trusted applications) from the normal area (i.e., the normal world for common applications)[2].

That is, the secure world possesses a separate computing unit and an independent OS that prevents unauthorized external peripherals from directly executing the trusted tasks. In addition, TEE provides trusted storage that can only be accessed via the provided API to securely persist data. Finally, TEE offers an API to the secure communication channel, as the only avenue for external entities to communicate with the secure world.

OP-TEE. [1] Following the Global Platform Specifications of TEE, OP-TEE provides a hardware isolation mechanism that primarily relies on the ARM TrustZone, with three essential features: 1) it isolates the Trusted OS from the Rich OS (e.g., Linux) to protect the executions of Trusted Applications (TAs) via underlying hardware support; 2) it requires reasonable space to reside in the on-chip memory; 3) it can be easily pluggable to various architectures and hardware.

SGX. [2] Another implementation of TEE is Intel's Software Guard Extensions (SGX). It protects computation integrity and confidentiality by extending the Intel architecture. In the same way as OP-TEE, SGX requires that developers divide the original code into two parts: regular and trusted. The former runs

[2]The *normal* and *secure* world are the terms commonly used in the TEE realm. That is, if the code runs in the secure world, it is considered "trusted" (i.e., under protection); if it runs in the normal world, then it is considered "untrusted" (i.e., without protection and may be compromised).

inside of *the enclave*, a protected area that isolates the execution resources from the outside environment (kernel, hypervisor, etc.), in which the latter runs. Furthermore, the regular components can only access the enclave via special CPU instructions. Hence, if run or loaded inside the enclave, the application's CPI becomes invulnerable to attacks perpetrated from compromised outside environments.

2.3. Real-Time Constraints

In general, real-time constraints [13] are the restrictions on the timing of events that should be satisfied by a real-time system; these restrictions can be classified into time deadlines and periodicity limits [14]. The former restricts the deadline by which a particular task must complete its execution. The latter restricts how often a given event should be triggered. For example, given the periodicity limit of 50ms and the time deadline of 20ms, a drone task must obtain its GPS location within 20ms for each 50ms period.

In our case, due to the memory limitation of the TEE, the event's memory consumption is another constraint. As we mentioned in Section 2.2, the TEE should maintain a small footprint by occupying limited space in memory. Also, if the TEE solution applies eMMC RPMB [15] as trusted storage only, the memory consumption is limited by the size of the RPMB partition, due to the persistent objects being stored in the RPMB.

As determined by how strict the timeliness requirements are, real-time constraints are categorized into hard and soft. The former constraints must be satisfied while the latter can be tolerated with associated ranges. For example, a drone's motor/flight surface control must respond on time (hard constraint), while its navigation according to waypoints is expected to be resilient to deviations caused by GPS signal being temporarily lost or even wind gusts (soft constraint).

2.4. Security Risks

Attackers are known to go after compromising CPI. A large amount of known relevant security risks have been reported by the Common Vulnerabilities and

Exposures (CVE) [16]. First, without a proper access control and authentication mechanism for critical functions, attackers can maliciously access and consume the significant amount of resources [17, 18, 19, 20, 21]. Secondly, the possibility of information leakage sharply rises by the vulnerable critical functions [22, 23, 24], especially the functions processing sensitive data. For example, by compromising the data transmitting process, attackers maliciously obtain the current GPS locations [25]. In addition, arbitrarily exposing critical functions for interaction with external actors can be illegally exploited, which causes file deletion [26] or credential disclosure [27]. Further, reverse engineering can disclose critical algorithms [28] or expose sensitive data (e.g., the encryption keys) [29].

3. Solution Overview

In this section, we introduce the toolchain of our compiler-based analyzer and code refactoring tool, and then we describe the input and output of RT-Trust.

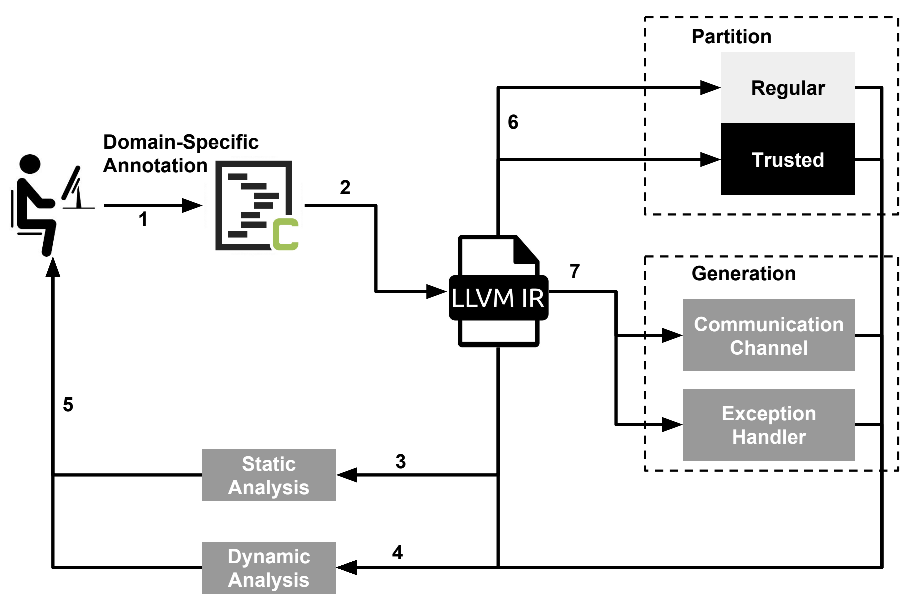

Figure 1: The RT-Trust Process

3.1. Software Development Process

Figure 1 describes the software development process of using RT-Trust to partition real-time systems to take advantage of TEEs. Given a real-time system, the developer first specifies the CPI-dependent functions in the source code using the RT-Trust domain-specific annotations (DSA) (step 1). The annotated source code is then compiled to LLVM intermediate representation (IR). The compilation customizes Clang to specially process the DSA metadata (step 2). After that, RT-Trust determines whether the TEE is implemented as OP-TEE or SGX by inspecting the execution environment or the build configuration. To check whether the specified partitioning scenario can be realized, RT-Trust statically analyzes the system's call graph (step 3). Given the system's call graph and a partitioning specification, RT-Trust constructs the partitionable function graph (PFG), which contains all the information required to determine if the specification is valid. While static analysis determines the semantic validity of a partitioning specification, a separate dynamic analysis phase estimates whether the partitioned system would continue complying with the original real-time constraints. To that end, RT-Trust instruments the system by inserting probes at the IR level (step 4). The inserted probes estimate the partitioning scenarios' memory consumption and function invocation latencies. The system is then exercised under expected loads. The results are then reported back to the developer (step 5). This prior analysis and validation routines make it possible for the developer to modify the original system make it possible to move the CPI functions to execute in the secure world. Once the developer determines that the system can be partitioned with satisfying performance, RT-Trust then automatically divides the system's IR into regular and trusted parts (step 6). The former will be run in the normal world, while the latter in the secure world. To enable these two portions to communicate with each other, RT-Trust generates communication channels customized for OP-TEE and SGX. In addition, to handle the violations of real-time constraints, RT-

TRUST generates exception handling code (step 7). Notice that all these code generation processes are configured entirely by the DSAs applied to the system's CPI functions. Having undergone a partitioning, the system then goes through the final round of verification by dynamically profiling the partitioned system (step 4). The profiling identifies the performance bottleneck while estimating whether the transformed system continues to satisfy the real-time constraints (step 5). Finally, RT-TRUST generates a descriptive report that includes the outcomes of various profiling scenarios and suggestions for the developer about how to remove various performance bottlenecks.

3.2. Code Transformation and Generation

Figure 2 shows RT-TRUST's code transformation and generation. As input, RT-TRUST receives the annotated source code. As output, it transforms the IR of the input source and also generates additional code that is compiled and integrated into the normal and secure world partitions. For the normal world, RT-TRUST transforms the IR by inserting profiling probes, exception handlers, and communication channels. All generated code can be further customized by hand if necessary. The transformed IR code, generated source code (i.e., RPC client stub for OP-TEE and an EDL file for SGX), and referenced libraries (e.g., encryption, profiling) are eventually linked with the normal world's executable. Similarly, for the secure world, the trusted IR, RPC server stub (for OP-TEE), and the referenced libraries are linked with the secure world's executable, which can run only in the secure world of TEE.

4. Meta-programming Model

To accommodate application programmers, RT-TRUST follows a declarative programming paradigm, supported by a meta-programming model. This model makes use of the annotation facility recently introduced into the C language. A C programmer can annotate functions, variables, parameters, and code blocks to assign a customized semantics. The semantics is realized by the compiler by

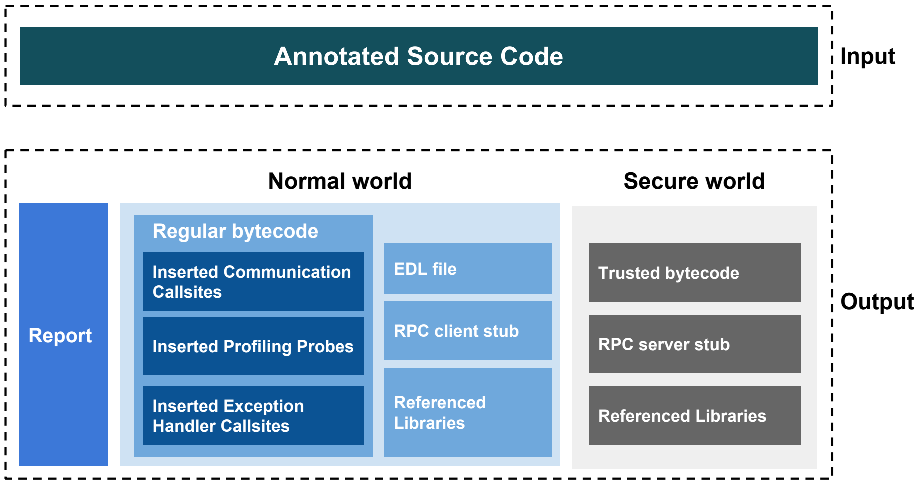

Figure 2: RT-Trust's Input and Output

means of a special processing plug-in. For example, if a function is annotated with `nothrow`, the compiler can check that the function contains no statement that can raise exceptions; if the check fails, an informative message can be displayed to the programmer, who then can modify the function's code accordingly. Despite the large set of built-in Clang annotations [30], none of them are designed for real-time systems and TEE.

For our meta-programming model, we design and implement a set of domain-specific annotations that describe the real-time constraints, code transformation and generation strategies required to automatically transform a real-time system, so its subset can be partitioned to TEE for trusted execution. We call our domain-specific annotations Real-Time Trust Annotations, or RTTA for short. We integrate RTTAs with the base Clang annotation system, so the compiler can analyze and transform real-time systems, as entirely based on the declarative annotations, thus reducing the development burden by enabling powerful compiler-based code analysis and transformation. In this section, we first describe the general syntax of RTTAs. Then, we introduce each annotation and its dependencies in turn. Finally, we illustrate how to use these annotations through an example.

4.1. General Syntax

In the code snippet below, RTTA follows the GNU style [31], one of the general syntaxes supported by Clang. The form of attribute specifier is __attribute__ ((annotation-list)). The annotation list (<annotation-list>) is a sequence of annotations separated by commas. Each annotation contains the annotation name and a parenthesized argument list (<annotation-list>). An argument list is a possibly empty comma-separated sequence of arguments.

```
__attribute__((<annotation-list>))
<annotation-list> ::= <annotation>,<annotation>*
<annotation> ::= name (argument-list)
<argument-list> ::= <argument>,<argument>*
<argument> ::= various arguments
```

4.2. Code Partition Annotation

The code partition annotation informs RT-Trust to perform two tasks: 1) analyze the validity of partitioning for each annotated function, and 2) extract the annotated functions that can be partitioned from the source code. The annotation partition can be applied to any declared function in the source code, and takes no arguments, as follows:

```
__attribute__((partition))
```

4.3. Code Generation Annotations

Code generation annotations that appear in the code snippet below enable developers to customize 1) a specific communication mechanism (e.g., RPC) for the normal and secure worlds to talk to each other, and 2) an exception handler for handling the cases of violating real-time constraints when executing a partitioned system. When annotating with rpc, the developer can specify the shared_memory or socket options as the underlying RPC delivery mechanism. The data transferred between the partitions can be specified to be encrypted or compressed by using the yes and no options. Note that the rpc annotation applies only to OP-TEE to specify how to generate RPC stubs. For SGX, RT-Trust instead generates an EDL file and proxy functions. By annotating pointer and array parameters with paramlen, the developer can indicate their length. The <length> attributes are used by the marshaling and unmarshaling phases on the communication channels. For the pointer parameters, the <length> attribute reports the size of the data the pointer is referencing. Although recent advances in complex static analysis make it possible to automatically infer the size of pointer-based parameters [3], our design still relies on the programmer specifying the length information by hand. This design choice allows for greater flexibility. The paramlen annotation makes it possible for the developer to reserve the required amount of space for the annotated parameters, and then specify how to generate customized marshaling and unmarshaling code. If the developer also annotates that function with memsize, the RT-Trust dynamic analysis suggests an approximated length value (details appear in Section 5.2.2). By annotating with exhandler, the developer can specify how to handle the exceptions potentially raised by the annotated function. The annotation has three parameters: a handler function's name (<method>), the target's real-time constraints (<constraint_type>), and the trigger threshold (<times>) (i.e., the number of times an annotated function can violate the target constraints before triggering the handler function). We explain how RT-Trust generates code, as based on these annotations, in Section 6.

```
__attribute__((rpc(<type>, <encryption>, <compression>)))
<type> ::= shared_memory | socket
<encryption> ::= yes | no
<compression> ::= yes | no

__attribute__((paramlen(<length>)))
<length> ::= n (n is integer, n > 0)

__attribute__((exhandler(<times>, <method>, <constraint_type>)))
<times> ::= n (n is integer, n > 0)
<method> ::= "default" | method name (string)
<constraint_type> ::= exetime | period | memsize
```

4.4. Profiling Annotations

The annotations in the code snippet below configure the RT-Trust profiler to determine if a partitioned system would still meet the original real-time constraints.

Profiling Real-Time Constraints. RTTA provides three annotations for profiling to determine whether given real-time constraints would remain satisfied: 1) `exetime` (i.e., execution time), 2) `period`, and 3) `memsize` (i.e., memory consumption). The `<type>` argument specifies whether the constraint is `hard` or `soft`. The `hard` mode means that violating the constraint is unacceptable, while the `soft` mode means such violations, to some extent, can be accepted. Based on these types, the profiler reports whether the annotated function can be transformed for trusted execution, without violating the specified real-time constraints. For the execution time attribute, the developer can specify the profiling method (i.e., `timestamping` and `sampling`) and the completion deadline (i.e., `<deadline>` to meet. For `period`, one can specify the time interval between invocations of a CPI function. For memory consumption, the memory size can be limited by setting an upper-bound via the `<limit>` argument.

```
__attribute__((exetime(<type>, <method>, <deadline>)))
<type> ::= hard | soft
<method> ::= timestamping | sampling
<deadline> ::= n (n is integer, n > 0)

__attribute__((period(<type>, <interval>)))
<type> ::= hard | soft
<interval> ::= n (n is integer, n > 0)

__attribute__((memsize(<type>, <limit>)))
<type> ::= hard | soft
<limit> ::= n (n is integer, n > 0)
```

4.5. RTTA Dependencies

As compared to the annotations that can be specified independently (e.g., `partition`, `rpc`, and the profiling annotations), other annotations must be specified with their dependencies. For example, the annotation `paramlen` cannot be specified, unless `rpc` also appears among the function's annotations. The `paramlen` annotation is used for generating the marshaling and unmarshaling logic of the communication channels. Likewise, without annotations specifying real-time constraints, the exception handling code is unnecessary: `exhandler` must come together with real-time constraint annotations. The RT-Trust analysis process checks the adherence to these domain-specific semantics of RTTA and reports the detected violations.

4.6. RTTA in Action

Consider the example originally described in Section 1: a drone navigates, with its autopilot continuously obtaining the current geolocation from the GPS sensor to adjust the flying trajectory in a timely fashion. The function of obtaining geolocations is CPI-dependent, and as such should be protected from potential interference by placing it in the secure world. To that end, the developer annotates that function, informing RT-Trust to transform the code, so the function is separated from the rest of the code, while also generating the necessary code for communicating and exception handling. Optionally, the system can be annotated to be profiled for the expected adherence to the original real-time constraints after it would be partitioned. The function `getGPSLocation` annotated with RTTAs appears below. Based on these annotations, our customized Clang recognizes that the function needs to be partitioned and moved to the secure world (`partition`). Meanwhile, RT-Trust will generate a communication channel over shared memory with the encrypted and compressed transferred data between the partitions (`rpc`). In addition, during the marshaling and unmarshaling procedure, the allocated memory space for the function's parameter will be 100 bytes (`paramlen`). Further, RT-Trust will insert the measurement code to profile the function's real-time constraints. It instruments the function's execution time with the `timestamping` algorithm and `hard` mode to check whether it meets the deadline (20 ms) (`exetime`), and checks whether the invocation interval would not exceed 50 ms (`period`). It estimates the memory consumption, and checks whether it exceeds 1024 bytes in the `soft` mode (`memsize`). Finally, if the real-time deadline constraint has been broken more than once, it will be handled by the exception handler function "myHandler" (`exhandler`). The declarative meta-programming model of RT-Trust automates some of the most burdensome tasks of real-time system profiling and refactoring. In the rest of the manuscript, we discuss some of the details of the RT-Trust profiling, code transformation, and code generation infrastructure.

```
Location loc; // global variable
Location getGPSLocation // CPI function
(GPSState * __attribute__((paramlen(100))) state)
__attribute__(( partition,
    rpc(shared_memory, yes, yes),
    exhandler(1, "myHandler", exetime),
    exetime(hard, timestamping, 20),
    period(hard, 50),
    memsize(soft, 1024) )) {...}
// adjusting Drone direction
void adjustDirection(Location l) {...}
void fly() {
    loc = getGPSLocation(state);
    adjustDirection(loc);
}

int main() {
fly(); ... }
```

5. Analyses for Real-Time Compliance

The automated refactoring described here has several applicability limitations. One set of limitations stems from the structure of the system and its subset that needs to be moved to the trusted partition. Another set of limitations are due to the increase in latency that results in placing a system's subset to the trusted execution zone and replacing direct function calls with RPC calls. The increase in latency can cause the system to miss its real-time deadlines, rendering the entire system unusable for its intended operation. To check if the structure of the system allows for the refactoring to be performed, RT-Trust features a domain-specific static analysis. To estimate if the refactored system would still meet real-time requirements, RT-Trust offers several profiling mechanisms, which are enabled and configured by means of RTTAs.

5.1. Static Analysis

The TEE implementation in place (i.e., OP-TEE or SGX) determines whether RT-Trust can realize a given partitioning scenario. That is, a scenario may work on the OP-TEE but not on the SGX, and vice versa. To that end, RT-Trust not only allows the developer to specify the TEE implementation, but it also automatically inspects the compilation environment to determine the TEE implementation. After that, RT-Trust checks whether the scenario adheres to the following three rules, referred to as `zigzag`, `pointers`, and `global variable`. If the code passes all three checks, RT-Trust can successfully carry out the specified partitioning scenario. A failed check report identifies why the code needs to be refactored to make it amenable to partitioning.

Zigzag Rule. Consider a set of functions T_1, annotated with the `partition` annotation, and another set of functions T_2, containing the rest of all the functions. The zigzag rule defines the restrictions imposed by different TEEs:

For OP-TEE, the zigzag rule states that functions in T_2 cannot invoke functions in T_1, as such invocations would form a zigzag pattern. This restriction is caused by the strict one-way invocation of the functions in the trusted zone from the normal world. The normal world can call functions in the trusted zone, but not vice versa. One can fix violations of the zigzag rule by annotating the offending function, called from the trusted zone, with `partition`, so it would be placed in the trusted partition as well, so it would be invocable via a local function call. Our assumption of relying on the static version of the call graph is reasonable for the target domain of real-time systems written in C, in which functions are bound statically to ensure predictable system execution.

For SGX, the zigzag rule states that even though functions in T_2 can invoke functions in T_1, such invocations must be restricted to some small number (i.e., threshold) due to the high communication latency between the normal and secure worlds. That is, although SGX supports the zigzag calls, the program performance suffers from the high latency of such invocations [32]. One can tune the threshold to balance the trade-off between efficiency and utility. Once the threshold comes to "0", the zigzag rule regresses to the one used for OP-TEE.

Global Variable Rule. Since the partitioning is performed at the function level, the distributed global state cannot be maintained. As a result, each global variable can be placed either in the normal or trusted partition and accessed locally by its co-located functions. Violations of this rule can be easily detected. One exception to this rule is constant global variables, which due to being unmodifiable can be replicated across partitions.

Pointers Rule. The pointers rule restricts the types that can be used as parameters of the partitioned functions: 1) function pointers and pointer arrays cannot be passed as parameters, and 2) struct parameters cannot contain pointer members. For SGX, RT-Trust strictly enforces this rule, as the SGX Enclave Definition Language (EDL) has no support for such pointer types. However, for OP-TEE, only function pointers cannot be supported. For their code to abide by this rule, developers can refactor the target program, so the partitioned functions take no such pointer parameters. Alternatively, developers can manually implement specialized logic for marshaling/unmarshaling these parameters.

Partitionable Function Graph. To check the above rules, RT-Trust introduces a partitionable function graph (PFG). This data structure extends a call graph with special markings for the functions that can be partitioned. To construct a PFG, RT-Trust starts by walking the call graph for the functions annotated

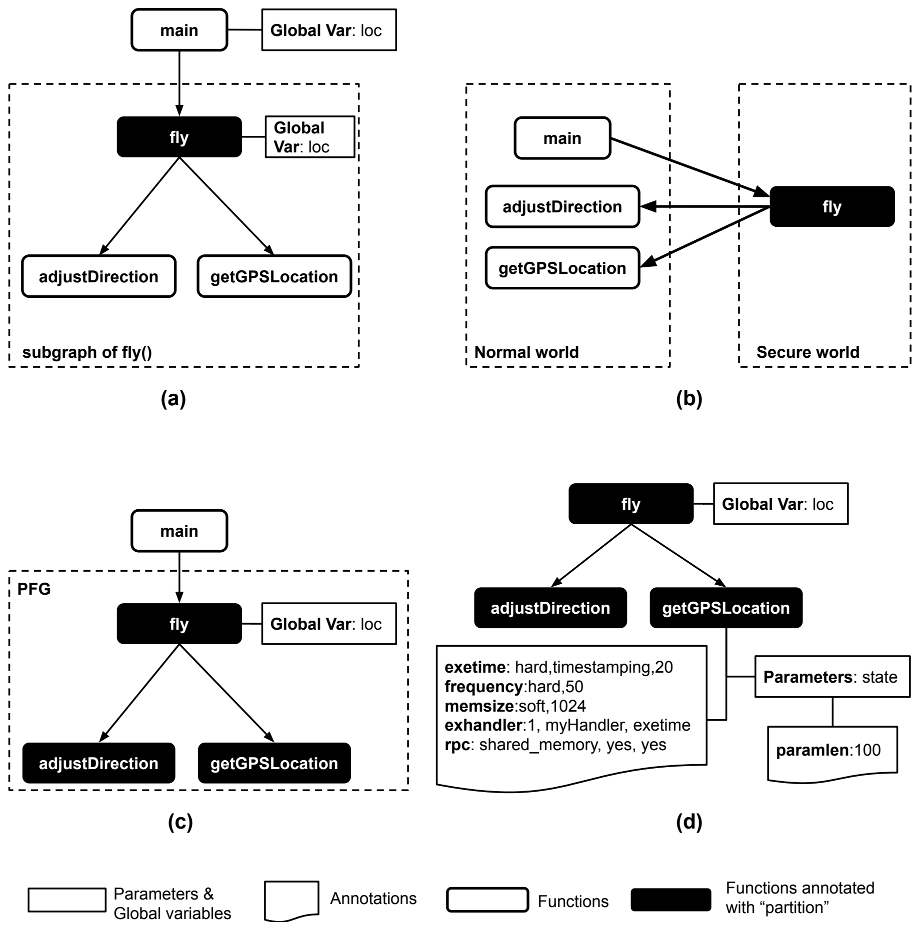

Figure 3: The RT-Trust PFG

with `partition`. By checking whether these functions comply with the zigzag and global variable rules, it removes the function nodes that break these rules. The resulting graph is the PFG.

Specifically, RT-Trust sets each function annotated with `partition` as the root function, and then traverses its subgraph. During the traversal, RT-Trust checks whether all subgraph elements are also annotated with `partition`. If so, RT-Trust adds the entire subgraph to the PFG, and then moves to the next annotated function. After examining the zigzag rule, the PFG contains several sub-callgraphs of non-zigzag functions annotated to be partitioned. Next, RT-

TRUST collects global variable information for each function already in the PFG. It then examines whether the variables are operated by the functions in the PFG only. If so, RT-TRUST adds these functions to the PFG. Otherwise, RT-TRUST removes the entire subgraph containing the violating function from the PFG. The final PFG contains all the necessary information (e.g., global variables, parameters, and annotations) required to partition the system. We deliberately chose to exclude any automatically calculated dependencies of the annotated functions, requiring the programmer to explicitly specify each function to be placed into the trusted zone in order to prevent any unexpected behavior.

Recall the example in Section 4.6: if the developer annotates only function `fly` as `partition`, as shown in Figure 3 (a), the sub-callgraph of `fly` is `fly` → `getGPSLocation` and `fly` → `adjustDirection`. In that case, placing function `fly` in the trusted partition leads to zigzag invocations between the normal and secure worlds (Figure 3 (b)). If `fly` runs in OP-TEE, or in SGX configured for the minimal zigzag call (i.e., the threshold of "0"), this partitioning specification violates the zigzag rule. To fix such violations, the developer can annotate the other two offending functions (i.e., `getGPSLocation` and `adjustDirection`) with `partition`, so that both of them will also be placed in the secure world along with their caller `fly`. After the zigzag violation is eliminated, RT-TRUST then adds `fly`'s sub-callgraph to the PFG.

Now, suppose the global variable `loc` are accessed not only by function `fly` (i.e., the secure world) but also by function `main` (i.e., the normal world). Because this scenario violates the global variable access rule, the entire sub-callgraph of `fly` should be removed from the PFG. To fix this violation, the developer can modify function `main`, so it would no longer access `loc` (Figure 3 (c)), or make this global variable constant. Finally, RT-TRUST constructs the PFG with all the necessary information for each function, as shown in Figure 3 (d).

5.2. Dynamic Analyses

RT-TRUST offers dynamic analyses to help identify how likely the specified partitioning would meet the original real-time constraints. Since it would be hard to guarantee whether the profiled execution produces the worst-case scenario, our analyses are applicable only to soft real-time systems. Figure 4 shows how RT-Trust provides the dynamic analyses capability. The analyses start with the transformation of the original LLVM IR program. That is, RT-Trust inserts profiling code at the affected call sites of the annotated functions for their corresponding real-time constraints. Instead of inlining the entire profiling code, RT-Trust inserts calls to special profiling functions, which are made available as part of shared libraries. Currently, RT-Trust provides them on its own, but similar profiling functionality can be provided by third-party libraries as well. This flexible design enables developers to provide their custom profiling libraries or add new features to the libraries provided by RT-Trust to further enhance the profiling logic. After linking these shared libraries with the transformed IR program, developers run the executable to trigger the inserted function calls to invoke the profiling functions in the shared libraries. These functions measure the real-time constraints and persist the result data for future analysis. Finally, RT-Trust analyzes the data, estimating whether the annotated functions can meet the original real-time requirements, and reporting the results back to the developer.

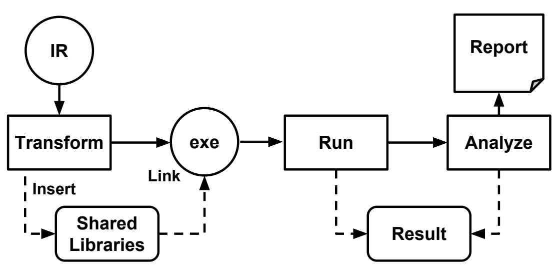

Figure 4: The RT-Trust Analyses Procedure

5.2.1. Analyzing Time Constraints

As mentioned in Section 2, time constraints mainly include the time deadline and the periodicity limit. The former defines the upper boundary for a function to complete its execution, the latter restricts the time that can elapse between any pair of invocations.

To analyze these constraints, RT-Trust first transforms the original LLVM IR program via two key steps: 1) find the correct call sites, and 2) insert the suitable function calls. In the transformation procedure below, given a function annotated with `exetime`, RT-Trust traverses its instructions to locate the first instruction in its entry basic-block[3], inserting the profiling probes and then that starts a profiling session. Likewise, RT-Trust locates each return instruction of the annotated function, inserting the probes that issue the end profiling session, which stops the profiling.

```
define i32 @function(i8* %param) { // annotated function
    entry:
     <--- start probe()
    %first instruction
    ...
     <--- stop probe()
    ret i32 %retval
}
```

Which probe functions are inserted depends on how RT-Trust is configured by means of RTTAs. The two main configurations are timestamping and sampling. For timestamping, RT-Trust inserts probes that invoke the timestamp functions to retrieve the current system time by means of `gettimeofday()` (in the normal world), or `TEE_GetREETime()` (in the secure world to check the adherence to real-time constraints post-partitioning). For sampling, RT-Trust inserts invocations to the sampling functions of `ProfilerStart()` and `ProfilerStop()`, which make use of gperftools (a third-party profiling tool). Similarly, to analyze periodicity limits, RT-Trust locates the first instruction of the function annotated with `period`, and then inserts invocations of the functions to record the current system time.

All these measured results are first stored in a hash table, with the key

[3]Basic-block is a straight-line code sequence. It has no *in* branches, except at the entry, and no *out* branches, except the exit.

corresponding to the annotated function's name and the value to its profiling record. Finally, the hash table is persisted into an external file for further exploration.

5.2.2. Memory Consumption Profiling

Memory consumption is an important issue for trusted execution. First, TEEs are designed to occupy limited memory space (as discussed in Section 2). In addition, pointer parameters of the trusted functions refer to data structures that need to be dynamically allocated as part of their marshaling/unmarshaling phases (as discussed in Section 4.3). To ascertain the expected memory consumption requirements of the CPI functions, RT-Trust profiles the amount of memory consumed by the functions annotated with `memsize`. The profiling comprises the traversal of the functions' IR instructions to locate all the allocation sites (i.e., the `alloca` instruction). Each allocation site is then instrumented to keep track of the total amount of allocated memory.

```
%var = alloca i32, align 4
<--- function(i32, 4)
```

The allocated memory volume is continuously monitored as the profiled system is being executed. The presence of pointers complicates the profiling procedure. To properly account for all the memory consumed by the data structure referenced by a pointer, RT-Trust implements a heuristic approach based on SoftBound [33]. To provide effective memory safety checking, SoftBound transforms the subject program to keep the base and bound information for each pointer as metadata. This metadata is passed along with the pointer. In other words, when passing the pointer as a parameter from one function to another, the metadata is also be passed. SoftBound makes use of this metadata to enforce program memory safety.

Based on SoftBound, RT-Trust inserts invocations to record the pointer metadata (base and bound) of the annotated function, whenever pointers are allocated or accepted as parameters from other functions. RT-Trust calculates each pointer's length via the formula $length = bound - base$. By combining the basic and pointer type's lengths, RT-Trust finally determines the upper boundary of the memory volume consumed by each annotated function.

5.3. Exception Handling

Having annotated a function with real-time constraints, developers can also specify how to handle the violation of these constraints via the `exhandler` annotation. To locate the correct call site for inserting exception handling code, RT-Trust traverses instructions of each defined function in the original program, finding the invocations to the annotated functions. Then, RT-Trust inserts "if-then-else" blocks by means of LLVM API `SplitBlockAndInsertIfThenElse`. The "if-then-else" blocks include: 1) the block that contains `if` condition, 2) "then" block, 3) "else" block, and 4) the block after "then" and "else" blocks. RT-Trust creates an `if` condition with the annotated threshold for the number of violations of a given real-time constraint. Then, it inserts the invocation to the specified exception handling function into the "then" block, and inserts the invocation to the original function into the "else" block as follows:

```
Ret = function(Args); // is transforms into:
Ret = (t reaches threshold) ? exhandling_function(Args)
                            : function(Args);
```

Then, RT-Trust inserts another invocation before the "if-then-else" blocks to calculate the number of observed violations of the given real-time constraint (i.e., "t" in the above code snippet). Finally, the inserted code logic can automatically switch between the original function and the exception handling function, which can be specified by the developer or generated by RT-Trust as a default option.

6. Inter-World Communication: Code Generation & Transformation

The partitioning process divides the program's IR into the trusted and regular parts. Our partitioning strategy is function-based: CPI-dependent functions execute in the trusted partition, while all other functions execute in the regular one. The TEE isolation mechanisms make it impossible to directly invoke CPI functions running in the trusted partition. However, each TEE provides special communication channels that can be accessed through environment-specific APIs. Hence, RT-Trust replaces the direct CPI function invocations with communication through the TEE channels for both OP-TEE and SGX.

For OP-TEE, RT-Trust first generates an RPC client stub (for the normal world) and a server stub (for the secure world). The client stub passes the function's parameters and its unique ID, which identifies the function to execute in the secure world. The server stub receives this information and invokes the corresponding CPI function in the trusted partition. For SGX, RT-Trust generates a proxy for each CPI functions and an Enclave Definition Language (EDL) file that provides metadata for all the CPI functions. By passing the generated EDL file as input to the Edger8r tool [34], developers then generate the required SGX communication logic for all interactions between the regular and trusted parts. For both OP-TEE and SGX, RT-Trust redirects the direct invocation of a CPI function to its RPC stub (for OP-TEE) or its proxy function (for SGX).

6.1. Generating RPC stubs for OP-TEE

RT-Trust generates RPC stubs based on the developer's configuration in annotation `rpc` and `paramlen`. The argument `<type>` of `rpc` specifies which underlying delivery mechanism (i.e., shared memory or socket) to generate. This delivery mechanism also depends on the actual TEE implementation in place. To exchange data between the normal and secure worlds, OP-TEE provides 4 shared memory buffers, used as the delivery mechanism. However, RT-Trust must marshal/unmarshal function parameters to and from these buffers. This explicit parameter marshaling makes the generated code suitable for any communication mechanism.

The client stub includes four code sections: 1) `prologue` initializes the TEE context and opens the communication session, 2) `epilogue` closes the session and finalizes the context, 3) `marshaling` allocates memory space and marshals the function's parameters, and 4) the RPC function communicates between the normal and secure worlds by calling TEE API methods `TEEC_InvokeCommand`. Correspondingly, the server stub also includes four code sections: 1) the entry points of opening and closing the communication session, 2) `unmarshaling` unmarshals the received data, 3) a dispatcher that receives invocations and data from the client stub, and forwards it to corresponding CPI wrapper functions, and 4) the wrapper functions receive the data from the dispatcher and invoke the actual CPI functions in the trusted partition.

During the code generation, RT-Trust checks the arguments `<encryption>` and `<compression>` of annotation `rpc`. If the developer specifies that `<encryption>` or `<compression>` is needed, RT-Trust encrypts and compresses the data after the `marshaling` phase in the client stub, and decrypts and decompresses the data before `unmarshaling` phase in the server stub. Although RT-Trust uses existing open source libraries for encryption and compression, developers can switch to using different implementations. Further, when generating the `marshaling` component for the client stub, RT-Trust checks the `paramlen` to determine how much memory to allocate.

For ease of portability, all generated code is compliant with the C language specification, without any custom extensions. Furthermore, all the referenced libraries are open source and plug-in replaceable. Finally, all the TEE APIs in the generated code conform to the Global Platform Specification of TEE. Thus, developers can either directly use the generated code for the trusted execution or extend that code in order to meet some special requirements.

6.2. Generating proxy functions and EDL file for SGX

Based on the partitionable functions' information in the PFG, RT-Trust generates an EDL file, assembling the declarations of trusted functions into the "trusted" block, and that of regular functions invoked from the trusted part in a zigzag pattern into the "untrusted" block. Most importantly, for each pointer parameter in both the trusted and untrusted function blocks, RT-Trust checks the `paramlen` annotation to generate the EDL attributes that determine the size of pointer-based parameters. For each function containing `struct` parameters, RT-Trust generates a complete definition of each `struct` in the EDL file. After that, RT-Trust generates a proxy function file to initialize/deallocate the communication channel and to handle the return values for each CPI function. Finally, RT-Trust executes the Edger8r tool to generate the required SGX communication logic for this partitioning scenario.

6.3. Redirecting Function Calls

As CPI functions are moved to the secure world, their callers need to be redirected to invoke the original function's RPC stubs (for OP-TEE) or proxy functions (for SGX) instead. RT-Trust exhaustively examines all function invocation instructions, locates the ones invoking the CPI functions, and replaces the callee's name to the CPI function's RPC stub or proxy function. Since CPI functions and their RPC stubs / proxy functions share the same signature, no other changes are necessary:

```
Ret = original_function(Args); // is transformed into:
Ret = RPC_function(Args); // for OP-TEE
Ret = un_function(Args); // for SGX
```

Now, the original function calls become RPC or proxy function invocations that end up calling the partitioned CPI functions in the secure world. As per the transformation of exception handling in Section 5.3, the original function can be specified to handle exceptions. That is, if the violations of real-time constraints reach the threshold, the inserted exception handling logic can automatically change back to invoking the original function rather than the function in the secure world:

```
Ret = RPC_function(Args); //is transformed into:
// for OP-TEE:
Ret = (reach threshold) ? original_function(Args) : RPC_function(Args);
// for SGX:
Ret = (reach threshold) ? original_function(Args) : un_function(Args);
```

6.4. Data Encoding Protocols

The normal and secure worlds are represented by distinct system components, running in separate address spaces. The inter-process communication facility, through which the worlds interact with each other, require that all the data passed between them be encoded as an array of bytes. RT-Trust has to be able to encode the regular part's data structures into this array of bytes, while the corresponding trusted part has to read these data structures from the array once it is transferred to the secure world. This problem is not new, and multiple marshaling mechanisms [35] have been introduced, including major framework platforms, such as CORBA [36] and gRPC [37]. For SGX, the Edger8r Tool parameterized with an Enclave Definition Language (EDL) file [38] automatically generates the required marshaling/unmarshaling logic. However, OP-TEE provides no such marshaling/unmarshaling facilities. To solve this problem, RT-Trust provides a custom marshaling framework that not only generates the required marshaling/unmarshaling logic for the parameters of CPI functions, but also introduces a novel space-efficient encoding for data collections. Given that TEE is frequently used as a secure data storage, this ability to encode data collection parameters space-efficiently increases the applicability of RT-Trust.

Figure 5 shows how RT-Trust differently encodes parameters that are: a) primitive types (e.g., `int`, `char`, `double`), and b) complex type (e.g., `struct`, `union`). The encoding represents all data as a byte array, and when storing both primitive and complex data, it starts with the same header that contains the `total len` (the total length of all the entries in this encoding), and `num` (the total number of items in the encoded collection) fields. These fields are both stored into a 4 bytes integer. The following entries differ depending on the encoded type. For primitive types, RT-Trust then stores the size of the encoded data type, which is then followed by the actual data content. For complex types, RT-Trust first stores the type header: the `total len` (the total length of all the members in this type), and `num` (the total number of members in this type) fields, followed by the size of each member and its actual content in turn. This scheme enables the receiving party to first extract the total length to be able to allocate the amount of memory required to contain the entire encoding. The transfer process needs to allocate memory twice: first in the shared memory, which serves as a delivery vehicle to the secure world, and then in the trusted part to be able to store the transferred data.

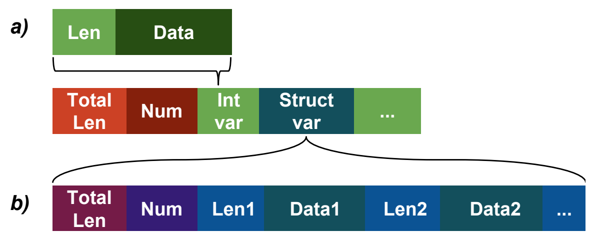

Figure 5: Format of Data Transmission.

7. Support for Partitioning Decision Making

As discussed in Section 4.4, for each function to partition, developers can indicate whether it must abide by `hard` or `soft` real-time constraints. Hard constraints cannot be violated, while soft ones can tolerate some violations. Hence, upon detecting a possible violation of a hard constraint, RT-Trust rejects the request to partition the offending function. For compliant CPI functions and those violating only the soft constraints, RT-Trust calculates their Function Performance Indicator (FPI) discussed next.

Function Performance Indicator. The Function Performance Indicator (FPI) reflects by how much a CPI function is expected to degrade its performance once moved to the TEE. For each appropriate CPI function, RT-Trust calculates and reports its *FPI*, upon which developers can determine whether or not to move the function to TEE. *FPI* correlates two platform-independent metrics: execution time loss (L_{exe}) and invocation interval loss (L_{inv}). We calculate the expected performance degradation (T_{after}/T_{before}), and then scale and normalize it by applying *log* and *tanh* functions in turn[4].

Finally, we calculate the maximum value of the normalized results to obtain FPI :

$L_{exe} = T_{after}/T_{before}$; ($T_{before}$, T_{after} are execution times) (1)

$L_{inv} = I_{after}/I_{before}$; ($I_{before}$, I_{after} are invocation intervals) (2)

$FPI = Max(tanh(log(L_{exe})), tanh(log(L_{inv})))$ (3)

FPI shows the expected performance degradation factor. Notice that *FPI* can take upon values that range between 0 and 1. We offer the following guidelines to developers, as based on the ranges of *FPI* values: between 0 and .25, the expected degradation is *minimal*; between .26 and .75, the degradation is *medium*; and between .76 and 1, the degradation is *high*. Which level of performance degradation is acceptable for a given application scenario is up to the developer to determine.

For example, a CPI function `f` is annotated to be moved to TEE. Before moving `f`, its execution time and invocation interval are 1 and 5 seconds, respectively. After moving `f` to TEE, its time and interval become 10 and 20 seconds, respectively. Hence, `f`'s L_{exe} is $10/1 = 10$, L_{inv} is $20/5 = 4$, resulting in *FPI* of $Max(tanh(log10), tanh(log4)) = 0.76$. In other words, moving `f` to TEE would increase its execution costs by a factor of 0.76. This performance degradation level is in the low range of high.

As a simple but intuitive metric, *FPI* provides a convenient heuristic that can help developers determine whether moving a CPI function to the TEE would continue satisfying the timeliness requirements. Under SGX and OP-TEE, *FPI* can differ for the same CPI functions. So this metric can also help developers select the most appropriate TEE implementation for a given real-time system.

[4]The *log* and *tanh* functions are classic data analysis tools. Here we apply *log* to display a large range of quantities in a small scale, and apply *tanh* to normalize the scaled result to fall within the range of 0 to 1.

8. Evaluation

We answer the following research questions in our evaluation:

- **Effort:** How much programmer effort is saved by applying RT-Trust?
- **Performance:** What is the added performance overhead imposed by performing a RT-Trust profiling on a representative real-time system?
- **Value:** How effectively can RT-Trust determine whether a planned refactoring would preserve the original real-time constraints?
- **Accuracy:** How accurately can our profiling infrastructure predict the expected performance deterioration caused by a RT-Trust refactoring?
- **Limitations:** What are some limitations of RT-Trust's applicability?

8.1. Experimental Setup

To answer the evaluation questions above, we have concretely implemented RT-Trust and assessed its various characteristics in a realistic deployment scenario, whose experimental setup is as follows.

Software and Hardware. RT-Trust integrates RTTAs with the public release of Clang 4.0 and implements a series of LLVM Passes (e.g., code analysis, partition, RPC stubs generation, profiling code insertion) in LLVM 4.0. Since our memory consumption profiler relies on SoftBound, which runs only in LLVM 3.4, RT-Trust implements a separate LLVM Pass that profiles the memory consumed by specified functions in that earlier LLVM version. For OP-TEE, the benchmarks that we use for evaluating RT-Trust are set up on Raspberry Pi 3 (RPi3), running OP-TEE 3.1.0 on Linux version 4.6.3, 1.4GHz 64-bit quad-core ARMv8 CPU, and 1 GB SDRAM. For SGX, the evaluation environment are set up on a Dell workstation, running Intel SGX Linux 2.0 Release on Ubuntu 16.04, 3.60GHz 8-core Intel i7-7700 CPU, with 31.2 GB memory.

Microbenchmarks and Realistic real-time system. Real-time systems that can benefit from RT-Trust possess two characteristics: 1) have CPI-dependent functions that should be protected in the secure world, and 2) have the execution of these functions restricted by some real-time constraints.

To establish the baseline for the performance behavior of such systems, we choose several classic algorithms as our microbenchmarks, which are widely used by existing real-time system. To mimic the real-time invocations of our microbenchmarks, we have written custom unit test suites that exercise the CPI-dependent functionality. For example, we simulate the invocation of a certain algorithm 50 times. The selected benchmarks are algorithmic in nature and include CRC32, DES, RC4, PC1, and MD5. One can imagine realistic application scenarios, in which the execution of these benchmarks needs to be protected under real-time constraints. Because both OP-TEE and SGX support only C code as running in the secure world, we select the C implementations of these algorithms provided by one of the LLVM test suites [39].

To ascertain the applicability of RT-Trust to an actual real-time system, we apply it to secure two CPI tasks of an open-source autopilot PX4 (v1.8.0) [40]: airspeed and waypoint computations.

Evaluation Design. As described in Section 5 and 6, developers can customize the implementations of profiling, EDL file and RPC stubs. However, we evaluate only the default options of using RT-Trust to establish its baseline performance, thus not unfairly benefiting our implementation.

We evaluate programmer effort as the uncommented lines of code (ULOC): 1) those required to write RTTAs, 2) those automatically generated by RT-Trust, and 3) those that the developer is expected to fine-tune by hand (e.g., some source code may need to be modified to fix the violations of our partitioning rules, or the parameter's length in an RPC stub / EDL file may need to be manually adjusted). Note that RT-Trust generates tight code, without any redundancies or unnecessary features, very similar to what a programmer would write by hand. Hence, we argue that without RT-Trust, programmers would be writing all the generated code by hand. By reporting on the size of this code, we measure how much programmer effort RT-Trust saves.

To evaluate performance, we measure the overhead of RT-Trust's profiling for execution time, invocation interval, and memory consumption. For the former two, RT-Trust provides different profiling libraries, applying TEE (i.e., OP-TEE or SGX) APIs in the secure world. So we evaluate them in both the normal and secure worlds. For the latter, memory consumption should be profiled before partitioning and generating RPC stubs or the EDL file. So, we evaluate it only in the normal world.

To evaluate value and accuracy, we first apply RT-Trust to profile the specified CPI functions before and after moving them to the secure world. Then, we compare the results reported by the profiling of the original unpartitioned system with respect to meeting the real-time constraints with that of its partitioned version.

However, the time measurement's granularity in the OP-TEE time API differs from that in the SGX API, which reports the time-elapsed quantities only at the seconds level of granularity. To effectively measure the CPI functions' performance (at the milliseconds level) under SGX, we modified the source code to repeat each benchmark 1000 times. Despite these repeats, we report the final results at the millisecond level of granularity by simply dividing them by 1000. By using the same measurement unit for both OP-TEE and SGX, our experimental results provide a realistic comparison of the expected performance degradation levels imposed by these TEE implementations. Also, by using *FPI*, developers can effectively compare the performance of a given CPI function in different TEE implementations.

Further, by analyzing the performance results, we discuss 1) which procedure causes the performance deterioration after moving the CPI function to the secure world, 2) whether we can accurately predict the specified function's performance in the secure world by analyzing its performance in the normal world, and 3) which TEE implementation can better preserve the timeliness requirements of our evaluation cases. To explain RT-Trust's limitations by describing several program cases that require a prohibitively high programmer effort to adjust the generated RPC stubs.

8.2. Results

We verify the correctness of RT-Trust by applying all its LLVM passes (i.e., code analysis, transformation, and generation) to microbenchmarks. We evaluate RT-Trust as follows.

Table 1: Programmer Effort (ULOC)

Algorithm	RTTAs	Generate & Transform		Adjust	
		OP-TEE	SGX	OP-TEE	SGX
CRC32	5	388	87	0	0
PC1	4	344	73	6	6
RC4	3	292	61	3	1
MD5	3	364	86	3	1
DES	2	244	46	15	3

Effort. Table 1 shows the effort saved by applying RT-Trust. Generally, the total number of ULOC automatically generated and transformed by RT-Trust ($244 \sim 388$ ULOC for OP-TEE; $46 \sim 87$ ULOC for SGX) greatly surpasses those required to manually annotate (< 5 ULOC) and modify ($0 \sim 15$ ULOC) the subject programs.

RT-Trust eliminates the need for the developer to write this code. In other words, to apply RT-Trust, the developer adds a tiny number of ULOC, mainly as annotations and minor adjustments of generated code. The number of annotations is directly proportional to the number of CPI functions. The manual adaptations are required to remove program patterns that prevent RT-Trust from successfully partitioning the code, and to support the pointer parameters of CPI functions.

Specifically, to move the 5 CPI functions of CRC32 to the secure world requires exactly 5 ULOC of RTTAs. No manual adjustment is necessary, as the code comes amenable to partitioning and no pointer parameters are used. In contrast, 15 (for OP-TEE) and 3 ULOC (for SGX) are required to adjust the generated RPC communication for DES, due to a CPI function's pointer parameter pointing to a `struct` of two `char` arrays. In other words, after profiling the amount of consumed memory, the developer needs to adjust the memory allocation for marshaling/unmarshaling these pointer parameters. For PC1, 6 additional ULOC are needed to fix a violated global variable rule.

Overall, the number of generated and adjusted lines of code needed for SGX is generally fewer than those for OP-TEE. The reason is that, for SGX, RT-Trust only needs to generate an EDL file to construct the communication channel, while the developer only needs to modify the size or count modifiers in the EDL file to adjust the amount of memory allocated for the pointer parameters.

Table 2: Overhead of RT-Trust profiling (ms)

Algorithm	Execution Time		Invocation Intervals		Memory	
	Normal	Secure	Normal	Secure	Parameter	Local
OP-TEE	0.442	144	0.418	139	0.051	0.053
SGX	0.2	52	0.212	25.5		

Performance. Table 2 reports on the overhead of RT-Trust profiling, which captures and calculates the execution time, invocation intervals, and memory consumption. Recall that RT-Trust profiles systems *before* and *after* refactoring them. The *before* mode estimates whether the refactored system would continue meeting real-time constraints, while the *after* mode compares the estimated execution characteristics with those performed on TEE hardware (OP-TEE on a Raspberry Pi3 and SGX on a Dell workstation). Hardware environments heavily impact the profiling overhead, with an order of magnitude difference: for OP-TEE, $\approx 0.4ms$ in the normal world vs. $\approx 140ms$ in the secure world. For SGX, $\approx 0.2ms$ in the normal world vs. $\approx 50ms$ in the secure world.

This drastic performance difference is mainly due to the differences between the efficiency of standard Linux system calls and their TEE counterparts. For example, the standard `gettimeofday` is more efficient than either `TEE_GetREETime` in the OP-TEE or `sgx_get_trusted_time` in the SGX.

The heavy performance overhead of trusted execution prevents the profiling of real trusted system operation. When estimating memory consumption, the overhead of capturing the memory allocated for local variables and the pointer parameters never exceeds $0.06ms$. However, the overall overhead depends on the total number of local variables and pointer parameters. For example, if a function allocates memory for n variables, the total overhead would be $\approx 0.053 * n$ (ms). Thus, to prevent the profiling overheads from affecting the real-time constraints, the RT-Trust profiling is best combined with the system's testing phase.

Table 3: Value and Accuracy of RT-Trust (ms)

Alg.	Comm.		Execution Time				Invocation Interval				Memory (bytes)	
			Before		After		Before		After		Parameter	Local
CRC32	253.17	28.91	1.150	0.21	1.3	≈ 0.0	1.240	0.23	269	29.15	40	92
PC1	273.38	29.03	68.22	7.64	13	8.95	68.10	8.10	314	37.67	32	22
RC4	236.96	29.62	500.52	32.89	447	66.00	506.95	33.19	705	97.10	240	1144
MD5	177.83	30.90	267.43	49.72	254	49.35	267.62	51.08	446	78.71	20000	316
DES	201.99	28.84	24.18	2.51	32	3.18	24.30	2.55	224	31.56	528	72
airspeed	256.35	32.87	≈ 0.0	0.01	≈ 0.0	≈ 0.0	50.16	53.75	305.0	83.29	12	12
waypoint	264.96	32.38	0.400	0.05	0.460	≈ 0.0	500.75	505.98	773.67	533.32	40	40

Value and Accuracy. Table 3 shows the results of profiling the CPI functions, with the profiling overhead subtracted. The value before "$\big|$" is the results for the OP-TEE, and after "$\big|$" is that for the SGX. For the execution time, generally, the time consumed by our micro-benchmarks and the CPI PX4 functions in the secure world ("After" column) is similar to that in the normal world ("Before" column). Hence, moving the CPI functions to TEE should not deteriorate their performance. Thus, it is reasonable to estimate the performance in the secure world based on that in the normal world. However, the communication channel between the normal and secure worlds slows down the invoked functions due to the introduction of two time-consuming mechanisms: connection maintenance to the secure world (e.g., initialize/finalize context, open/close session), and invoking the partitioned functions in the secure world (e.g., allocate/release shared memory, marshal and unmarshal parameters).

Given a real-time deadline to complete the execution of a CPI function, the post-refactoring profiling helps determine if the deadline is being met. The source code for PX4's airspeed calculation sets the *execution timeout* to 300 milliseconds. Since the maximum post-refactoring latency of 256.35 (in OP-TEE) is below this deadline, moving this CPI function to TEE preserves its real-time constraints.

The time spent in the communication channel increases the invocation intervals of our micro-benchmarks and the CPI PX4 functions. The micro-benchmarks invoke functions consecutively in a loop. Thus, in the normal world, each function's invocation interval ("Before" column of "Invocation Interval") is similar to its execution time ("Before" column of "Execution Time"). However, in the secure world, these invocation intervals increase, becoming similar to the time consumed by Communication ("Communication" column) plus the time in the secure world ("After" column of "Execution Time"). For the PX4 autopilot, which computes the airspeed and next waypoint values every 50ms and 500ms, respectively, the time spent in the communication channel increases these invocation intervals to 305ms ($\approx 256.35(communication) + 0$(execution time) +50) and 773.67ms ($\approx 264.96(communication) + 0.46$ (execution time) +500) in OP-TEE, and to 83.29ms ($\approx 32.87(communication) + 0$(execution time) +50) and 553.32ms ($\approx 32.38(communication) + 0$(execution time) +500) in SGX. Hence, the introduced remote communication between the normal and secure worlds is the performance bottleneck of trusted execution.

The memory consumption profiling helps determine which functions can be run in the secure world. Based on the profiled memory consumed, developers can increase the size of TEE's shared memory. For example, if the TEE's memory size is limited to $10 * 1024$ bytes, and the MD5's `char` pointer parameter requires 20000 bytes, to run MD5 in the secure world requires modifying the TEE hardware configuration. The PX4 CPI functions (i.e., airspeed and next_waypoint), which perform numeric computations, require limited memory (i.e., for the `double` / `float` parameters / variables).

9. Discussion

In this section, we first discuss the limitations of TEE implementations and RT-Trust. Then after comparing the OP-TEE with the SGX, we discuss their most suitable usage scenarios.

9.1. Limitations

TEE Limitations. Table 4 shows the limitations of the OP-TEE and the SGX. For language support, the trusted part for the OP-TEE can only be written in C; that for the SGX can be written in both C and C++, while the communication channel between the trusted and untrusted parts can be written only in C. For memory allocation, the OP-TEE has no fixed size limit, with the upper bound becoming the amount of physical memory. In contrast, the maximum size of the SGX's protected memory is limited by the system BIOS with 64MB or 128MB as the typical value. Besides, neither the OP-TEE nor the SGX provides any support for multi-threading in the secure world. That is, one cannot spawn a thread (e.g., by using `pthreads`) inside the secure world. Furthermore, both TEEs re-implement their special versions of the standard system and C/C++ libraries. For example, the `printf` implementation of the OP-TEE cannot print `float` or `double` values. Similarly, the SGX provides neither `strcpy` nor `strcat`, instead requiring that developers use the provided `strncpy` and `strncat` instead [41].

RT-Trust Limitations. For OP-TEE, consider the scenario of passing a struct pointer to the specified function. The struct pointer is a linked list that has 100 elements. Each element has a `char` pointer as the data field. In that case, developers need to modify more than 100 ULOC in the generated RPC

Table 4: TEE Limitations

Limitations	OP-TEE	SGX
Language	C	C/C++
Memory	no limit	hard limit
Threading	no	no
Sys./lang. APIs	special version	special version

stubs to allocate the correct memory size for the marshaling and unmarshaling operations. In other words, the more complex pointer-based data structures are, the greater the programming effort is required to adapt generated code. Thus, the utility of RT-Trust diminishes rapidly for refactoring functions with complex pointer parameters.

For the SGX, RT-Trust requires that developers write specialized logic to marshal/unmarshal such complex pointer parameters. If the size of a pointer-based parameter happens to be larger than the limit set by the system BIOS, developers need to do extra work. First, modify the source code to divide the parameter data into several smaller parts and then write the required code to marshal/unmarshal the divided data to be transferred across the normal and secure worlds.

For both OP-TEE and SGX, RT-Trust restricts CPI functions from having function pointer parameters. Further, RT-Trust rejects the refactoring requests in which a CPI function assigns function pointers within its body. By inspecting the `AllocaInst` instructions during the static analysis phase, RT-Trust locates function pointers in the bodies of CPI functions. Upon detecting the presence of a function pointer, RT-Trust raises a partition failure. Besides, sometimes dynamically allocated objects can significantly differ in size depending on input. Hence, systems must be profiled with typical input parameters.

Table 5: FPI of OP-TEE and SGX

Algorithm	OP-TEE	SGX
CRC32	0.982	0.973
PC1	0.581	0.6
RC4	0.142	0.435
MD5	0.218	0.205
DES	0.756	0.803

9.2. Choosing between OP-TEE or SGX

Table 5 shows each micro-benchmark's Function Performance Indicator (FPI) for the OP-TEE and the SGX. Overall, the FPI values are comparable for both TEEs in all benchmarks. The faster the execution before moving to the TEE, the larger the FPI value (i.e., more performance degradation). The reason is that if a function runs fast (e.g., 1.15 ms for CRC32), the additional costs of the communication channel (i.e., 253.17 ms for CRC 32) dominate the total execution time. Another concern is the execution latencies in the secure world. In the case of RC4, moving the CPI functions to the SGX doubles their execution time. However, after moving the same functions to the OP-TEE, the execution time stays similar (as shown in Table 3). Hence, RC4's FPI for the SGX (i.e., 0.435) is larger than that for the OP-TEE (i.e., 0.142). To sum up, developers should always use the TEE with the smallest FPI value. However, if a CPI function's execution time is much smaller than the time taken by the communication channel, then both the OP-TEE and the SGX impose a comparable high-performance degradation.

10. Related Work

RT-Trust is related to DSLs for real-time systems, execution profiling, application partitioning, and code refactoring for trusted execution.

DSLs for real-time systems: Real Time Logic (RTL) formalizes real-time execution properties [42]. Subsequent DSLs for real-time systems include Hume that helps ensure that resource-limited, real-time systems meet execution constraints [43]. Flake et al. [44] add real-time constraints to the Object Constraint Language (OCL). Several efforts extend high-level programming languages to meet real-time execution requirements [45, 46, 47]. RT-Trust's RTTAs can also be seen as a declarative DSL for real-time constraints, albeit to be maintained when the original real-time system is refactored to protect its CPI functionality.

Execution Profiling: Several existing dynamic profiling tools, such as Pin tool [8], gperftools [9], and Gprof [48], ascertain program performance behavior. However, Pin and gperftools require that developers manually add profiling probes. Further, to profile program in TEE, one would have to pre-deploy their dependent libraries, which may be incompatible with particular TEE implementations. RT-Trust differs by automatically inserting profiling probes into the specified functions. Further, it estimates TEE-based execution characteristics without any pre-deployment.

Application Partitioning: J-Orchestra partitions the Java bytecode of a centralized application into a distributed application [49]. Given programmer annotations, Swift transforms a web application into a secure web application, in which the server-side Java part and the client-side JavaScript part interact with each other via HTTP [50]. ZØ compiles annotated C# code of a centralized application into a distributed multi-tier version to improve confidentiality and integrity, as directed by an automatically produced zero-knowledge proof of knowledge [51]. By enforcing a dynamic information flow control mechanism, Fission automatically and securely splits a JavaScript program into the client and server parts [52]. Pyxis automatically partitions database-backed applications into the application server and database parts [53]. Yang et al. optimize the code partitioning of mobile data stream applications [54].

Code refactoring for trusted execution: PtrSplit partitions C-language systems, while automatically tracking pointer bounds, thus enabling the automatic marshaling and unmarshaling of pointer parameters in RPC communication [3]. Senier et al. present a toolset that separates security protocols into several isolated partitions to fulfill security requirements [55]. Rubinov et al. leverage taint analysis to automatically partition Android applications for trusted execution [56]. TZSlicer automatically detects and slices away sensitive code fragments [57]. Lind et al.'s source-to-source transformation framework extracts subsets of C programs to take advantage of Intel SGX enclaves [58].

As compared with these works, RT-Trust not only supports the correct and automatic partitioning of legacy C code, but it also takes the real-time performance implications of the partitioning into account. By means of its profiling infrastructure and the *FPI* metric, RT-Trust predicts the degree to which a requested partitioning would decrease the system's real-time performance and also informs developers how to select between TEE implementations.

11. Future Work and Conclusion

One future work direction is to reduce the programmer effort required to provide the code for marshaling and unmarshaling complicated struct pointers with unknown bounds information. Another direction in this area is to automatically detect which functions are CPI-dependent and need to be protected in the secure world. Finally, we plan to experiment with symbolic analysis as another way of estimating the performance of refactored systems.

We have presented RT-Trust that provides a fully declarative meta-programming model with RTTA, static and dynamic analyses for determining whether the suggested partitioning strategy is reasonable, and whether the partitioned system would comply with the original real-time constraints, and an automated refactoring that transforms the original system while generating custom RPC communication and exception handling code. Our approach automatically refactors real-time systems with CPI-dependent functions for trusted execution under real-time constraints. The evaluation results of applying RT-Trust to micro-benchmarks and a drone autopilot indicate the promise of declarative meta-programming as a means of reducing the programmer effort required to isolate CPI under real-time constraints.

Acknowledgements

The research is supported by the NSF through the grants #1650540 and #1717065.

References

[1] OP-TEE, Open portable trusted execution environment, https://www.op-tee.org/ (2018).

[2] V. Costan, S. Devadas, Intel SGX explained., IACR Cryptology ePrint Archive 2016 (086) (2016) 1–118.

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

[4] GlobalPlatform Device Technology, TEE internal core API specification, https://www.globalplatform.org/specificationsdevice.asp (June 2016).

[5] GlobalPlatform Device Technology, Trusted user interface API, https://www.globalplatform.org/specificationsdevice.asp (June 2013).

[6] GlobalPlatform Device Technology, TEE client API specification, https://www.globalplatform.org/specificationsdevice.asp (June 2010).

[7] Intel, Intel software guard extensions (Intel SGX) SDK for linux, https://download.01.org/intel-sgx/linux-2.2/docs/Intel_SGX_Developer_Reference_Linux_2.2_Open_Source.pdf (2018).

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: Acm sigplan notices, Vol. 40, ACM, 2005, pp. 190–200.

[9] Google Inc., gperftools, https://github.com/gperftools/gperftools (2018).

[10] Y. Liu, K. An, E. Tilevich, RT-trust: automated refactoring for trusted execution under real-time constraints, in: Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, ACM, 2018, pp. 175–187.

[11] Department of Defense, Critical program information (CPI) identification and protection within research, development, test, and evaluation (RDT & E), http://www.secnav.navy.mil/ig/Lists/Instructions%20Links/DispForm.aspx?ID=15 (2015).

[12] GlobalPlatform, GlobalPlatform, TEE system architecture, technical report, https://www.globalplatform.org/specificationsdevice.asp (2011).

[13] P.-A. Hsiung, Real-time constraints, in: Institute of Information Science, Academia Sinica, Taipei, 2001.

[14] C. L. Liu, J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Journal of the ACM (JACM) 20 (1) (1973) 46–61.

[15] A. K. Reddy, P. Paramasivam, P. B. Vemula, Mobile secure data protection using emmc rpmb partition, in: Computing and Network Communications (CoCoNet), 2015 International Conference on, IEEE, 2015, pp. 946–950.

[16] CVE - Common Vulnerabilities and Exposures, https://cve.mitre.org/ (2019).

[17] CVE-2017-13997, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13997 (2017).

[18] CVE-2017-12733, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-12733 (2017).

[19] CVE-2018-8922, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-8922 (2018).

[20] CVE-2018-1219, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1219 (2018).

[21] CVE-2017-7493, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7493 (2017).

[22] CVE-2018-6412, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6412 (2018).

[23] CVE-2016-9103, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9103 (2016).

[24] CVE-2015-8944, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8944 (2015).

[25] CVE-2017-5239, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5239 (2017).

[26] CVE-2017-17672, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17672 (2017).

[27] CVE-2017-1500, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1500 (2017).

[28] CVE-2017-6094, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6094 (2017).

[29] CVE-2017-2704, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-2704 (2017).

[30] The Clang Team, Attributes in Clang, https://clang.llvm.org/docs/AttributeReference.html (2018).

[31] GNU, Using the GNU compiler collection (GCC), http://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html (2018).

[32] J. M., I. R., Intel software guard extensions part 3: Design an application, https://software.intel.com/en-us/articles/software-guard-extensions-tutorial-series-part-3 (2016).

[33] S. Nagarakatte, J. Zhao, M. M. Martin, S. Zdancewic, Softbound: Highly compatible and complete spatial memory safety for C, ACM Sigplan Notices 44 (6) (2009) 245–258.

[34] Intel, The Edger8r tool, https://software.intel.com/en-us/sgx-sdk-dev-reference-the-edger8r-tool (2018).

[35] W. Tansey, E. Tilevich, Efficient automated marshaling of C++ data structures for MPI applications, in: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, IEEE, 2008, pp. 1–12.

[36] S. Vinoski, CORBA: integrating diverse applications within distributed heterogeneous environments, IEEE Communications magazine 35 (2) (1997) 46–55.

[37] Google Inc., gRPC a high performance, open-source universal RPC framework, https://grpc.io (2017).

[38] Intel, Enclave definition language file syntax, https://software.intel.com/en-us/sgx-sdk-dev-reference-enclave-definition-language-file-syntax (2018).

[39] Mirror of official LLVM git repository, https://github.com/llvm-mirror/test-suite (2018).

[40] PX4 Dev Team, PX4, http://px4.io/ (2018).

[41] Intel, Intel software guard extensions SDK - string functions, https://software.intel.com/en-us/sgx-sdk-dev-reference-string-functions (2018).

[42] F. Jahanian, A. Goyal, A formalism for monitoring real-time constraints at run-time, in: Digest of Papers. Fault-Tolerant Computing: 20th International Symposium, IEEE, 1990, pp. 148–155.

[43] K. Hammond, G. Michaelson, Hume: a domain-specific language for real-time embedded systems, in: International Conference on Generative Programming and Component Engineering, Springer, 2003, pp. 37–56.

[44] S. Flake, W. Mueller, An OCL extension for real-time constraints, in: Object Modeling with the OCL, Springer, 2002, pp. 150–171.

[45] Y. Ishikawa, H. Tokuda, Object-oriented real-time language design: Constructs for timing constraints, Vol. 25, ACM, 1990.

[46] G. Bollella, J. Gosling, The real-time specification for Java, Computer 33 (6) (2000) 47–54.

[47] N. Gehani, K. Ramamritham, Real-time Concurrent C: A language for programming dynamic real-time systems, Real-Time Systems 3 (4) (1991) 377–405.

[48] S. L. Graham, P. B. Kessler, M. K. Mckusick, Gprof: A call graph execution profiler, in: ACM Sigplan Notices, Vol. 17, ACM, 1982, pp. 120–126.

[49] E. Tilevich, Y. Smaragdakis, J-orchestra: Automatic Java Application Partitioning, in: European conference on object-oriented programming, Springer, 2002, pp. 178–204.

[50] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, X. Zheng, Secure web applications via automatic partitioning, ACM SIGOPS Operating Systems Review 41 (6) (2007) 31–44.

[51] M. Fredrikson, B. Livshits, Zø: an optimizing distributing zero-knowledge compiler, in: Proceedings of the 23rd USENIX conference on Security Symposium, USENIX Association, 2014, pp. 909–924.

[52] A. Guha, J.-B. Jeannin, R. Nigam, J. Tangen, R. Shambaugh, Fission: Secure dynamic code-splitting for JavaScript, in: 2nd Summit on Advances in Programming Languages (SNAPL 2017), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[53] A. Cheung, S. Madden, O. Arden, A. C. Myers, Automatic partitioning of database applications, Proceedings of the VLDB Endowment 5 (11) (2012) 1471–1482.

[54] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, A. Chan, A framework for partitioning and execution of data stream applications in mobile cloud computing, ACM SIGMETRICS Performance Evaluation Review 40 (4) (2013) 23–32.

[55] A. Senier, M. Beck, T. Strufe, Prettycat: Adaptive guarantee-controlled software partitioning of security protocols, arXiv preprint arXiv:1706.04759.

[56] K. Rubinov, L. Rosculete, T. Mitra, A. Roychoudhury, Automated partitioning of Android applications for trusted execution environments, in: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, IEEE, 2016, pp. 923–934.

[57] M. Ye, J. Sherman, W. Srisa-an, S. Wei, Tzslicer: Security-aware dynamic program slicing for hardware isolation, in: 2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), IEEE, 2018, pp. 17–24.

[58] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, et al., Glamdring: Automatic application partitioning for Intel SGX, in: USENIX ATC, 2017.