RT-Trust: Automated Refactoring for Trusted Execution Under Real-Time Constraints

Yin Liu, Kijin An, and Eli Tilevich
Software Innovations Lab
Virginia Tech
{yinliu,ankijin,tilevich}@cs.vt.edu

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 the regular 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 for the 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. 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

ACM Reference Format:
Yin Liu, Kijin An, and Eli Tilevich. 2018. RT-Trust: Automated Refactoring for Trusted Execution Under Real-Time Constraints. In *Proceedings of 17th International Conference on Generative Programming: Concepts & Experience (GPCE'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.475/123_4

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, e.g., 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 crisis situations. 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 properly adjust its trajectory 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 the navigation module 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.

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 TEE. Adapting existing real-time systems to use TEE

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

GPCE'18, November 2018, Boston, Massachusetts, United States

© 2018 Copyright held by the owner/author(s).

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

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. To ascertain these dependencies requires deep familiarity with the source code. 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 [29]. 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 needs to manually write the communication logic required for the normal and secure worlds to talk to each other, correctly applying suitable TEE APIs that establish customized remote procedure calls (RPC). However, the TEE provides more than 130 APIs and about 40 data types [39–41], requiring a great deal of time to learn and master. 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 [30] and gperftools [24], 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 paper presents RT-Trust, a program analysis and transformation toolset that supports developers in partitioning C-language systems in the presence of real-time constraints. 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. 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 RPC communication 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 three-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, 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 RPC communication and real-time deadline violation handling.

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.

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 evaluation. Section 8 discusses related work. Section 9 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 [32], 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 [19] 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). 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 [33] 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.

2.3 Real-Time Constraints

In general, real-time constraints [23] 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 [28]. 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 [34] 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, the drone’s navigation algorithm must respond on time (hard constraint), otherwise, it will lose its way or crash. In contrast, when watching a movie online, the delay in network transfer can be accepted, as it will not crash the video player (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). First, without a proper access control and authentication mechanism for critical functions, attackers can maliciously access and consume the significant amount of resources [3, 4, 10, 11, 13]. Secondly, the possibility of information leakage sharply rises by the vulnerable critical functions [1, 2, 12], especially the functions processing sensitive data. For example, by compromising the data transmitting process, attackers maliciously obtain the current GPS locations [8]. In addition, arbitrarily exposing critical functions for interaction with external actors can be illegally exploited, which causes file deletion [6] or credential disclosure [5]. Further, reverse engineering can disclose critical algorithms [9] or expose sensitive data (e.g., the encryption keys) [7].

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 & output of RT-Trust.

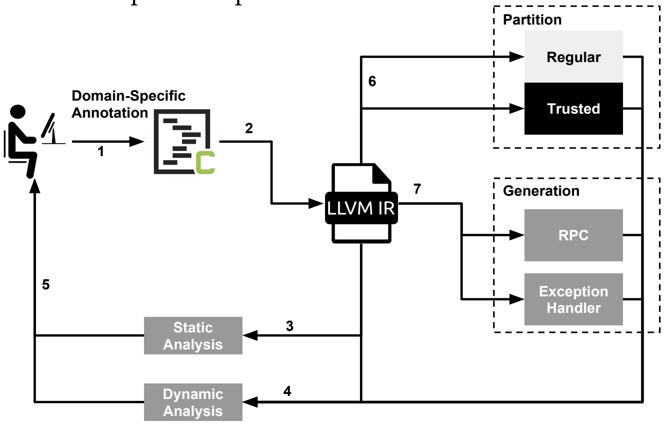

Figure 1. The RT-Trust Process

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). 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 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 customized RPCs, including marshaling and unmarshaling logic. 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 RPC communication. All generated code can be further customized by hand if necessary. The transformed IR code, generated source code (i.e., RPC client stub), and referenced libraries (e.g., encryption, profiling, etc.) are eventually linked with the normal world's executable. Similarly, for the secure world, the trusted IR, RPC server stub, 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 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 [38], none of them are designed for real-time systems and TEE.

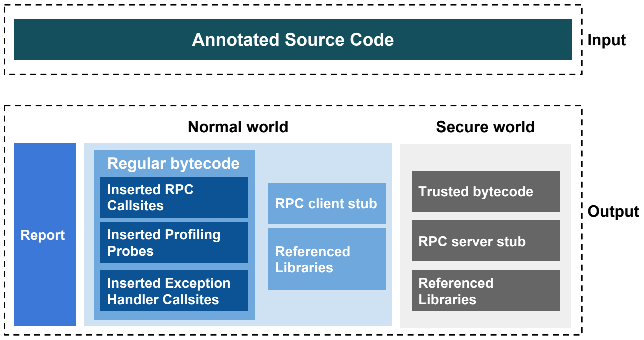

Figure 2. RT-Trust's Input & Output

For our meta-programming model, we design and implement a set of domain-specific annotations that describe the real-time constraints, code transformation & 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 [20], one of the general syntaxes supported by Clang. The form of attribute specifier is **__attribute__**((annotation-list)). The annotation list is a sequence of annotations separated by commas. Each annotation contains the annotation name and a parenthesized arguments list. An arguments list is a possibly empty comma-separated sequence of arguments.

```
__attribute__((annotation-list))
annotation-list ::= annotation_1, ..., annotation_n
annotation ::= name (argument-list)
argument-list ::= argument_1, ..., argument_n
```

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))
subjects ::= Function
```

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 options of "yes" and "no" specify whether to encrypt or compress the data transferred between the partitions. 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 RPC communication mechanism. 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 [29], 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 includes a handler function's name, and after how many violations of the real-time constraints it should be triggered. 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
subjects ::= Function

__attribute__((paramlen(length)))
length ::= n (n is integer, n > 0)
subjects ::= Variable and Parameter;

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

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)
subjects ::= Function

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

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

4.5 RTTA Dependencies

As compared to the annotations that can be specified independently, such as **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 RPCs. 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 byte 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

RT-Trust determines whether a given partitioning scenario can be realized, as specified by the annotated functions, by checking the following two rules that we call "zigzag" and "global variable." RT-Trust checks these rules in turn to immediately identify and report those cases when a specified partitioning request cannot be fulfilled.

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 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.

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.

Partitionable Function Graph To check the above rules, RT-Trust introduces a partitionable function graph (PFG).

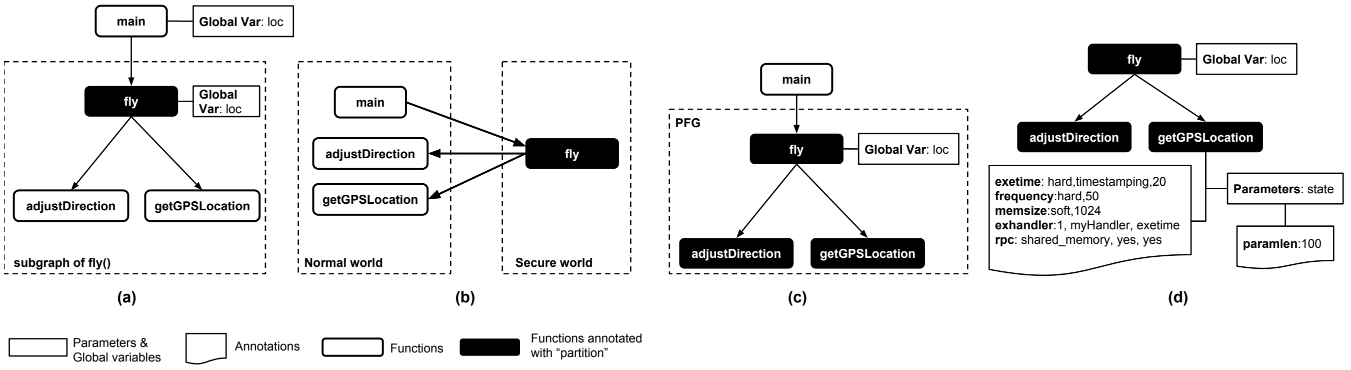

Figure 3. The RT-Trust PFG

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 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, annotations, etc.) required to partition the system. Our deliberate design choice is 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)), which 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 impossible to guarantee whether the profiled execution would produce 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.

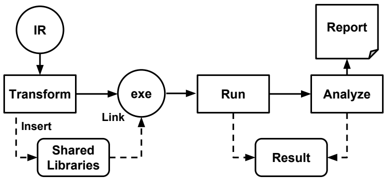

Figure 4. The RT-Trust Analyses Procedure

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[1], 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 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 [31]. 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,

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

and inserts the invocation to the original function into the "else" block as follows:

```
Ret = function(Args); // is transforms  to :
    if (t reaches threshold)
      Ret = exhandling_function(Args);
    else
      Ret = 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 Customizable RPC Communication Generation and Transformation

The partitioning process entails dividing the original IR program into two partitions: trusted and regular. The former contains the partitioned CPI-dependent functions to be put to execute in the secure world. The latter contains the remaining functions to keep executing in the normal world. To enable these two partitions to communicate with each other, 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. Then, RT-Trust transforms the functions in the regular partition, redirecting local function invocations of the CPI function to the invocations of the corresponding RPC stub.

6.1 Generating RPC stubs

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 & 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 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 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. Since CPI functions and their RPC stubs share the same signature, no other changes are necessary:

```
Ret = original_function(Args); // is transformed into:
Ret = RPC_function(Args);
```

Now, the original function call is redirected to RPC call to invoke the corresponding partitioned function in the secure world. Recall the transformation of exception handling in Section 5.3, in that case, developers can specify the original function as the exception handling function. 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:
Ret = (reach threshold) ? original_function(Args)
                        : RPC_function(Args);
```

7 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?

7.1 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 & 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. 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.

Microbenchmarks & 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 OP-TEE supports 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 [14].

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) [37]: airspeed and waypoint computations.

Evaluation Design As described in Section 5 and 6, developers can customize the implementations of profiling 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 may need to be manually adjusted). Since without RT-Trust, the programmer would have to write all the code by hand, we measure how much effort is saved by using RT-Trust.

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

7.2 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)

Alg.	RTTAs	Generate & Transform	Adjust
CRC32	5	388	0
PC1	4	344	6
RC4	3	292	3
MD5	3	364	3
DES	2	244	15

Effort Table 1 shows the effort saved by applying RT-Trust. Generally, the total number of ULOC automatically generated & transformed by RT-Trust (244 ~ 388 ULOC) greatly

Table 2. Value & Accuracy of RT-Trust

Algorithm	RPC (ms)	Execution Time (ms)		Invocation Interval (ms)		Memory (byte)	
		Before	After	Before	After	Parameter	Local
CRC32	253.17	1.15	1.30	1.24	269	40	92
PC1	273.38	68.22	13	68.10	314	32	22
RC4	236.96	500.52	447	506.95	705	240	1144
MD5	177.83	267.43	254	267.62	446	20000	316
DES	201.99	24.18	32	24.30	224	528	72
PX4 - airspeed	256.35	≈ 0	≈ 0	50.16	305	12	12
PX4 - next_waypoint	264.96	0.40	0.46	500.75	773.67	40	40

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 ULOC 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.

Table 3. Overhead of RT-Trust profiling (ms)

Exec. Time		Invocation Intvl.		Memory	
Normal	Secure	Normal	Secure	Parameter	Local
0.442	144	0.418	139	0.051	0.053

Performance Table 3 reports on the overhead of RT-Trust profiling, which captures and calculates the execution time, invocation interval, 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. Hardware environments heavily impact the profiling overhead, with an order of magnitude difference: $\approx 0.4ms$ in the normal worlds vs. $\approx 140ms$ in the secure world. This drastic performance difference is due to the Linux system calls using in the normal world (e.g., `gettimeofday`) being greatly more efficient than the TEE APIs (e.g., `TEE_GetREETime`). 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 $\approx 0.053 * n$. 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.

Value & Accuracy Table 2 shows the results of profiling the CPI functions, with the profiling overhead subtracted. 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 RPC communication 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 is below this deadline, moving this CPI function to TEE preserves its real-time constraints.

RPC communication delays the invocation interval of our micro-benchmarks and the CPI PX4 functions. For the micro-benchmarks, due to invoking each function in a loop, the next invocation immediately follows the previous invoke. Thus, in the normal world, each function's invocation interval ("Before" column of "Execution Time") is similar to its execution time ("Before" column of "Invocation Interval"). However, in the secure world, these invocation intervals become very long, which is similar to the time consumed by RPC communication ("RPC" column) plus the time in the secure world

("After" column of "Execution Time"). For the PX4 autopilot, which computes airspeed and next waypoint every 50ms and 500ms, respectively, the RPC communication delays these invocation intervals to 305ms ($\approx 256.35(RPC) + 0$(execution time) +50) and 773.67ms ($\approx 264.96(RPC) + 0.46$(execution time) +500). Hence, RPC communication 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).

Limitations 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 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. In addition, sometimes dynamically allocated objects can greatly differ in size depending on input. Hence, systems must be profiled with typical input parameters.

8 Related Work

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

DSLs for real-time systems: Real Time Logic (RTL) formalizes real-time execution properties [26]. Subsequent DSLs for real-time systems include Hume that helps ensure that resource-limited, real-time systems meet execution constraints [22]. Flake et al. [16] add real-time constraints to the Object Constraint Language (OCL). Several efforts extend high-level programming languages to meet real-time execution requirements [15, 18, 25]. The RTTAs of RT-Trust can also be considered a declarative DSL for real-time constraints, albeit to be maintained when the original real-time system is refactored to protect its CPI functionality.

Code refactoring for trusted execution: 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 [17]. PtrSplit partitions C-language systems, while automatically tracking pointer bounds, thus enabling the automatic marshaling and unmarshaling of pointer parameters in RPC communication [29]. Senier et al. present a toolset that separates security protocols into several isolated partitions to fulfill security requirements [36]. Rubinov et al. leverage taint analysis to automatically partition Android applications for trusted execution [35]. TZSlicer automatically detects and slices away sensitive code fragments [42]. Lind et al. present a source-to-source code transformation framework that extracts subsets of C programs to take advantage of Intel SGX enclaves [27]. RT-Trust differs in its ability to protect CPI under real-time constraints. RT-Trust not only transforms code, but also applies static and dynamic analyses to determine the validity of a partitioning plan and its likelihood of meeting the original real-time constraints.

Execution Profiling: Several existing dynamic profiling tools, such as Pin tool [30], gperftools [24], and Gprof [21], 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.

9 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 authors would like to thank Dr. Thidapat (Tam) Chantem, Zheng "Jason" Song, and the anonymous reviewers, whose insightful comments helped improve the technical content of this paper. The research is supported by the NSF through the grant # 1650540.

References

[1] 2015. CVE-2015-8944. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8944

[2] 2016. CVE-2016-9103. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9103

[3] 2017. CVE-2017-12733. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-12733

[4] 2017. CVE-2017-13997. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13997

[5] 2017. CVE-2017-1500. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1500

[6] 2017. CVE-2017-17672. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17672

[7] 2017. CVE-2017-2704. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-2704

[8] 2017. CVE-2017-5239. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5239

[9] 2017. CVE-2017-6094. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6094

[10] 2017. CVE-2017-7493. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7493

[11] 2018. CVE-2018-1219. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1219

[12] 2018. CVE-2018-6412. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6412

[13] 2018. CVE-2018-8922. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-8922

[14] 2018. Mirror of official llvm git repository. https://github.com/llvm-mirror/test-suite

[15] Gregory Bollella and James Gosling. 2000. The real-time specification for Java. *Computer* 33, 6 (2000), 47–54.

[16] Stephan Flake and Wolfgang Mueller. 2002. An OCL extension for real-time constraints. In *Object Modeling with the OCL*. Springer, 150–171.

[17] Matthew Fredrikson and Benjamin Livshits. 2014. ZØ: an optimizing distributing zero-knowledge compiler. In *Proceedings of the 23rd USENIX conference on Security Symposium*. USENIX Association, 909–924.

[18] Narain Gehani and Krithi Ramamritham. 1991. Real-time concurrent C: A language for programming dynamic real-time systems. *Real-Time Systems* 3, 4 (1991), 377–405.

[19] GlobalPlatform. 2011. GlobalPlatform, TEE System Architecture, Technical Report. https://www.globalplatform.org/specificationsdevice.asp

[20] GNU. 2018. Using the GNU Compiler Collection (GCC). http://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html

[21] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. 1982. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, Vol. 17. ACM, 120–126.

[22] Kevin Hammond and Greg Michaelson. 2003. Hume: a domain-specific language for real-time embedded systems. In *International Conference on Generative Programming and Component Engineering*. Springer, 37–56.

[23] Pao-Ann Hsiung. 2001. Real-Time Constraints. In *Institute of Information Science, Academia Sinica, Taipei*.

[24] Google Inc. 2018. gperftools. https://github.com/gperftools/gperftools

[25] Yutaka Ishikawa and Hideyuki Tokuda. 1990. *Object-oriented real-time language design: Constructs for timing constraints*. Vol. 25. ACM.

[26] Farnam Jahanian and Ambuj Goyal. 1990. A formalism for monitoring real-time constraints at run-time. In *Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*. IEEE, 148–155.

[27] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, P Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic application partitioning for Intel SGX. USENIX.

[28] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

[29] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2359–2371.

[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[31] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices* 44, 6 (2009), 245–258.

[32] United States. Department of Defense. 2015. 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

[33] OP-TEE. 2018. Open Portable Trusted Execution Environment. https://www.op-tee.org/

[34] Anil Kumar Reddy, Periyasamy Paramasivam, and Prakash Babu Vemula. 2015. Mobile secure data protection using eMMC RPMB partition. In *Computing and Network Communications (CoCoNet), 2015 International Conference on*. IEEE, 946–950.

[35] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. 2016. Automated partitioning of android applications for trusted execution environments. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 923–934.

[36] Alexander Senier, Martin Beck, and Thorsten Strufe. 2017. PrettyCat: Adaptive guarantee-controlled software partitioning of security protocols. *arXiv preprint arXiv:1706.04759* (2017).

[37] PX4 Dev Team. 2018. PX4. http://px4.io/

[38] The Clang Team. 2018. Attributes in Clang. https://clang.llvm.org/docs/AttributeReference.html

[39] GlobalPlatform Device Technology. June 2010. TEE Client API Specification. https://www.globalplatform.org/specificationsdevice.asp

[40] GlobalPlatform Device Technology. June 2013. Trusted User Interface API. https://www.globalplatform.org/specificationsdevice.asp

[41] GlobalPlatform Device Technology. June 2016. TEE Internal Core API Specification. https://www.globalplatform.org/specificationsdevice.asp

[42] Mengmei Ye, Jonathan Sherman, Witawas Srisa-an, and Sheng Wei. 2018. TZSlicer: Security-aware dynamic program slicing for hardware isolation. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 17–24.